Hypothesis Documentation

Release 3.5.3

David R. MacIver

October 05, 2016

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 the 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, or check out some of the introductory articles.

CHAPTER 1

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):

```
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.

Lets 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
```

Suppose we had a more interesting bug and forgot to reset the count each time. Say we missed a line in our `encode` method:

```
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  # Missing reset operation
            prev = character
        else:
            count += 1
    else:
        entry = (character, count)
        lst.append(entry)
    return lst
```

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

```
easy_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:

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

We could also have done this as a unittest TestCase:

```python
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 parameterizing 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.

CHAPTER 2

Hypothesis for Django users

Hypothesis offers a number of features specific for Django testing, available in the `hypothesis[django]` extra.

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 eachother). 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:

```
>>> 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\U000519b0\U0003fabf\U000f1910
>>> c.age
-873375803
```

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

Obviously the customer's age is implausible, so lets fix that:

```
>>> 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.

Foreign keys are not automatically derived. If they're nullable they will default to always being null, otherwise you always have to specify them. e.g. suppose we had a Shop type with a foreign key to company, we would define a strategy for it as:

```
shop_strategy = models(Shop, company=models(Company))
```

2.1 Tips and tricks

2.1.1 Custom field types

If you have a custom Django field type you can register it with Hypothesis's model deriving functionality by registering a default strategy for it:

```
>>> from toystore.models import CustomishField, Customish
>>> models(Customish).example()
hypothesis.errors.InvalidArgument: Missing arguments for mandatory field
    customish for model Customish
>>> 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.

2.1.2 Generating child models

For the moment there's no explicit support in hypothesis-django for generating dependent models. i.e. a Company model will generate no Shops. However if you want to generate some dependent models as well, you can emulate this by using the *flatmap* function as follows:

```
from hypothesis.strategies import lists, just

def generate_with_shops(company):
  return lists(models(Shop, company=just(company))).map(lambda _: company)

company_with_shops_strategy = models(Company).flatmap(generate_with_shops)
```

Lets unpack what this is doing:

The way flatmap works is that we draw a value from the original strategy, then apply a function to it which gives us a new strategy. We then draw a value from *that* strategy. So in this case we're first drawing a company, and then we're drawing a list of shops belonging to that company: The *just* strategy is a strategy such that drawing it always produces the individual value, so `models(Shop, company=just(company))` is a strategy that generates a Shop belonging to the original company.

So the following code would give us a list of shops all belonging to the same company:

```
models(Company).flatmap(lambda c: lists(models(Shop, company=just(c))))
```

The only difference from this and the above is that we want the company, not the shops. This is where the inner map comes in. We build the list of shops and then throw it away, instead returning the company we started for. This works because the models that Hypothesis generates are saved in the database, so we're essentially running the inner strategy purely for the side effect of creating those children in the database.

2.1.3 Using default field values

Hypothesis ignores field defaults and always tries to generate values, even if it doesn't know how to. You can tell it to use the default value for a field instead of generating one by passing `fieldname=default_value` to `models()`:

```
>>> from toystore.models import DefaultCustomish
>>> models(DefaultCustomish).example()
hypothesis.errors.InvalidArgument: Missing arguments for mandatory field
    customish for model DefaultCustomish
>>> from hypothesis.extra.django.models import default_value
>>> x = models(DefaultCustomish, customish=default_value).example()
>>> x.customish
'b'
```

Details and advanced features

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.

3.1 Additional test output

Normally the output of a failing test will look something like:

```
Falsifying example: test_a_thing(x=1, y="foo")
```

With the `repr` of each keyword argument being printed.

Sometimes this isn't enough, either because you have values with a `repr` that isn't very descriptive or because you need to see the output of some intermediate steps of your test. That's where the `note` function comes in:

```
>>> from hypothesis import given, note, strategies as st
>>> @given(st.lists(st.integers()), st.randoms())
... def test_shuffle_is_noop(ls, r):
...     ls2 = list(ls)
...     r.shuffle(ls2)
...     note("Shuffle: %r" % (ls2))
...     assert ls == ls2
...
>>> test_shuffle_is_noop()
Falsifying example: test_shuffle_is_noop(ls=[0, 0, 1], r=RandomWithSeed(0))
Shuffle: [0, 1, 0]
Traceback (most recent call last):
    ...
AssertionError
```

The note is printed in the final run of the test in order to include any additional information you might need in your test.

3.2 Test Statistics

If you are using py.test you can see a number of statistics about the executed tests by passing the command line argument –hypothesis-show-statistics. This will include some general statistics about the test:

For example if you ran the following with –hypothesis-show-statistics:

```
from hypothesis import given, strategies as st

@given(st.integers())
def test_integers(i):
    pass
```

You would see:

```
test_integers:

  - 200 passing examples, 0 failing examples, 0 invalid examples
  - Typical runtimes: < 1ms
  - Stopped because settings.max_examples=200
```

The final "Stopped because" line is particularly important to note: It tells you the setting value that determined when the test should stop trying new examples. This can be useful for understanding the behaviour of your tests. Ideally you'd always want this to be max_examples.

In some cases (such as filtered and recursive strategies) you will see events mentioned which describe some aspect of the data generation:

```
from hypothesis import given, strategies as st

@given(st.integers().filter(lambda x: x % 2 == 0))
def test_even_integers(i):
    pass
```

You would see something like:

```
test_even_integers:

  - 200 passing examples, 0 failing examples, 16 invalid examples
  - Typical runtimes: < 1ms
  - Stopped because settings.max_examples=200
  - Events:
    * 30.56%, Retried draw from integers().filter(lambda x: x % 2 == 0) to satisfy filter
    * 7.41%, Aborted test because unable to satisfy integers().filter(lambda x: x % 2 == 0)
```

You can also mark custom events in a test using the 'event' function:

```
from hypothesis import given, event, strategies as st

@given(st.integers().filter(lambda x: x % 2 == 0))
def test_even_integers(i):
    event("i mod 3 = %d" % (i % 3,))
```

You will then see output like:

```
test_even_integers:

  - 200 passing examples, 0 failing examples, 28 invalid examples
  - Typical runtimes: < 1ms
  - Stopped because settings.max_examples=200
  - Events:
    * 47.81%, Retried draw from integers().filter(lambda x: x % 2 == 0) to satisfy filter
    * 31.14%, i mod 3 = 2
    * 28.95%, i mod 3 = 1
    * 27.63%, i mod 3 = 0
    * 12.28%, Aborted test because unable to satisfy integers().filter(lambda x: x % 2 == 0)
```

Arguments to event() can be any hashable type, but two events will be considered the same if they are the same when converted to a string with str().

3.3 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:

```
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:

```
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:

```
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 have got the exception:

```
Unsatisfiable: Unable to satisfy assumptions of hypothesis test_negation_is_self_inverse_for_non_nan
```

3.3.1 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:

```
@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:

```
@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]
[6, 2, 29, 30, 25, 34, 19, 15, 50, 16, 10, 3, 16]
[25, 17, 9, 19, 15, 2, 2, 4, 22, 10, 10, 27, 3, 1, 14, 17, 13, 8, 16, 9, 2...
[17, 65, 78, 1, 8, 29, 2, 79, 28, 18, 39]
[13, 26, 8, 3, 4, 76, 6, 14, 20, 27, 21, 32, 14, 42, 9, 24, 33, 9, 5, 15, ...
[2, 1, 2, 2, 3, 10, 12, 11, 21, 11, 1, 16]
```

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.

3.4 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:

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

```
>>> 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:

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

Further details are available in a separate document.

3.5 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:

```
@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())
def d(x, y):
  pass

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

```
@given(x=integers(), y=integers())
def f(x, *args, **kwargs):
  pass


class SomeTest(TestCase):
    @given(integers())
    def test_a_thing(self, x):
        pass
```

The following are not:

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

@given(integers())
def h(x, *args):
    pass

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

@given()
def j(x, y):
    pass
```

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

1. You may pass any keyword argument to given.
2. Positional arguments to given are equivalent to the rightmost named arguments for the test function.
3. positional arguments may not be used if the underlying test function has varargs or arbitrary keywords.
4. Functions tested with given may not have any defaults.

The reason for the "rightmost named arguments" behaviour is so that using `@given` with instance methods works: self will be passed to the function as normal and not be parametrized over.

The function returned by given has all the arguments that the original test did , minus the ones that are being filled in by given.

3.6 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:

```
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:

```
from unittest import TestCase

class TestTryReallyHard(TestCase):
    @given(integers())
    def test_something(self, i):
        perform_some_unreliable_operation(i)

    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.

An executor must be able to handle being passed a function which returns None, otherwise it won't be able to run normal test cases. So for example the following executor is invalid:

```
from unittest import TestCase

class TestRunTwice(TestCase):
    def execute_example(self, f):
        return f()()
```

and should be rewritten as:

```
from unittest import TestCase
import inspect

class TestRunTwice(TestCase):
    def execute_example(self, f):
        result = f()
        if inspect.isfunction(result):
            result = result()
        return result
```

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.

3.7 Using Hypothesis to find values

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

```
>>> 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]
>>> find(sets(integers()), lambda x: sum(x) >= 10 and len(x) >= 3)
{0, 1, 9}
```

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:

```
>>> 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.NoSuchExample: No examples of condition lambda x: <unknown>
```

(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.

3.8 Providing explicit examples

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

```
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.

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:

```
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")
    def test_some_code(self, x):
        assert True
```

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.

CHAPTER 4

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 set up a @given based test to use this using a settings decorator:

`@given` invocation as follows:

```
from hypothesis import given, settings

@given(integers())
@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.

This may be applied either before or after the given and the results are the same. The following is exactly equivalent:

```
from hypothesis import given, settings

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

4.1 Available settings

class hypothesis.**settings**(*parent=None*, ***kwargs*)

A settings object controls a variety of parameters that are used in falsification. These may control both the falsification strategy and the details of the data that is generated.

Default values are picked up from the settings.default object and changes made there will be picked up in newly created settings.

database_file

database: An instance of hypothesis.database.ExampleDatabase that will be used to save examples to and load previous examples from. May be None in which case no storage will be used. default value: '/home/docs/checkouts/readthedocs.org/user_builds/hypothesis/checkouts/latest/docs/.hypothesis/examples'

database

An ExampleDatabase instance to use for storage of examples. May be None.

If this was explicitly set at settings instantiation then that value will be used (even if it was None). If not and the database_file setting is not None this will be lazily loaded as an SQLite backed ExampleDatabase using that file the first time this property is accessed on a particular thread.

`max_examples`
Once this many satisfying examples have been considered without finding any counter-example, falsification will terminate. default value: 200

`max_iterations`
Once this many iterations of the example loop have run, including ones which failed to satisfy assumptions and ones which produced duplicates, falsification will terminate. default value: 1000

`max_shrinks`
Once this many successful shrinks have been performed, Hypothesis will assume something has gone a bit wrong and give up rather than continuing to try to shrink the example. default value: 500

`min_satisfying_examples`
Raise Unsatisfiable for any tests which do not produce at least this many values that pass all assume() calls and which have not exhaustively covered the search space. default value: 5

`perform_health_check`
If set to True, Hypothesis will run a preliminary health check before attempting to actually execute your test. default value: True

`stateful_step_count`
Number of steps to run a stateful program for before giving up on it breaking. default value: 50

`strict`
If set to True, anything that would cause Hypothesis to issue a warning will instead raise an error. Note that new warnings may be added at any time, so running with strict set to True means that new Hypothesis releases may validly break your code.

You can enable this setting temporarily by setting the HYPOTHESIS_STRICT_MODE environment variable to the string 'true'. default value: False

`suppress_health_check`
A list of health checks to disable default value: []

`timeout`
Once this many seconds have passed, falsify will terminate even if it has not found many examples. This is a soft rather than a hard limit - Hypothesis won't e.g. interrupt execution of the called function to stop it. If this value is <= 0 then no timeout will be applied. default value: 60

4.1.1 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).

```
>>> 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
>>> @given(integers())
... @settings(verbosity=Verbosity.verbose)
... 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.

4.2 Building settings objects

settings can be created by calling settings with any of the available settings values. Any absent ones will be set to defaults:

```
>>> from hypothesis import settings
>>> settings()
settings(average_list_length=25.0, database_file='/home/david/projects/hypothesis/.hypothesis/example
>>> settings().max_examples
200
>>> settings(max_examples=10).max_examples
10
```

You can also copy settings off other settings:

```
>>> s = settings(max_examples=10)
>>> t = settings(s, max_iterations=20)
>>> s.max_examples
10
>>> t.max_iterations
20
>>> s.max_iterations
```

```
1000
>>> s.max_shrinks
500
>>> t.max_shrinks
500
```

4.3 Default settings

At any given point in your program there is a current default settings, available as settings.default. As well as being a settings object in its own right, all newly created settings objects which are not explicitly based off another settings are based off the default, so will inherit any values that are not explicitly set from it.

You can change the defaults by using profiles (see next section), but you can also override them locally by using a settings object as a context manager

```
>>> with settings(max_examples=150):
...     print(settings.default.max_examples)
...     print(settings().max_examples)
...
150
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:

```
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.

Warning: If you use define test functions which don't use @given inside a context block, these will not use the enclosing settings. This is because the context manager only affects the definition, not the execution of the function.

4.3.1 settings Profiles

Depending on your environment you may want different default settings. For example: during development you may want to lower the number of examples to speed up the tests. However, in a CI environment you may want more examples so you are more likely to find bugs.

Hypothesis allows you to define different settings profiles. These profiles can be loaded at any time.

Loading a profile changes the default settings but will not change the behavior of tests that explicitly change the settings.

```
>>> from hypothesis import settings
>>> settings.register_profile("ci", settings(max_examples=1000))
>>> settings().max_examples
200
```

```
>>> settings.load_profile("ci")
>>> settings().max_examples
1000
```

Instead of loading the profile and overriding the defaults you can retrieve profiles for specific tests.

```
>>> with settings.get_profile("ci"):
...     print(settings().max_examples)
...
1000
```

Optionally, you may define the environment variable to load a profile for you. This is the suggested pattern for running your tests on CI. The code below should run in a *conftest.py* or any setup/initialization section of your test suite. If this variable is not defined the Hypothesis defined defaults will be loaded.

```
>>> from hypothesis import settings
>>> settings.register_profile("ci", settings(max_examples=1000))
>>> settings.register_profile("dev", settings(max_examples=10))
>>> settings.register_profile("debug", settings(max_examples=10, verbosity=Verbosity.verbose))
>>> settings.load_profile(os.getenv(u'HYPOTHESIS_PROFILE', 'default'))
```

If you are using the hypothesis pytest plugin and your profiles are registered by your conftest you can load one with the command line option `--hypothesis-profile`.

```
$ py.test tests --hypothesis-profile <profile-name>
```

CHAPTER 5

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.`**`nothing`**`()`

This strategy never successfully draws a value and will always reject on an attempt to draw.

`hypothesis.strategies.`**`just`**`(`*value*`)`

Return a strategy which only generates value.

Note: value is not copied. Be wary of using mutable values.

`hypothesis.strategies.`**`none`**`()`

Return a strategy which only generates None.

`hypothesis.strategies.`**`one_of`**`(`**args*`)`

Return a strategy which generates values from any of the argument strategies.

This may be called with one iterable argument instead of multiple strategy arguments. In which case one_of(x) and one_of(*x) are equivalent.

`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*, *allow_nan=None*, *allow_infinity=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.
- If min_value or max_value is not None, it is an error to enable allow_nan.

- If both min_value and max_value are not None, it is an error to enable allow_infinity.

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.`**`sampled_from`**`(`*elements*`)`

Returns a strategy which generates any value present in the iterable elements.

Note that as with just, values will not be copied and thus you should be careful of using mutable data.

`hypothesis.strategies.`**`lists`**`(`*elements=None*, *min_size=None*, *average_size=None*, *max_size=None*, *unique_by=None*, *unique=False*`)`

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.

If unique is True (or something that evaluates to True), we compare direct object equality, as if unique_by was *lambda x: x*. This comparison only works for hashable types.

if unique_by is not None it must be a function returning a hashable type when given a value drawn from elements. The resulting list will satisfy the condition that for i != j, unique_by(result[i]) != unique_by(result[j]).

`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.

`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.

`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.

`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.

`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.

`hypothesis.strategies.`**`characters`**(*whitelist_categories=None*, *blacklist_categories=None*, *blacklist_characters=None*, *min_codepoint=None*, *max_codepoint=None*)

Generates unicode text type (unicode on python 2, str on python 3) characters following specified filtering rules.

This strategy accepts lists of Unicode categories, characters of which should (*whitelist_categories*) or should not (*blacklist_categories*) be produced.

Also there could be applied limitation by minimal and maximal produced code point of the characters.

If you know what exactly characters you don't want to be produced, pass them with *blacklist_characters* argument.

`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.

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

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

min_size, average_size and max_size have the usual interpretations.

`hypothesis.strategies.`**`randoms`**()

Generates instances of Random (actually a Hypothesis specific RandomWithSeed class which displays what it was initially seeded with)

`hypothesis.strategies.`**`random_module`**()

If your code depends on the global random module then you need to use this.

It will explicitly seed the random module at the start of your test so that tests are reproducible. The value it passes you is an opaque object whose only useful feature is that its repr displays the random seed. It is not itself a random number generator. If you want a random number generator you should use the randoms() strategy which will give you one.

`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).

`hypothesis.strategies.`**`fractions`**(*min_value=None*, *max_value=None*, *max_denominator=None*)

Returns a strategy which generates Fractions.

If min_value is not None then all generated values are no less than min_value.

If max_value is not None then all generated values are no greater than max_value.

If max_denominator is not None then the absolute value of the denominator of any generated values is no greater than max_denominator. Note that max_denominator must be at least 1.

`hypothesis.strategies.`**`decimals`**(*min_value=None*, *max_value=None*)

Generates instances of decimals.Decimal.

If min_value is not None then all generated values are no less than min_value.

If max_value is not None then all generated values are no greater than max_value.

`hypothesis.strategies.`**`recursive`**(*base*, *extend*, *max_leaves=100*)

base: A strategy to start from.

extend: A function which takes a strategy and returns a new strategy.

max_leaves: The maximum number of elements to be drawn from base on a given run.

This returns a strategy S such that S = extend(base | S). That is, values maybe drawn from base, or from any strategy reachable by mixing applications of | and extend.

An example may clarify: recursive(booleans(), lists) would return a strategy that may return arbitrarily nested and mixed lists of booleans. So e.g. False, [True], [False, []], [[[[True]]]], are all valid values to be drawn from that strategy.

`hypothesis.strategies.`**`permutations`**`(`*values*`)`
Return a strategy which returns permutations of the collection "values".

`hypothesis.strategies.`**`composite`**`(`*f*`)`
Defines a strategy that is built out of potentially arbitrarily many other strategies.

This is intended to be used as a decorator. See the full documentation for more details about how to use this function.

`hypothesis.strategies.`**`shared`**`(`*base*, *key=None*`)`
Returns a strategy that draws a single shared value per run, drawn from base. Any two shared instances with the same key will share the same value, otherwise the identity of this strategy will be used. That is:

```
>>> x = shared(s)
>>> y = shared(s)
```

In the above x and y may draw different (or potentially the same) values. In the following they will always draw the same:

```
>>> x = shared(s, key="hi")
>>> y = shared(s, key="hi")
```

`hypothesis.strategies.`**`choices`**`()`
Strategy that generates a function that behaves like random.choice.

Will note choices made for reproducibility.

`hypothesis.strategies.`**`runner`**`(`*default=not_set*`)`
A strategy for getting "the current test runner", whatever that may be. The exact meaning depends on the entry point, but it will usually be the associated 'self' value for it.

If there is no current test runner and a default is provided, return that default. If no default is provided, raises InvalidArgument.

5.1 Choices

Sometimes you need an input to be from a known set of items. hypothesis gives you 2 ways to do this, choice() and sampled_from().

Examples on how to use them both are below. First up choice:

```
from hypothesis import given, strategies as st

@given(user=st.text(min_size=1), service=st.text(min_size=1), choice=st.choices())
def test_tickets(user, service, choice):
    t=choice(('ST', 'LT', 'TG', 'CT'))
    # asserts go here.
```

This means t will randomly be one of the items in the list ('ST', 'LT', 'TG', 'CT'). just like if you were calling random.choice() on the list.

A different, and probably better way to do this, is to use sampled_from:

```
from hypothesis import given, strategies as st

@given(
    user=st.text(min_size=1), service=st.text(min_size=1),
    t=st.sampled_from(('ST', 'LT', 'TG', 'CT')))
def test_tickets(user, service, t):
    # asserts and test code go here.
```

Values from sampled_from will not be copied and thus you should be careful of using mutable data. Which makes it great for the above use case, but may not always work out.

5.2 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.

```
>>> from hypothesis.strategies import streaming, integers
>>> x = 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):

```
>>> 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:

```
>>> 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:

```
>>> t = streaming(int).example()
>>> tm = t.map(lambda n: n * 2)
>>> tm[0]
26
>>> t[0]
13
>>> tm
Stream(26, ...)
>>> t
Stream(13, ...)
```

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.

5.3 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.

5.3.1 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.:

```
>>> lists(integers()).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 hypothesis.strategies.

5.3.2 Filtering

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

```
>>> integers().filter(lambda x: x > 11).example()
1873
>>> integers().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:

```
>>> integers().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 ex
    '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:

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

5.3.3 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 eachother.

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

```
>>> 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).

5.3.4 Recursive data

Sometimes the data you want to generate has a recursive definition. e.g. if you wanted to generate JSON data, valid JSON is:

1. Any float, any boolean, any unicode string.

2. Any list of valid JSON data
3. Any dictionary mapping unicode strings to valid JSON data.

The problem is that you cannot call a strategy recursively and expect it to not just blow up and eat all your memory.

The way Hypothesis handles this is with the 'recursive' function in hypothesis.strategies which you pass in a base case and a function that given a strategy for your data type returns a new strategy for it. So for example:

```
>>> import hypothesis.strategies as st
>>> json = st.recursive(st.floats() | st.booleans() | st.text() | st.none(),
... lambda children: st.lists(children) | st.dictionaries(st.text(), children))
>>> json.example()
{'': None, '\U000b3407\U000b3407\U000b3407': {
    '': '"é""é\x11', '\x13': 1.6153068016570349e-282,
    '\x00': '\x11\x11\x11"\x11"é"éé\x11""éé"\x11"éé\x11éé\x11é\x11',
  '\x80': 'é\x11\x11\x11\x11\x11\x11', '\x13\x13\x00\x80\x80\x00': 4.643602465868519e-144
  }, '\U000b3407': None}
>>> json.example()
[]
>>> json.example()
'\x06ě\U000d25e4H\U000d25e4\x06ě'
```

That is, we start with our leaf data and then we augment it by allowing lists and dictionaries of anything we can generate as JSON data.

The size control of this works by limiting the maximum number of values that can be drawn from the base strategy. So for example if we wanted to only generate really small JSON we could do this as:

```
>>> small_lists = st.recursive(st.booleans(), st.lists, max_leaves=5)
>>> small_lists.example()
False
>>> small_lists.example()
[[False], [], [], [], [], []]
>>> small_lists.example()
False
>>> small_lists.example()
[]
```

5.4 Composite strategies

The @composite decorator lets you combine other strategies in more or less arbitrary ways. It's probably the main thing you'll want to use for complicated custom strategies.

The composite decorator works by giving you a function as the first argument that you can use to draw examples from other strategies. For example, the following gives you a list and an index into it:

```
@composite
def list_and_index(draw, elements=integers()):
    xs = draw(lists(elements, min_size=1))
    i = draw(integers(min_value=0, max_value=len(xs) - 1))
    return (xs, i)
```

'draw(s)' is a function that should be thought of as returning s.example(), except that the result is reproducible and will minimize correctly. The decorated function has the initial argument removed from the list, but will accept all the others in the expected order. Defaults are preserved.

```
>>> list_and_index()
list_and_index()
>>> list_and_index().example()
([5585, 4073], 1)

>>> list_and_index(booleans())
list_and_index(elements=booleans())
>>> list_and_index(booleans()).example()
([False, False, True], 1)
```

Note that the repr will work exactly like it does for all the built-in strategies: It will be a function that you can call to get the strategy in question, with values provided only if they do not match the defaults.

You can use assume inside composite functions:

```
@composite
def distinct_strings_with_common_characters(draw):
    x = draw(text(), min_size=1)
    y = draw(text(alphabet=x))
    assume(x != y)
    return (x, y)
```

This works as assume normally would, filtering out any examples for which the passed in argument is falsey.

5.5 Drawing interactively in tests

There is also the `data()` strategy, which gives you a means of using strategies interactively. Rather than having to specify everything up front in `@given` you can draw from strategies in the body of your test:

```
@given(data())
def test_draw_sequentially(data):
    x = data.draw(integers())
    y = data.draw(integers(min_value=x))
    assert x < y
```

If the test fails, each draw will be printed with the falsifying example. e.g. the above is wrong (it has a boundary condition error), so will print:

```
Falsifying example: test_draw_sequentially(data=data(...))
Draw 1: 0
Draw 2: 0
```

As you can see, data drawn this way is simplified as usual.

CHAPTER 6

Additional packages

Hypothesis itself does not have any dependencies, but there are some packages that need additional things installed in order to work.

You can install these dependencies using the setuptools extra feature as e.g. `pip install hypothesis[django]`. This will check installation of compatible versions.

You can also just install hypothesis into a project using them, ignore the version constraints, and hope for the best.

In general "Which version is Hypothesis compatible with?" is a hard question to answer and even harder to regularly test. Hypothesis is always tested against the latest compatible version and each package will note the expected compatibility range. If you run into a bug with any of these please specify the dependency version.

6.1 hypothesis[datetime]

As might be expected, this provides strategies for which generating instances of objects from the `datetime` module: `datetimes`, `dates`, and `times`. It depends on `pytz` to work.

It should work with just about any version of `pytz`. `pytz` has a very stable API and Hypothesis works around a bug or two in older versions.

It lives in the `hypothesis.extra.datetime` package.

`datetimes`(*allow_naive=None*, *timezones=None*, *min_year=None*, *max_year=None*)

This strategy generates `datetime` objects. For example:

```
>>> from hypothesis.extra.datetime import datetimes
>>> datetimes().example()
datetime.datetime(1705, 1, 20, 0, 32, 0, 973139, tzinfo=<DstTzInfo 'Israel...
>>> datetimes().example()
datetime.datetime(7274, 6, 9, 23, 0, 31, 75498, tzinfo=<DstTzInfo 'America...
```

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:

```
>>> datetimes(min_year=2001, max_year=2010).example()
datetime.datetime(2010, 7, 7, 0, 15, 0, 614034, tzinfo=<DstTzInfo 'Pacif...
>>> datetimes(min_year=2001, max_year=2010).example()
datetime.datetime(2006, 9, 26, 22, 0, 0, 220365, tzinfo=<DstTzInfo 'Asia...
```

You can also specify timezones:

```
>>> import pytz
>>> pytz.all_timezones[:3]
['Africa/Abidjan', 'Africa/Accra', 'Africa/Addis_Ababa']
>>> datetimes(timezones=pytz.all_timezones[:3]).example()
datetime.datetime(6257, 8, 21, 13, 6, 24, 8751, tzinfo=<DstTzInfo 'Africa/Accra' GMT0:00:00 STD>
>>> datetimes(timezones=pytz.all_timezones[:3]).example()
datetime.datetime(7851, 2, 3, 0, 0, 0, 767400, tzinfo=<DstTzInfo 'Africa/Accra' GMT0:00:00 STD>)
>>> datetimes(timezones=pytz.all_timezones[:3]).example()
datetime.datetime(8262, 6, 22, 16, 0, 0, 154235, tzinfo=<DstTzInfo 'Africa/Abidjan' GMT0:00:00 S
```

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

```
>>> datetimes(timezones=[]).example()
datetime.datetime(918, 11, 26, 2, 0, 35, 916439)
```

You can also explicitly get a mix of naive and non-naive datetimes if you want:

```
>>> datetimes(allow_naive=True).example()
datetime.datetime(2433, 3, 20, 0, 0, 44, 460383, tzinfo=<DstTzInfo 'Asia/Hovd' HOVT+7:00:00 STD>
>>> datetimes(allow_naive=True).example()
datetime.datetime(7003, 1, 22, 0, 0, 52, 401259)
```

dates (*min_year=None*, *max_year=None*)

This strategy generates `date` objects. For example:

```
>>> from hypothesis.extra.datetime import dates
>>> dates().example()
datetime.date(1687, 3, 23)
>>> dates().example()
datetime.date(9565, 5, 2)
```

Again, you can restrict the range with the `min_year` and `max_year` arguments.

times (*allow_naive=None*, *timezones=None*)

This strategy generates `time` objects. For example:

```
>>> from hypothesis.extra.datetime import times
>>> times().example()
datetime.time(0, 15, 55, 188712, tzinfo=<DstTzInfo 'US/Hawaii' LMT-1 day, 13:29:00 STD>)
>>> times().example()
datetime.time(9, 0, 47, 959374, tzinfo=<DstTzInfo 'Pacific/Bougainville' BST+11:00:00 STD>)
```

The `allow_naive` and `timezones` arguments act the same as the datetimes strategy.

6.2 hypothesis[fakefactory]

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

The fake-factory API is extremely unstable, even between patch releases, and Hypothesis's support for it is unlikely to work with anything except the exact version it has been tested against.

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:

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

>>> fake_factory('name').example()
'Zbyněk Černý CSc.'
```

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

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

```
>>> 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[django]

hypothesis.extra.django adds support for testing your Django models with Hypothesis.

It should be compatible with any Django since 1.7, but is only tested extensively against 1.8.

It's large enough that it is documented elsewhere.

6.4 hypothesis[numpy]

hypothesis.extra.numpy adds support for testing NumPy-based implementations with Hypothesis by providing an `arrays` function.

It lives in the `hypothesis.extra.numpy` package.

`arrays`(*dtype*, *shape*, *elements=None*)

Arrays of specified *dtype* and *shape* are generated for example like this:

```
>>> import numpy as np
>>> arrays(np.int8, (2, 3)).example()
array([[-8,  6,  3],
       [-6,  4,  6]], dtype=int8)
```

However, to obtain more fine grained control over the elements, use the *elements* keyword (see also What you can generate and how):

```
>>> import numpy as np
>>> from hypothesis.strategies import floats
>>> arrays(np.float, 3, elements=floats(min_value=0, max_value=1)).example()
array([ 0.88974794,  0.77387938,  0.1977879 ])
```

By combining different strategies, the shape of the array can be modified as well:

```
>>> import numpy as np
>>> from hypothesis.strategies import integers, floats
>>>
>>> def rnd_len_arrays(dtype, min_len=0, max_len=3, elements=None):
...     lengths = integers(min_value=min_len, max_value=max_len)
...     return lengths.flatmap(lambda n: arrays(dtype, n, elements=elements))
...
>>>
>>> rla = rnd_len_arrays(np.int8)
>>> rla.example()
array([], dtype=int8)
>>> rla.example()
array([-2], dtype=int8)
>>> rla.example()
array([ 7, -6, -2], dtype=int8)
```

CHAPTER 7

Health checks

Hypothesis tries to detect common mistakes and things that will cause difficulty at run time in the form of a number of 'health checks'.

These include detecting and warning about:

- Strategies with very slow data generation
- Strategies which filter out too much
- Recursive strategies which branch too much
- Use of the global random module

If any of these scenarios are detected, Hypothesis will emit a warning about them.

The general goal of these health checks is to warn you about things that you are doing that might appear to work but will either cause Hypothesis to not work correctly or to perform badly.

To selectively disable health checks, use the suppress_health_check settings. The argument for this parameter is a list with elements drawn from any of the class-level attributes of the HealthCheck class.

To disable all health checks, set the perform_health_check settings parameter to False.

CHAPTER 8

The Hypothesis Example Database

When Hypothesis finds a bug it stores enough information in its database to reproduce it. This enables you to have a classic testing workflow of find a bug, fix a bug, and be confident that this is actually doing the right thing because Hypothesis will start by retrying the examples that broke things last time.

8.1 Limitations

The database is best thought of as a cache that you never need to invalidate: Information may be lost when you upgrade a Hypothesis version or change your test, so you shouldn't rely on it for correctness - if there's an example you want to ensure occurs each time then *there's a feature for including them in your source code* - but it helps the development workflow considerably by making sure that the examples you've just found are reproduced.

8.2 File locations

The default storage format is as a fairly opaque directory structure. Each test corresponds to a directory, and each example to a file within that directory. The standard location for it is .hypothesis/examples in your current working directory. You can override this, either by setting either the database_file property on a settings object (you probably want to specify it on settings.default) or by setting the HYPOTHESIS_DATABASE_FILE environment variable.

There is also a legacy sqlite3 based format. This is mostly still supported for compatibility reasons, and support will be dropped in some future version of Hypothesis. If you use a database file name ending in .db, .sqlite or .sqlite3 that format will be used instead.

8.3 Upgrading Hypothesis and changing your tests

The design of the Hypothesis database is such that you can put arbitrary data in the database and not get wrong behaviour. When you upgrade Hypothesis, old data *might* be invalidated, but this should happen transparently. It should never be the case that e.g. changing the strategy that generates an argument sometimes gives you data from the old strategy.

8.4 Sharing your example database

The only currently supported workflow for this (though it would be easy enough to add new ones) is via checking the examples directory into version control.

The directory structure is designed so that it is entirely suitable for checking in to git, mercurial, or any similar version control system. It will be updated reasonably often, so you might not want to do that in the course of normal development, but it will correctly handle merges, deletes, etc without a problem if you just add the directory into version control.

Note: There are other files in .hypothesis but everything other than the examples will be transparently created on demand. You don't need to and probably shouldn't check those into git. Adding .hypothesis/eval_source to your .gitignore or equivalent is probably a good idea.

CHAPTER 9

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.

9.1 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:

```
import unittest
from collections import namedtuple

from hypothesis import strategies as st
from hypothesis.stateful import RuleBasedStateMachine, Bundle, rule


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


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

    @rule(target=trees, x=st.integers())
    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=trees)
    def check_balanced(self, tree):
        if isinstance(tree, Leaf):
            return
        else:
            assert abs(self.size(tree.left) - self.size(tree.right)) <= 1
            self.check_balanced(tree.left)
            self.check_balanced(tree.right)

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

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:

```
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 lets balance some trees.

```
from collections import namedtuple

from hypothesis import strategies as st
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=st.integers())
    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(
                self.split_leaves(leaves[:mid]),
                self.split_leaves(leaves[mid:]),
            )
```

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:

```
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:

```
TestTrees.settings = settings(max_examples=100, 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.

9.1.1 Preconditions

While it's possible to use `assume` in RuleBasedStateMachine rules, if you use it in only a few rules you can quickly run into a situation where few or none of your rules pass their assumptions. Thus, Hypothesis provides a `precondition` decorator to avoid this problem. The `precondition` decorator is used on `rule`-decorated functions, and must be given a function that returns True or False based on the RuleBasedStateMachine instance.

```
from hypothesis.stateful import RuleBasedStateMachine, rule, precondition

class NumberModifier(RuleBasedStateMachine):

    num = 0

    @rule()
    def add_one(self):
        self.num += 1

    @precondition(lambda self: self.num != 0)
    @rule()
    def divide_with_one(self):
        self.num = 1 / self.num
```

By using `precondition` here instead of `assume`, Hypothesis can filter the inapplicable rules before running them. This makes it much more likely that a useful sequence of steps will be generated.

Note that currently preconditions can't access bundles; if you need to use preconditions, you should store relevant data on the instance instead.

9.2 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:

```
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:

```
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):

```
import unittest

from hypothesis.stateful import GenericStateMachine
from hypothesis.strategies import tuples, sampled_from, just, integers


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

    def steps(self):
        add_strategy = tuples(just("add"), integers())
        if not self.data:
            return add_strategy
        else:
            return (
                add_strategy |
                tuples(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__':
    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)
----------------------------------------------------------------------
Traceback (most recent call last):
(...)
    assert value not in self.data
AssertionError
```

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.

9.3 More fine grained control

If you want to bypass the TestCase infrastructure you can invoke these manually. The stateful module exposes the function run_state_machine_as_test, which takes an arbitrary function returning a GenericStateMachine and an optional settings parameter and does the same as the class based runTest provided.

In particular this may be useful if you wish to pass parameters to a custom __init__ in your subclass.

CHAPTER 10

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.

10.1 Python versions

Hypothesis is supported and tested on python 2.7 and python 3.3+. Python 3.0 through 3.2 are unsupported and definitely don't work. It's not infeasible to make them work but would need a very good reason.

Python 2.6 and 3.3 are supported on a "best effort" basis. They probably work, and bugs that affect them *might* get fixed.

Hypothesis also supports PyPy (PyPy3 does not work because it only runs 3.2 compatible code, but if and when there's a 3.3 compatible version it will be supported), and should support 32-bit and narrow builds, though this is currently only tested on Windows.

Hypothesis does not currently work on Jython (it requires sqlite), though could feasibly be made to do so. IronPython might work but hasn't been tested.

In general Hypothesis does not officially support anything except the latest patch release of any version of Python it supports. Earlier releases should work and bugs in them will get fixed if reported, but they're not tested in CI and no guarantees are made.

10.2 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.

10.3 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.

If your testing relies on doing something other than calling a function and seeing if it raises an exception then it probably *won't* work out of the box. In particular things like tests which return generators and expect you to do something with them (e.g. nose's yield based tests) will not work. Use a decorator or similar to wrap the test to take this form.

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.
- py.test fixtures work correctly with Hypothesis based functions, but note that function based fixtures will only run once for the whole function, not once per example.
- 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. yield based tests simply won't work.
- Integration with Django's testing requires use of the *Hypothesis for Django users* 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.

10.4 Django Versions

The Hypothesis Django integration is supported on 1.7 and 1.8. It will probably not work on versions prior to that.

10.5 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.

CHAPTER 11

Some more examples

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).

11.1 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:

```
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:

```
from functools import total_ordering


@total_ordering
class TopoKey(object):
    def __init__(self, node):
        self.value = node
    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:

```
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:

```
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:

```
from hypothesis import given

@given(s.lists(NodeStrategy))
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. Lets change the test to do that.

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


NodeSet = s.lists(Node).map(deduplicate_nodes_by_label)
```

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:

```
@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:

```
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.

11.2 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.

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

```
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
```

11.3 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_votes = []
    for vote in votes:
        rv = []
        for v in vote:
            if v not in rv:
                rv.append(v)
        assert len(rv) == len(all_candidates)
        rebuilt_votes.append(rv)
    return rebuilt_votes


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

11.4 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.

```
import unittest
from hypothesis import given, assume, settings, strategies as st
from collections import namedtuple
import requests
import os
import random
import time
import math

# 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 = st.fixed_dictionaries({
    'title': st.text(),
    'goal_type': st.sampled_from([
        "hustler", "biker", "gainer", "fatloser", "inboxer",
        "drinker", "custom"]),
    'goaldate': st.one_of(st.none(), st.floats()),
    'goalval': st.one_of(st.none(), st.floats()),
    'rate': st.one_of(st.none(), st.floats()),
    'initval': st.floats(),
    'panic': st.floats(),
    'secret': st.booleans(),
    'datapublic': st.booleans(),
})


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()
```

CHAPTER 12

Community

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).

12.1 Code of conduct

Hypothesis's community is an inclusive space, and everyone in it is expected to abide by a code of conduct.

At the high level the code of conduct goes like this:

1. Be kind
2. Be respectful
3. Be helpful

While it is impossible to enumerate everything that is unkind, disrespectful or unhelpful, here are some specific things that are definitely against the code of conduct:

1. -isms and -phobias (e.g. racism, sexism, transphobia and homophobia) are unkind, disrespectful *and* unhelpful. Just don't.
2. All software is broken. This is not a moral failing on the part of the authors. Don't give people a hard time for bad code.
3. It's OK not to know things. Everybody was a beginner once, nobody should be made to feel bad for it.
4. It's OK not to *want* to know something. If you think someone's question is fundamentally flawed, you should still ask permission before explaining what they should actually be asking.
5. Note that "I was just joking" is not a valid defence.

12.1.1 What happens when this goes wrong?

For minor infractions, I'll just call people on it and ask them to apologise and not do it again. You should feel free to do this too if you're comfortable doing so.

Major infractions and repeat offenders will be banned from the community.

Also, people who have a track record of bad behaviour outside of the Hypothesis community may be banned even if they obey all these rules if their presence is making people uncomfortable.

At the current volume level it's not hard for me to pay attention to the whole community, but if you think I've missed something please feel free to alert me. You can either message me as DRMacIver on freenode or send a me an email at david@drmaciver.com.

CHAPTER 13

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 them 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, let's fix it by figuring out how to integrate Hypothesis into it. Too hard to generate custom data for your application? That's a bug. Let's 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.

CHAPTER 14

Who is using Hypothesis?

This is a page for listing people who are using Hypothesis and how excited they are about that. If that's you and your name is not on the list, this file is in Git and I'd love it if you sent me a pull request to fix that.

14.1 Kristian Glass - Director of Technology at LaterPay GmbH

Hypothesis has been brilliant for expanding the coverage of our test cases, and also for making them much easier to read and understand, so we're sure we're testing the things we want in the way we want.

14.2 Seth Morton

When I first heard about Hypothesis, I knew I had to include it in my two open-source Python libraries, natsort and fastnumbers . Quite frankly, I was a little appalled at the number of bugs and "holes" I found in the code. I can now say with confidence that my libraries are more robust to "the wild." In addition, Hypothesis gave me the confidence to expand these libraries to fully support Unicode input, which I never would have had the stomach for without such thorough testing capabilities. Thanks!

14.3 Sixty North

At Sixty North we use Hypothesis for testing Segpy an open source Python library for shifting data between Python data structures and SEG Y files which contain geophysical data from the seismic reflection surveys used in oil and gas exploration.

This is our first experience of property-based testing – as opposed to example-based testing. Not only are our tests more powerful, they are also much better explanations of what we expect of the production code. In fact, the tests are much closer to being specifications. Hypothesis has located real defects in our code which went undetected by traditional test cases, simply because Hypothesis is more relentlessly devious about test case generation than us mere humans! We found Hypothesis particularly beneficial for Segpy because SEG Y is an antiquated format that uses legacy text encodings (EBCDIC) and even a legacy floating point format we implemented from scratch in Python.

Hypothesis is sure to find a place in most of our future Python codebases and many existing ones too.

14.4 mulkieran

Just found out about this excellent QuickCheck for Python implementation and ran up a few tests for my bytesize package last night. Refuted a few hypotheses in the process.

Looking forward to using it with a bunch of other projects as well.

14.5 Adam Johnson

I have written a small library to serialize `dicts` to MariaDB's dynamic columns binary format, mariadb-dyncol. When I first developed it, I thought I had tested it really well - there were hundreds of test cases, some of them even taken from MariaDB's test suite itself. I was ready to release.

Lucky for me, I tried Hypothesis with David at the PyCon UK sprints. Wow! It found bug after bug after bug. Even after a first release, I thought of a way to make the tests do more validation, which revealed a further round of bugs! Most impressively, Hypothesis found a complicated off-by-one error in a condition with 4095 versus 4096 bytes of data - something that I would never have found.

Long live Hypothesis! (Or at least, property-based testing).

14.6 Josh Bronson

Adopting Hypothesis improved bidict's test coverage and significantly increased our ability to make changes to the code with confidence that correct behavior would be preserved. Thank you, David, for the great testing tool.

14.7 Cory Benfield

Hypothesis is the single most powerful tool in my toolbox for working with algorithmic code, or any software that produces predictable output from a wide range of sources. When using it with Priority, Hypothesis consistently found errors in my assumptions and extremely subtle bugs that would have taken months of real-world use to locate. In some cases, Hypothesis found subtle deviations from the correct output of the algorithm that may never have been noticed at all.

When it comes to validating the correctness of your tools, nothing comes close to the thoroughness and power of Hypothesis.

14.8 Jon Moore

One extremely satisfied user here. Hypothesis is a really solid implementation of property-based testing, adapted well to Python, and with good features such as failure-case shrinkers. I first used it on a project where we needed to verify that a vendor's Python and non-Python implementations of an algorithm matched, and it found about a dozen cases that previous example-based testing and code inspections had not. Since then I've been evangelizing for it at our firm.

14.9 Russel Winder

I am using Hypothesis as an integral part of my Python workshops. Testing is an integral part of Python programming and whilst unittest and, better, py.test can handle example-based testing, property-based testing is increasingly far more important than example-base testing, and Hypothesis fits the bill.

14.10 Your name goes here

I know there are many more, because I keep finding out about new people I'd never even heard of using Hypothesis. If you're looking to way to give back to a tool you love, adding your name here only takes a moment and would really help a lot. As per instructions at the top, just send me a pull request and I'll add you to the list.

CHAPTER 15

Open Source Projects using Hypothesis

The following is a non-exhaustive list of open source projects I know are using Hypothesis. If you're aware of any others please add them to the list! The only inclusion criterion right now is that if it's a Python library then it should be available on pypi.

- aur
- axelrod
- bidict
- binaryornot
- brotlipy
- chardet
- cmph-cffi
- cryptography
- dbus-signature-pyparsing
- fastnumbers
- flocker
- flownetpy
- funsize
- fusion-index
- hyper-h2
- justbases
- justbytes
- mariadb-dyncol
- mercurial
- natsort
- pretext
- priority
- PyPy
- pyrsistent

- pyudev
- qutebrowser
- RubyMarshal
- Segpy
- simoa
- srt
- tchannel
- vdirsyncer
- wcag-contrast-ratio
- yacluster
- yturl

CHAPTER 16

Changelog

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.

16.1 3.5.3 - 2016-10-05

This is a bug fix release.

Bugs fixed:

- If the same test was running concurrently in two processes and there were examples already in the test database which no longer failed, Hypothesis would sometimes fail with a FileNotFoundError (IOError on Python 2) because an example it was trying to read was deleted before it was read. (Issue #372).
- Drawing from an integers() strategy with both a min_value and a max_value would reject too many examples needlessly. Now it repeatedly redraws until satisfied. (Pull request #366. Thanks to Calen Pennington for the contribution).

16.2 3.5.2 - 2016-09-24

This is a bug fix release.

- The Hypothesis pytest plugin broke pytest support for doctests. Now it doesn't.

16.3 3.5.1 - 2016-09-23

This is a bug fix release.

- Hypothesis now runs cleanly in -B and -BB modes, avoiding mixing bytes and unicode.
- unittest.TestCase tests would now have shown up in the new statistics mode. Now they do.
- Similarly, stateful tests would not have shown up in statistics and now they do.
- Statistics now print with pytest node IDs (the names you'd get in pytest verbose mode).

16.4 3.5.0 - 2016-09-22

This is a feature release.

- fractions() and decimals() strategies now support min_value and max_value parameters. Thanks go to Anne Mulhern for the development of this feature.
- The Hypothesis pytest plugin now supports a –hypothesis-show-statistics parameter that gives detailed statistics about the tests that were run. Huge thanks to Jean-Louis Fuchs and Adfinis-SyGroup for funding the development of this feature.
- There is a new event() function that can be used to add custom statistics.

Additionally there have been some minor bug fixes:

- In some cases Hypothesis should produce fewer duplicate examples (this will mostly only affect cases with a single parameter).
- py.test command line parameters are now under an option group for Hypothesis (thanks to David Keijser for fixing this)
- Hypothesis would previously error if you used function annotations on your tests under Python 3.4.
- The repr of many strategies using lambdas has been improved to include the lambda body (this was previously supported in many but not all cases).

16.5 3.4.2 - 2016-07-13

This is a bug fix release, fixing a number of problems with the settings system:

- Test functions defined using @given can now be called from other threads (Issue #337)
- Attempting to delete a settings property would previously have silently done the wrong thing. Now it raises an AttributeError.
- Creating a settings object with a custom database_file parameter was silently getting ignored and the default was being used instead. Now it's not.

16.6 3.4.1 - 2016-07-07

This is a bug fix release for a single bug:

- On Windows when running two Hypothesis processes in parallel (e.g. using pytest-xdist) they could race with each other and one would raise an exception due to the non-atomic nature of file renaming on Windows and the fact that you can't rename over an existing file. This is now fixed.

16.7 3.4.0 - 2016-05-27

This release is entirely provided by Lucas Wiman:

models() strategies from hypothesis.extra.django will now respect much more of Django's validations out of the box. Wherever possible full_clean() should succeed.

In particular:

- The max_length, blank and choices kwargs are now respected.
- Add support for DecimalField.
- If a field includes validators, the list of validators are used to filter the field strategy.

16.8 3.3.0 - 2016-05-27

This release went wrong and is functionally equivalent to 3.2.0. Ignore it.

16.9 3.2.0 - 2016-05-19

This is a small single-feature release:

- All tests using @given now fix the global random seed. This removes the health check for that. If a non-zero seed is required for the final falsifying example, it will be reported. Otherwise Hypothesis will assume randomization was not a significant factor for the test and be silent on the subject. If you use the random_module() strategy this will continue to work and will always display the seed.

16.10 3.1.3 - 2016-05-01

Single bug fix release

- Another charmap problem. In 3.1.2 text/characters would break on systems which had /tmp/ mounted on a different partition than the Hypothesis storage directory (usually in home). This fixes that.

16.11 3.1.2 - 2016-04-30

Single bug fix release:

- Anything which used a text() or characters() strategy was broken on Windows and I hadn't updated appveyor to use the new repository location so I didn't notice. This is now fixed and windows support should work correctly.

16.12 3.1.1 - 2016-04-29

Minor bug fix release.

- Fix concurrency issue when running tests that use text() from multiple processes at once (Bug #302, thanks to Alex Chan).
- Improve performance of code using lists with max_size (thanks to Cristi Cobzarenco).
- Fix install on Python 2 with ancient versions of pip so that it installs the enum34 backport (thanks to Donald Stufft for telling me how to do this).
- Remove duplicated __all__ exports from hypothesis.strategies (thanks to Piët Delport).
- Update headers to point to new repository location.
- Allow use of strategies that can't be used in find() (e.g. choices) in stateful testing.

16.13 3.1.0 - 2016-03-06

- Add a 'nothing' strategy that never successfully generates values.
- sampled_from() and one_of() can both now be called with an empty argument list, in which case they also never generate any values.
- one_of may now be called with a single argument that is a collection of strategies as well as as varargs.
- Add a 'runner' strategy which returns the instance of the current test object if there is one.
- 'Bundle' for RuleBasedStateMachine is now a normal(ish) strategy and can be used as such.
- Tests using RuleBasedStateMachine should now shrink significantly better.
- Hypothesis now uses a pretty-printing library internally, compatible with IPython's pretty printing protocol (actually using the same code). This may improve the quality of output in some cases.
- As a 'phases' setting that allows more fine grained control over which parts of the process Hypothesis runs
- Add a suppress_health_check setting which allows you to turn off specific health checks in a fine grained manner.
- Fix a bug where lists of non fixed size would always draw one more element than they included. This mostly didn't matter, but if would cause problems with empty strategies or ones with side effects.
- Add a mechanism to the Django model generator to allow you to explicitly request the default value (thanks to Jeremy Thurgood for this one).

16.14 3.0.5 - 2016-02-25

- Fix a bug where Hypothesis would now error on py.test development versions.

16.15 3.0.4 - 2016-02-24

- Fix a bug where Hypothesis would error when running on Python 2.7.3 or earlier because it was trying to pass a bytearray object to struct.unpack (which is only supported since 2.7.4).

16.16 3.0.3 - 2016-02-23

- Fix version parsing of py.test to work with py.test release candidates
- More general handling of the health check problem where things could fail because of a cache miss - now one "free" example is generated before the start of the health check run.

16.17 3.0.2 - 2016-02-18

- Under certain circumstances, strategies involving text() buried inside some other strategy (e.g. text().filter(...) or recursive(text(), ...)) would cause a test to fail its health checks the first time it ran. This was caused by having to compute some related data and cache it to disk. On travis or anywhere else where the .hypothesis directory was recreated this would have caused the tests to fail their health check on every run. This is now fixed for all the known cases, although there could be others lurking.

16.18 3.0.1 - 2016-02-18

- Fix a case where it was possible to trigger an "Unreachable" assertion when running certain flaky stateful tests.
- Improve shrinking of large stateful tests by eliminating a case where it was hard to delete early steps.
- Improve efficiency of drawing binary(min_size=n, max_size=n) significantly by provide a custom implementation for fixed size blocks that can bypass a lot of machinery.
- Set default home directory based on the current working directory at the point Hypothesis is imported, not whenever the function first happens to be called.

16.19 3.0.0 - 2016-02-17

Codename: This really should have been 2.1.

Externally this looks like a very small release. It has one small breaking change that probably doesn't affect anyone at all (some behaviour that never really worked correctly is now outright forbidden) but necessitated a major version bump and one visible new feature.

Internally this is a complete rewrite. Almost nothing other than the public API is the same.

New features:

- Addition of data() strategy which allows you to draw arbitrary data interactively within the test.
- New "exploded" database format which allows you to more easily check the example database into a source repository while supporting merging.
- Better management of how examples are saved in the database.
- Health checks will now raise as errors when they fail. It was too easy to have the warnings be swallowed entirely.

New limitations:

- choices and streaming strategies may no longer be used with find(). Neither may data() (this is the change that necessitated a major version bump).

Feature removal:

- The ForkingTestCase executor has gone away. It may return in some more working form at a later date.

Performance improvements:

- A new model which allows flatmap, composite strategies and stateful testing to perform *much* better. They should also be more reliable.
- Filtering may in some circumstances have improved significantly. This will help especially in cases where you have lots of values with individual filters on them, such as lists(x.filter(...)).
- Modest performance improvements to the general test runner by avoiding expensive operations

In general your tests should have got faster. If they've instead got significantly slower, I'm interested in hearing about it.

Data distribution:

The data distribution should have changed significantly. This may uncover bugs the previous version missed. It may also miss bugs the previous version could have uncovered. Hypothesis is now producing less strongly correlated data than it used to, but the correlations are extended over more of the structure.

Shrinking:

Shrinking quality should have improved. In particular Hypothesis can now perform simultaneous shrinking of separate examples within a single test (previously it was only able to do this for elements of a single collection). In some cases performance will have improved, in some cases it will have got worse but generally shouldn't have by much.

16.20 2.0.0 - 2016-01-10

Codename: A new beginning

This release cleans up all of the legacy that accrued in the course of Hypothesis 1.0. These are mostly things that were emitting deprecation warnings in 1.19.0, but there were a few additional changes.

In particular:

- non-strategy values will no longer be converted to strategies when used in given or find.
- FailedHealthCheck is now an error and not a warning.
- Handling of non-ascii reprs in user types have been simplified by using raw strings in more places in Python 2.
- given no longer allows mixing positional and keyword arguments.
- given no longer works with functions with defaults.
- given no longer turns provided arguments into defaults - they will not appear in the argspec at all.
- the basic() strategy no longer exists.
- the n_ary_tree strategy no longer exists.
- the average_list_length setting no longer exists. Note: If you're using using recursive() this will cause you a significant slow down. You should pass explicit average_size parameters to collections in recursive calls.
- @rule can no longer be applied to the same method twice.
- Python 2.6 and 3.3 are no longer officially supported, although in practice they still work fine.

This also includes two non-deprecation changes:

- given's keyword arguments no longer have to be the rightmost arguments and can appear anywhere in the method signature.
- The max_shrinks setting would sometimes not have been respected.

16.21 1.19.0 - 2016-01-09

Codename: IT COMES

This release heralds the beginning of a new and terrible age of Hypothesis 2.0.

It's primary purpose is some final deprecations prior to said release. The goal is that if your code emits no warnings under this release then it will probably run unchanged under Hypothesis 2.0 (there are some caveats to this: 2.0 will drop support for some Python versions, and if you're using internal APIs then as usual that may break without warning).

It does have two new features:

- New @seed() decorator which allows you to manually seed a test. This may be harmlessly combined with and overrides the derandomize setting.
- settings objects may now be used as a decorator to fix those settings to a particular @given test.

API changes (old usage still works but is deprecated):

- Settings has been renamed to settings (lower casing) in order to make the decorator usage more natural.
- Functions for the storage directory that were in hypothesis.settings are now in a new hypothesis.configuration module.

Additional deprecations:

- the average_list_length setting has been deprecated in favour of being explicit.
- the basic() strategy has been deprecated as it is impossible to support it under a Conjecture based model, which will hopefully be implemented at some point in the 2.x series.
- the n_ary_tree strategy (which was never actually part of the public API) has been deprecated.
- Passing settings or random as keyword arguments to given is deprecated (use the new functionality instead)

Bug fixes:

- No longer emit PendingDeprecationWarning for __iter__ and StopIteration in streaming() values.
- When running in health check mode with non strict, don't print quite so many errors for an exception in reify.
- When an assumption made in a test or a filter is flaky, tests will now raise Flaky instead of UnsatisfiedAssumption.

16.22 1.18.1 - 2015-12-22

Two behind the scenes changes:

- Hypothesis will no longer write generated code to the file system. This will improve performance on some systems (e.g. if you're using PythonAnywhere which is running your code from NFS) and prevent some annoying interactions with auto-restarting systems.
- Hypothesis will cache the creation of some strategies. This can significantly improve performance for code that uses flatmap or composite and thus has to instantiate strategies a lot.

16.23 1.18.0 - 2015-12-21

Features:

- Tests and find are now explicitly seeded off the global random module. This means that if you nest one inside the other you will now get a health check error. It also means that you can control global randomization by seeding random.
- There is a new random_module() strategy which seeds the global random module for you and handles things so that you don't get a health check warning if you use it inside your tests.
- floats() now accepts two new arguments: allow_nan and allow_infinity. These default to the old behaviour, but when set to False will do what the names suggest.

Bug fixes:

- Fix a bug where tests that used text() on Python 3.4+ would not actually be deterministic even when explicitly seeded or using the derandomize mode, because generation depended on dictionary iteration order which was affected by hash randomization.
- Fix a bug where with complicated strategies the timing of the initial health check could affect the seeding of the subsequent test, which would also render supposedly deterministic tests non-deterministic in some scenarios.
- In some circumstances flatmap() could get confused by two structurally similar things it could generate and would produce a flaky test where the first time it produced an error but the second time it produced the other value, which was not an error. The same bug was presumably also possible in composite().
- flatmap() and composite() initial generation should now be moderately faster. This will be particularly noticeable when you have many values drawn from the same strategy in a single run, e.g. constructs like lists(s.flatmap(f)). Shrinking performance *may* have suffered, but this didn't actually produce an interestingly worse result in any of the standard scenarios tested.

16.24 1.17.1 - 2015-12-16

A small bug fix release, which fixes the fact that the 'note' function could not be used on tests which used the @example decorator to provide explicit examples.

16.25 1.17.0 - 2015-12-15

This is actually the same release as 1.16.1, but 1.16.1 has been pulled because it contains the following additional change that was not intended to be in a patch release (it's perfectly stable, but is a larger change that should have required a minor version bump):

- Hypothesis will now perform a series of "health checks" as part of running your tests. These detect and warn about some common error conditions that people often run into which wouldn't necessarily have caued the test to fail but would cause e.g. degraded performance or confusing results.

16.26 1.16.1 - 2015-12-14

Note: This release has been removed.

A small bugfix release that allows bdists for Hypothesis to be built under 2.7 - the compat3.py file which had Python 3 syntax wasn't intended to be loaded under Python 2, but when building a bdist it was. In particular this would break running setup.py test.

16.27 1.16.0 - 2015-12-08

There are no public API changes in this release but it includes a behaviour change that I wasn't comfortable putting in a patch release.

- Functions from hypothesis.strategies will no longer raise InvalidArgument on bad arguments. Instead the same errors will be raised when a test using such a strategy is run. This may improve startup time in some cases, but the main reason for it is so that errors in strategies won't cause errors in loading, and it can interact correctly with things like pytest.mark.skipif.
- Errors caused by accidentally invoking the legacy API are now much less confusing, although still throw NotImplementedError.
- hypothesis.extra.django is 1.9 compatible.
- When tests are run with max_shrinks=0 this will now still rerun the test on failure and will no longer print "Trying example:" before each run. Additionally note() will now work correctly when used with max_shrinks=0.

16.28 1.15.0 - 2015-11-24

A release with two new features.

- A 'characters' strategy for more flexible generation of text with particular character ranges and types, kindly contributed by Alexander Shorin.
- Add support for preconditions to the rule based stateful testing. Kindly contributed by Christopher Armstrong

16.29 1.14.0 - 2015-11-01

New features:

- Add 'note' function which lets you include additional information in the final test run's output.
- Add 'choices' strategy which gives you a choice function that emulates random.choice.
- Add 'uuid' strategy that generates UUIDs'
- Add 'shared' strategy that lets you create a strategy that just generates a single shared value for each test run

Bugs:

- Using strategies of the form streaming(x.flatmap(f)) with find or in stateful testing would have caused InvalidArgument errors when the resulting values were used (because code that expected to only be called within a test context would be invoked).

16.30 1.13.0 - 2015-10-29

This is quite a small release, but deprecates some public API functions and removes some internal API functionality so gets a minor version bump.

- All calls to the 'strategy' function are now deprecated, even ones which pass just a SearchStrategy instance (which is still a no-op).
- Never documented hypothesis.extra entry_points mechanism has now been removed (it was previously how hypothesis.extra packages were loaded and has been deprecated and unused for some time)

- Some corner cases that could previously have produced an OverflowError when simplifying failing cases using hypothesis.extra.datetimes (or dates or times) have now been fixed.
- Hypothesis load time for first import has been significantly reduced - it used to be around 250ms (on my SSD laptop) and now is around 100-150ms. This almost never matters but was slightly annoying when using it in the console.
- hypothesis.strategies.randoms was previously missing from __all__.

16.31 1.12.0 - 2015-10-18

- Significantly improved performance of creating strategies using the functions from the hypothesis.strategies module by deferring the calculation of their repr until it was needed. This is unlikely to have been an performance issue for you unless you were using flatmap, composite or stateful testing, but for some cases it could be quite a significant impact.
- A number of cases where the repr of strategies build from lambdas is improved
- Add dates() and times() strategies to hypothesis.extra.datetimes
- Add new 'profiles' mechanism to the settings system
- Deprecates mutability of Settings, both the Settings.default top level property and individual settings.
- A Settings object may now be directly initialized from a parent Settings.
- @given should now give a better error message if you attempt to use it with a function that uses destructuring arguments (it still won't work, but it will error more clearly),
- A number of spelling corrections in error messages
- py.test should no longer display the intermediate modules Hypothesis generates when running in verbose mode
- Hypothesis should now correctly handle printing objects with non-ascii reprs on python 3 when running in a locale that cannot handle ascii printing to stdout.
- Add a unique=True argument to lists(). This is equivalent to unique_by=lambda x: x, but offers a more convenient syntax.

16.32 1.11.4 - 2015-09-27

- Hide modifications Hypothesis needs to make to sys.path by undoing them after we've imported the relevant modules. This is a workaround for issues cryptography experienced on windows.
- Slightly improved performance of drawing from sampled_from on large lists of alternatives.
- Significantly improved performance of drawing from one_of or strategies using | (note this includes a lot of strategies internally - floats() and integers() both fall into this category). There turned out to be a massive performance regression introduced in 1.10.0 affecting these which probably would have made tests using Hypothesis significantly slower than they should have been.

16.33 1.11.3 - 2015-09-23

- Better argument validation for datetimes() strategy - previously setting max_year < datetime.MIN_YEAR or min_year > datetime.MAX_YEAR would not have raised an InvalidArgument error and instead would have behaved confusingly.

- Compatibility with being run on pytest < 2.7 (achieved by disabling the plugin).

16.34 1.11.2 - 2015-09-23

Bug fixes:

- Settings(database=my_db) would not be correctly inherited when used as a default setting, so that newly created settings would use the database_file setting and create an SQLite example database.
- Settings.default.database = my_db would previously have raised an error and now works.
- Timeout could sometimes be significantly exceeded if during simplification there were a lot of examples tried that didn't trigger the bug.
- When loading a heavily simplified example using a basic() strategy from the database this could cause Python to trigger a recursion error.
- Remove use of deprecated API in pytest plugin so as to not emit warning

Misc:

- hypothesis-pytest is now part of hypothesis core. This should have no externally visible consequences, but you should update your dependencies to remove hypothesis-pytest and depend on only Hypothesis.
- Better repr for hypothesis.extra.datetimes() strategies.
- Add .close() method to abstract base class for Backend (it was already present in the main implementation).

16.35 1.11.1 - 2015-09-16

Bug fixes:

- When running Hypothesis tests in parallel (e.g. using pytest-xdist) there was a race condition caused by code generation.
- Example databases are now cached per thread so as to not use sqlite connections from multiple threads. This should make Hypothesis now entirely thread safe.
- floats() with only min_value or max_value set would have had a very bad distribution.
- Running on 3.5, Hypothesis would have emitted deprecation warnings because of use of inspect.getargspec

16.36 1.11.0 - 2015-08-31

- text() with a non-string alphabet would have used the repr() of the the alphabet instead of its contexts. This is obviously silly. It now works with any sequence of things convertible to unicode strings.
- @given will now work on methods whose definitions contains no explicit positional arguments, only varargs (bug #118). This may have some knock on effects because it means that @given no longer changes the argspec of functions other than by adding defaults.
- Introduction of new @composite feature for more natural definition of strategies you'd previously have used flatmap for.

16.37 1.10.6 - 2015-08-26

Fix support for fixtures on Django 1.7.

16.38 1.10.4 - 2015-08-21

Tiny bug fix release:

- If the database_file setting is set to None, this would have resulted in an error when running tests. Now it does the same as setting database to None.

16.39 1.10.3 - 2015-08-19

Another small bug fix release.

- lists(elements, unique_by=some_function, min_size=n) would have raised a ValidationError if n > Settings.default.average_list_length because it would have wanted to use an average list length shorter than the minimum size of the list, which is impossible. Now it instead defaults to twice the minimum size in these circumstances.
- basic() strategy would have only ever produced at most ten distinct values per run of the test (which is bad if you e.g. have it inside a list). This was obviously silly. It will now produce a much better distribution of data, both duplicated and non duplicated.

16.40 1.10.2 - 2015-08-19

This is a small bug fix release:

- star imports from hypothesis should now work correctly.
- example quality for examples using flatmap will be better, as the way it had previously been implemented was causing problems where Hypothesis was erroneously labelling some examples as being duplicates.

16.41 1.10.0 - 2015-08-04

This is just a bugfix and performance release, but it changes some semi-public APIs, hence the minor version bump.

- Significant performance improvements for strategies which are one_of() many branches. In particular this included recursive() strategies. This should take the case where you use one recursive() strategy as the base strategy of another from unusably slow (tens of seconds per generated example) to reasonably fast.
- Better handling of just() and sampled_from() for values which have an incorrect __repr__ implementation that returns non-ASCII unicode on Python 2.
- Better performance for flatmap from changing the internal morpher API to be significantly less general purpose.
- Introduce a new semi-public BuildContext/cleanup API. This allows strategies to register cleanup activities that should run once the example is complete. Note that this will interact somewhat weirdly with find.
- Better simplification behaviour for streaming strategies.
- Don’t error on lambdas which use destructuring arguments in Python 2.

- Add some better reprs for a few strategies that were missing good ones.
- The Random instances provided by randoms() are now copyable.
- Slightly more debugging information about simplify when using a debug verbosity level.
- Support using given for functions with varargs, but not passing arguments to it as positional.

16.42 1.9.0 - 2015-07-27

Codename: The great bundling.

This release contains two fairly major changes.

The first is the deprecation of the hypothesis-extra mechanism. From now on all the packages that were previously bundled under it other than hypothesis-pytest (which is a different beast and will remain separate). The functionality remains unchanged and you can still import them from exactly the same location, they just are no longer separate packages.

The second is that this introduces a new way of building strategies which lets you build up strategies recursively from other strategies.

It also contains the minor change that calling .example() on a strategy object will give you examples that are more representative of the actual data you'll get. There used to be some logic in there to make the examples artificially simple but this proved to be a bad idea.

16.43 1.8.5 - 2015-07-24

This contains no functionality changes but fixes a mistake made with building the previous package that would have broken installation on Windows.

16.44 1.8.4 - 2015-07-20

Bugs fixed:

- When a call to floats() had endpoints which were not floats but merely convertible to one (e.g. integers), these would be included in the generated data which would cause it to generate non-floats.
- Splitting lambdas used in the definition of flatmap, map or filter over multiple lines would break the repr, which would in turn break their usage.

16.45 1.8.3 - 2015-07-20

"Falsifying example" would not have been printed when the failure came from an explicit example.

16.46 1.8.2 - 2015-07-18

Another small bugfix release:

- When using ForkingTestCase you would usually not get the falsifying example printed if the process exited abnormally (e.g. due to os._exit).

- Improvements to the distribution of characters when using text() with a default alphabet. In particular produces a better distribution of ascii and whitespace in the alphabet.

16.47 1.8.1 - 2015-07-17

This is a small release that contains a workaround for people who have bad reprs returning non ascii text on Python 2.7. This is not a bug fix for Hypothesis per se because that's not a thing that is actually supposed to work, but Hypothesis leans more heavily on repr than is typical so it's worth having a workaround for.

16.48 1.8.0 - 2015-07-16

New features:

- Much more sensible reprs for strategies, especially ones that come from hypothesis.strategies. These should now have as reprs python code that would produce the same strategy.
- lists() accepts a unique_by argument which forces the generated lists to be only contain elements unique according to some function key (which must return a hashable value).
- Better error messages from flaky tests to help you debug things.

Mostly invisible implementation details that may result in finding new bugs in your code:

- Sets and dictionary generation should now produce a better range of results.
- floats with bounds now focus more on 'critical values', trying to produce values at edge cases.
- flatmap should now have better simplification for complicated cases, as well as generally being (I hope) more reliable.

Bug fixes:

- You could not previously use assume() if you were using the forking executor.

16.49 1.7.2 - 2015-07-10

This is purely a bug fix release:

- When using floats() with stale data in the database you could sometimes get values in your tests that did not respect min_value or max_value.
- When getting a Flaky error from an unreliable test it would have incorrectly displayed the example that caused it.
- 2.6 dependency on backports was incorrectly specified. This would only have caused you problems if you were building a universal wheel from Hypothesis, which is not how Hypothesis ships, so unless you're explicitly building wheels for your dependencies and support Python 2.6 plus a later version of Python this probably would never have affected you.
- If you use flatmap in a way that the strategy on the right hand side depends sensitively on the left hand side you may have occasionally seen Flaky errors caused by producing unreliable examples when minimizing a bug. This use case may still be somewhat fraught to be honest. This code is due a major rearchitecture for 1.8, but in the meantime this release fixes the only source of this error that I'm aware of.

16.50 1.7.1 - 2015-06-29

Codename: There is no 1.7.0.

A slight technical hitch with a premature upload means there's was a yanked 1.7.0 release. Oops.

The major feature of this release is Python 2.6 support. Thanks to Jeff Meadows for doing most of the work there.

Other minor features

- strategies now has a permutations() function which returns a strategy yielding permutations of values from a given collection.
- if you have a flaky test it will print the exception that it last saw before failing with Flaky, even if you do not have verbose reporting on.
- Slightly experimental git merge script available as "python -m hypothesis.tools.mergedbs". Instructions on how to use it in the docstring of that file.

Bug fixes:

- Better performance from use of filter. In particular tests which involve large numbers of heavily filtered strategies should perform a lot better.
- floats() with a negative min_value would not have worked correctly (worryingly, it would have just silently failed to run any examples). This is now fixed.
- tests using sampled_from would error if the number of sampled elements was smaller than min_satisfying_examples.

16.51 1.6.2 - 2015-06-08

This is just a few small bug fixes:

- Size bounds were not validated for values for a binary() strategy when reading examples from the database.
- sampled_from is now in __all__ in hypothesis.strategies
- floats no longer consider negative integers to be simpler than positive non-integers
- Small floating point intervals now correctly count members, so if you have a floating point interval so narrow there are only a handful of values in it, this will no longer cause an error when Hypothesis runs out of values.

16.52 1.6.1 - 2015-05-21

This is a small patch release that fixes a bug where 1.6.0 broke the use of flatmap with the deprecated API and assumed the passed in function returned a SearchStrategy instance rather than converting it to a strategy.

16.53 1.6.0 - 2015-05-21

This is a smallish release designed to fix a number of bugs and smooth out some weird behaviours.

- Fix a critical bug in flatmap where it would reuse old strategies. If all your flatmap code was pure you're fine. If it's not, I'm surprised it's working at all. In particular if you want to use flatmap with django models, you desperately need to upgrade to this version.

- flatmap simplification performance should now be better in some cases where it previously had to redo work.
- Fix for a bug where invalid unicode data with surrogates could be generated during simplification (it was already filtered out during actual generation).
- The Hypothesis database is now keyed off the name of the test instead of the type of data. This makes much more sense now with the new strategies API and is generally more robust. This means you will lose old examples on upgrade.
- The database will now not delete values which fail to deserialize correctly, just skip them. This is to handle cases where multiple incompatible strategies share the same key.
- find now also saves and loads values from the database, keyed off a hash of the function you're finding from.
- Stateful tests now serialize and load values from the database. They should have before, really. This was a bug.
- Passing a different verbosity level into a test would not have worked entirely correctly, leaving off some messages. This is now fixed.
- Fix a bug where derandomized tests with unicode characters in the function body would error on Python 2.7.

16.54 1.5.0 - 2015-05-14

Codename: Strategic withdrawal.

The purpose of this release is a radical simplification of the API for building strategies. Instead of the old approach of @strategy.extend and things that get converted to strategies, you just build strategies directly.

The old method of defining strategies will still work until Hypothesis 2.0, because it's a major breaking change, but will now emit deprecation warnings.

The new API is also a lot more powerful as the functions for defining strategies give you a lot of dials to turn. See the updated data section for details.

Other changes:

- Mixing keyword and positional arguments in a call to @given is deprecated as well.
- There is a new setting called 'strict'. When set to True, Hypothesis will raise warnings instead of merely printing them. Turning it on by default is inadvisable because it means that Hypothesis minor releases can break your code, but it may be useful for making sure you catch all uses of deprecated APIs.
- max_examples in settings is now interpreted as meaning the maximum number of unique (ish) examples satisfying assumptions. A new setting max_iterations which defaults to a larger value has the old interpretation.
- Example generation should be significantly faster due to a new faster parameter selection algorithm. This will mostly show up for simple data types - for complex ones the parameter selection is almost certainly dominated.
- Simplification has some new heuristics that will tend to cut down on cases where it could previously take a very long time.
- timeout would previously not have been respected in cases where there were a lot of duplicate examples. You probably wouldn't have previously noticed this because max_examples counted duplicates, so this was very hard to hit in a way that mattered.
- A number of internal simplifications to the SearchStrategy API.
- You can now access the current Hypothesis version as hypothesis.__version__.
- A top level function is provided for running the stateful tests without the TestCase infrastructure.

16.55 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.

16.56 1.3.0 - 2015-05-22

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.

16.57 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.

16.58 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.

16.59 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

16.60 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.

16.61 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.

16.62 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.

16.63 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.

16.64 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.

16.65 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.

16.66 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 triggers when you have duplicates but you drew e.g. [-17, -17], this will now simplify to [0, 0].

16.67 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)

16.68 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.

16.69 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.

16.70 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 OS X 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 OS X. Fix this by only generating unicode codepoints in the range supported by the system.

16.71 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.

16.72 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...

16.73 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.

16.74 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.

16.75 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.

16.76 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.

16.77 0.2.2 - 2015-01-08

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

16.78 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.

16.79 0.2.0 - 2015-01-07

- It's aliiive.
- 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

16.80 0.1.4 - 2013-12-14

- Make verification runs time bounded with a configurable timeout

16.81 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

16.82 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.

16.83 0.1.1 - 2013-03-24

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

16.84 0.1.0 - 2013-03-23

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

16.85 0.0.5 - 2013-03-13

- No changes except trying to fix packaging

16.86 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

16.87 0.0.3 - 2013-03-13

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

16.88 0.0.2 - 2013-03-12

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

16.89 0.0.1 - 2013-03-10

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

CHAPTER 17

Ongoing Hypothesis Development

Hypothesis releases and development are managed by me, David R. MacIver. I am the primary author of Hypothesis.

However, I no longer do unpaid feature development on Hypothesis. My roles as leader of the project are:

1. Helping other people do feature development on Hypothesis
2. Fixing bugs and other code health issues
3. Improving documentation
4. General release management work
5. Planning the general roadmap of the project
6. Doing sponsored development on tasks that are too large or in depth for other people to take on

So all new features must either be sponsored or implemented by someone else. That being said, I take a fairly active role in shepherding pull requests and helping people write a new feature (see the contributing guidelines for details and this pull request for an example of how the process goes). This isn't "patches welcome", it's "I will help you write a patch".

All enhancement tickets on GitHub are tagged with either help-wanted if I think they're viable for someone else to pick up or for-a-modest-fee if I think they are not and that if you want them you should probably talk to me about paid development.

You are of course entirely welcome to ask for paid development on something that is marked help-wanted, or indeed to try to tackle something marked for-a-modest-fee yourself if you're feeling ambitious. These labels are very much intended as guidelines rather than rules.

CHAPTER 18

Help and Support

For questions you are happy to ask in public, the Hypothesis community is a friendly place where I or others will be more than happy to help you out. You're also welcome to ask questions on Stack Overflow. If you do, please tag them with 'python-hypothesis' so someone sees them.

For bugs and enhancements, please file an issue on the GitHub issue tracker. Note that as per the development policy, enhancements will probably not get implemented unless you're willing to pay for development or implement them yourself (with assistance from me). Bugs will tend to get fixed reasonably promptly, though it is of course on a best effort basis.

If you need to ask questions privately or want more of a guarantee of bugs being fixed promptly, please contact me on hypothesis-support@drmaciver.com to talk about availability of support contracts.

CHAPTER 19

Packaging Guidelines

Downstream packagers often want to package Hypothesis. Here are some guidelines.

The primary guideline is this: If you are not prepared to keep up with the Hypothesis release schedule, don't. You will annoy me and are doing your users a disservice.

Hypothesis has quite a frequent release schedule. It's very rare that it goes a month without a release, and there are often multiple releases in a given month.

Many people not only fail to follow the release schedule but also seem included to package versions which are months out of date even at the point of packaging. This will cause me to be very annoyed with you and you will consequently get very little co-operation from me.

If you *are* prepared to keep up with the Hypothesis release schedule, the rest of this document outlines some information you might find useful.

19.1 Release tarballs

These are available from the GitHub releases page. The tarballs on pypi are intended for installation from a Python tool such as pip or easy_install and should not be considered complete releases. Requests to include additional files in them will not be granted. Their absence is not a bug.

19.2 Dependencies

19.2.1 Python versions

Hypothesis is designed to work with a range of Python versions. Currently supported are:

- pypy-2.6.1 (earlier versions of pypy *may* work)
- CPython 2.7.x
- CPython 3.4.x
- CPython 3.5.x

If you feel the need to have separate Python 3 and Python 2 packages you can, but Hypothesis works unmodified on either.

19.2.2 Other Python libraries

Hypothesis has *optional* dependencies on the following libraries:

- pytz (almost any version should work)
- fake-factory (0.5.2 or 0.5.3)
- Django, 1.7 through 1.9 (This requires fake-factory to be installed)
- numpy, 1.10.x (earlier versions will probably work fine)
- py.test (2.7.0 or greater). This is a mandatory dependency for testing Hypothesis itself but optional for users.

The way this works when installing Hypothesis normally is that these features become available if the relevant library is installed.

19.3 Testing Hypothesis

If you want to test Hypothesis as part of your packaging you will probably not want to use the mechanisms Hypothesis itself uses for running its tests, because it has a lot of logic for installing and testing against different versions of Python.

The tests must be run with py.test. A version more recent than 2.7.0 is strongly encouraged, but it may work with earlier versions (however py.test specific logic is disabled before 2.7.0).

Tests are organised into a number of top level subdirectories of the tests/ directory.

- cover: This is a small, reasonably fast, collection of tests designed to give 100% coverage of all but a select subset of the files when run under Python 3.
- nocover: This is a much slower collection of tests that should not be run under coverage for performance reasons.
- py2: Tests that can only be run under Python 2
- py3: Tests that can only be run under Python 3
- datetime: This tests the subset of Hypothesis that depends on pytz
- fakefactory: This tests the subset of Hypothesis that depends on fakefactory.
- django: This tests the subset of Hypothesis that depends on django (this also depends on fakefactory).

An example invocation for running the coverage subset of these tests:

```
python setup.py install
pip install pytest # you will probably want to use your own packaging here
python -m pytest tests/cover
```

19.4 Examples

- arch linux
- gentoo (slightly behind at the time of this writing)

Other distros appear to really like Hypothesis 1.11. I do not encourage following their example.

Index