Hypothesis Documentation

Release 3.55.4

David R. MacIver

Apr 13, 2018

Contents

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.

Quick start guide

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

1.1 An example

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

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

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


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

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

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

Lets see how you'd do that with Hypothesis:

```python
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).

```python
if not input_string:
    return []
```

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

```python
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:

```python
@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:

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

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:

```
pip install -e .
```

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*.

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.

2.1 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:

```python
>>> 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
...
>>> try:
...     test_shuffle_is_noop()
... except AssertionError:
...     print('ls != ls2')
Falsifying example: test_shuffle_is_noop(ls=[0, 1], r=RandomWithSeed(53))
Shuffle: [1, 0]
ls != ls2
```

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

2.2 Test Statistics

If you are using pytest 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`:

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

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

You would see:

```
test_integers:

  - 100 passing examples, 0 failing examples, 0 invalid examples
  - Typical runtimes: ~ 1ms
  - Fraction of time spent in data generation: ~ 12%
  - Stopped because settings.max_examples=100
```

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:

```python
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:

  - 100 passing examples, 0 failing examples, 36 invalid examples
  - Typical runtimes: 0-1 ms
  - Fraction of time spent in data generation: ~ 16%
  - Stopped because settings.max_examples=100
  - Events:
    * 80.88%, Retried draw from integers().filter(lambda x: <unknown>) to satisfy
→filter
    * 26.47%, Aborted test because unable to satisfy integers().filter(lambda x:
→<unknown>)
```

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

hypothesis.**event**(*value*)
> Record an event that occurred this test. Statistics on number of test runs with each event will be reported at the end if you run Hypothesis in statistics reporting mode.
>
> Events should be strings or convertible to them.

```python
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:

  - 100 passing examples, 0 failing examples, 38 invalid examples
  - Typical runtimes: 0-1 ms
  - Fraction of time spent in data generation: ~ 16%
  - Stopped because settings.max_examples=100
  - Events:
    * 80.43%, Retried draw from integers().filter(lambda x: <unknown>) to satisfy
→filter
    * 31.88%, i mod 3 = 0
    * 27.54%, Aborted test because unable to satisfy integers().filter(lambda x:
→<unknown>)
    * 21.74%, i mod 3 = 1
    * 18.84%, i mod 3 = 2
```

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`.

2.3 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 100 examples per test (the default) and it turns out 70 of those examples don't match your needs, that's a lot of wasted time.

hypothesis.**assume**(*condition*)

> Calling `assume` is like an assert that marks the example as bad, rather than failing the test.
>
> This allows you to specify properties that you *assume* will be true, and let Hypothesis try to avoid similar examples in future.

For example suppose you had the following test:

```python
@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 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.

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:

```
@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. Only 0 examples considered satisfied assumptions
```

2.3.1 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: the 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 one-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(1, 1000)` is a lot better than `assume(1 <= x <= 1000)`, but `assume` will take you a long way if you can't.

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

```
>>> integers(min_value=0, max_value=10).example()
1
```

Many strategies are built 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()
(-24597, 12566)
```

Further details are *available in a separate document*.

2.5 The gory details of given parameters

hypothesis.**given**(*given_arguments*, **given_kwargs*)

A decorator for turning a test function that accepts arguments into a randomized test.

This is the main entry point to Hypothesis.

The *@given* decorator may be used to specify which 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, arbitrary keywords, or keyword-only arguments.

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 same arguments as the original test, minus those that are filled in by *@given*.

2.6 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. For example, `TransactionTestCase` in the Django extra runs each example in a separate database transaction.

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

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

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

```python
from unittest import TestCase

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

    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 `execute_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:

```python
from unittest import TestCase

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

and should be rewritten as:

```python
from unittest import TestCase
import inspect

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

2.7 Using Hypothesis to find values

You can use Hypothesis's data exploration features to find values satisfying some predicate. This is generally useful for exploring custom strategies defined with *@composite*, or experimenting with conditions for filtering data.

hypothesis.**find**(*specifier*, *condition*, *settings=None*, *random=None*, *database_key=None*)
> Returns the minimal example from the given strategy `specifier` that matches the predicate function `condition`.

```
>>> 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>
```

(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)

2.8 Inferred Strategies

In some cases, Hypothesis can work out what to do when you omit arguments. This is based on introspection, *not* magic, and therefore has well-defined limits.

builds() will check the signature of the `target` (using `getfullargspec()`). If there are required arguments with type annotations and no strategy was passed to *builds()*, *from_type()* is used to fill them in. You can also pass the special value `hypothesis.infer` as a keyword argument, to force this inference for arguments with a default value.

```
>>> def func(a: int, b: str):
...     return [a, b]
>>> builds(func).example()
[-6993, '']
```

@given does not perform any implicit inference for required arguments, as this would break compatibility with pytest fixtures. `infer` can be used as a keyword argument to explicitly fill in an argument from its type annotation.

```
@given(a=infer)
def test(a: int): pass
# is equivalent to
@given(a=integers())
def test(a): pass
```

2.8.1 Limitations

PEP 3107 type annotations are not supported on Python 2, and Hypothesis does not inspect **PEP 484** type comments at runtime. While *from_type()* will work as usual, inference in *builds()* and *@given* will only work if you manually create the __annotations__ attribute (e.g. by using @annotations(...) and @returns(...) decorators). The typing module is fully supported on Python 2 if you have the backport installed.

The typing module is provisional and has a number of internal changes between Python 3.5.0 and 3.6.1, including at minor versions. These are all supported on a best-effort basis, but you may encounter problems with an old version of the module. Please report them to us, and consider updating to a newer version of Python as a workaround.

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 is as follows:

```python
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:

```python
from hypothesis import given, settings

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

3.1 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.

buffer_size
> The size of the underlying data used to generate examples. If you need to generate really large examples you may want to increase this, but it will make your tests slower.
>
> default value: 8192

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, *:memory:* for an in-memory database, or any path for a directory-based example database.
>
> default value: (dynamically calculated)

database_file
> The file or directory location to save and load previously tried examples; *:memory:* for an in-memory cache or None to disable caching entirely.
>
> default value: (dynamically calculated)
>
> The *database_file* setting is deprecated in favor of the *database* setting, and will be removed in a future version. It only exists at all for complicated historical reasons and you should just use *database* instead.

deadline
> If set, a time in milliseconds (which may be a float to express smaller units of time) that each individual example (i.e. each time your test function is called, not the whole decorated test) within a test is not allowed to exceed. Tests which take longer than that may be converted into errors (but will not necessarily be if close to the deadline, to allow some variability in test run time).
>
> Set this to None to disable this behaviour entirely.
>
> In future this will default to 200. For now, a HypothesisDeprecationWarning will be emitted if you exceed that default deadline and have not explicitly set a deadline yourself.
>
> default value: not_set

derandomize
> If this is True then hypothesis will run in deterministic mode where each falsification uses a random number generator that is seeded based on the hypothesis to falsify, which will be consistent across multiple runs. This has the advantage that it will eliminate any randomness from your tests, which may be preferable for some situations. It does have the disadvantage of making your tests less likely to find novel breakages.
>
> default value: False

max_examples
> Once this many satisfying examples have been considered without finding any counter-example, falsification will terminate.
>
> default value: 100

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 `hypothesis.assume()` calls and which have not exhaustively covered the search space.

Note that examples are compared at the level of the underlying byte-stream - for example, `booleans()` contains 256 unique examples at this level because it always uses one byte.

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

phases

Control which phases should be run. See *the full documentation for more details*

default value: (<Phase.explicit: 0>, <Phase.reuse: 1>, <Phase.generate: 2>, <Phase.shrink: 3>)

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. Note also that, as strict mode is itself deprecated, enabling it is now an error!

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

default value: False

Strict mode is deprecated and will go away in a future version of Hypothesis. To get the same behaviour, use warnings.simplefilter('error', HypothesisDeprecationWarning).

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

The timeout setting is deprecated and will be removed in a future version of Hypothesis. To get the future behaviour set `timeout=hypothesis.unlimited` instead (which will remain valid for a further deprecation period after this setting has gone away).

use_coverage

Whether to use coverage information to improve Hypothesis's ability to find bugs.

You should generally leave this turned on unless your code performs poorly when run under coverage. If you turn it off, please file a bug report or add a comment to an existing one about the problem that prompted you to do so.

default value: True

3.1.1 Controlling What Runs

Hypothesis divides tests into four logically distinct phases:

1. Running explicit examples *provided with the @example decorator*.

2. Rerunning a selection of previously failing examples to reproduce a previously seen error

3. Generating new examples.

4. Attempting to shrink an example found in phases 2 or 3 to a more manageable one (explicit examples cannot be shrunk).

The phases setting provides you with fine grained control over which of these run, with each phase corresponding to a value on the `Phase` enum:

1. `Phase.explicit` controls whether explicit examples are run.

2. `Phase.reuse` controls whether previous examples will be reused.

3. `Phase.generate` controls whether new examples will be generated.

4. `Phase.shrink` controls whether examples will be shrunk.

The phases argument accepts a collection with any subset of these. e.g. `settings(phases=[Phase.generate, Phase.shrink])` will generate new examples and shrink them, but will not run explicit examples or reuse previous failures, while `settings(phases=[Phase.explicit])` will only run the explicit examples.

3.1.2 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*.

```
>>> from hypothesis import find, settings, Verbosity
>>> from hypothesis.strategies import lists, booleans
>>> find(lists(integers()), any, settings=settings(verbosity=Verbosity.verbose))
Tried non-satisfying example []
Found satisfying example [-1198601713, -67, 116, -29578]
Shrunk example to [-67, 116, -29578]
Shrunk example to [116, -29578]
Shrunk example to [-29578]
Shrunk example to [-115]
Shrunk example to [115]
Shrunk example to [-57]
Shrunk example to [29]
Shrunk example to [-14]
Shrunk example to [-7]
Shrunk example to [4]
Shrunk example to [2]
Shrunk example to [1]
[1]
```

The four levels are quiet, normal, verbose and debug. normal is the default, while in quiet mode Hypothesis will not print anything out, not 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.

If you are using pytest, you may also need to disable output capturing for passing tests.

3.2 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().max_examples
100
>>> settings(max_examples=10).max_examples
10
```

You can also copy settings from 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
```

3.3 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
100
```

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.

3.3.1 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", max_examples=1000)
>>> settings().max_examples
100
>>> 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.

```
>>> import os
>>> from hypothesis import settings, Verbosity
>>> settings.register_profile("ci", max_examples=1000)
>>> settings.register_profile("dev", max_examples=10)
>>> settings.register_profile("debug", 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>
```

3.3.2 Timeouts

The timeout functionality of Hypothesis is being deprecated, and will eventually be removed. For the moment, the timeout setting can still be set and the old default timeout of one minute remains.

If you want to future proof your code you can get the future behaviour by setting it to the value `hypothesis.unlimited`.

```
from hypothesis import given, settings, unlimited
from hypothesis import strategies as st
```

```python
@settings(timeout=unlimited)
@given(st.integers())
def test_something_slow(i):
    ...
```

This will cause your code to run until it hits the normal Hypothesis example limits, regardless of how long it takes. `timeout=unlimited` will remain a valid setting after the timeout functionality has been deprecated (but will then have its own deprecation cycle).

There is however now a timing related health check which is designed to catch tests that run for ages by accident. If you really want your test to run forever, the following code will enable that:

```python
from hypothesis import given, settings, unlimited, HealthCheck
from hypothesis import strategies as st

@settings(timeout=unlimited, suppress_health_check=[
    HealthCheck.hung_test
])
@given(st.integers())
def test_something_slow(i):
    ...
```

What you can generate and how

Most things should be easy to generate and everything should be possible.

To support this principle Hypothesis provides strategies for most built-in types with arguments to constrain or adjust the output, as well as higher-order strategies that can be composed to generate more complex types.

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.
>
> Examples from this strategy do not shrink (because there are none).

hypothesis.strategies.**just**(*value*)
> Return a strategy which only generates value.
>
> Note: value is not copied. Be wary of using mutable values.
>
> If value is the result of a callable, you can use *builds(callable)* instead of just(callable()) to get a fresh value each time.
>
> Examples from this strategy do not shrink (because there is only one).

hypothesis.strategies.**none**()
> Return a strategy which only generates None.
>
> Examples from this strategy do not shrink (because there is only one).

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.

Examples from this strategy will generally shrink to ones that come from strategies earlier in the list, then shrink according to behaviour of the strategy that produced them. In order to get good shrinking behaviour, try to put simpler strategies first. e.g. `one_of(none(), text())` is better than `one_of(text(), none())`.

This is especially important when using recursive strategies. e.g. `x = st.deferred(lambda: st.none() | st.tuples(x, x))` will shrink well, but `x = st.deferred(lambda: st.tuples(x, x) | st.none())` will shrink very badly indeed.

`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

    Examples from this strategy will shrink towards zero, and negative values will also shrink towards positive (i.e. -n may be replaced by +n).

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

    Examples from this strategy will shrink towards False (i.e. shrinking will try to replace True with False where possible).

`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.

    Examples from this strategy have a complicated and hard to explain shrinking behaviour, but it tries to improve "human readability". Finite numbers will be preferred to infinity and infinity will be preferred to NaN.

`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.

    Examples from this strategy shrink by shrinking their component parts.

`hypothesis.strategies.`**`sampled_from`**`(elements)`
    Returns a strategy which generates any value present in `elements`.

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

    `sampled_from` supports ordered collections, as well as `Enum` objects. `Flag` objects may also generate any combination of their members.

    Examples from this strategy shrink by replacing them with values earlier in the list. So e.g. sampled_from((10, 1)) will shrink by trying to replace 1 values with 10, and sampled_from((1, 10)) will shrink by trying to replace 10 values with 1.

`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 with 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.

The average_size argument is deprecated. Internal upgrades since Hypothesis 1.x mean we no longer needed this hint to generate useful data.

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]).

Examples from this strategy shrink by trying to remove elements from the list, and by shrinking each individual element of the list.

`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.

Examples from this strategy shrink by trying to remove elements from the set, and by shrinking each individual element of the set.

`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.`**`iterables`**(*elements=None*,   *min_size=None*,   *average_size=None*, *max_size=None*, *unique_by=None*, *unique=False*)
This has the same behaviour as lists, but returns iterables instead.

Some iterables cannot be indexed (e.g. sets) and some do not have a fixed length (e.g. generators). This strategy produces iterators, which cannot be indexed and do not have a fixed length. This ensures that you do not accidentally depend on sequence behaviour.

`hypothesis.strategies.`**`fixed_dictionaries`**(*mapping*)
Generates 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.

Examples from this strategy shrink by shrinking each individual value in the generated dictionary.

`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.

Examples from this strategy shrink by trying to remove keys from the generated dictionary, and by shrinking each generated key and value.

`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.

Examples from this strategy shrink by trying to shrink each value drawn.

Deprecated since version 3.15.0: Use [*data()*](#) instead.

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

> Generates unicode text type (unicode on python 2, str on python 3) characters following specified filtering rules.
>
> - When no filtering rules are specifed, any character can be produced.
>
> - If `min_codepoint` or `max_codepoint` is specifed, then only characters having a codepoint in that range will be produced.
>
> - If `whitelist_categories` is specified, then only characters from those Unicode categories will be produced. This is a further restriction, characters must also satisfy `min_codepoint` and `max_codepoint`.
>
> - If `blacklist_categories` is specified, then any character from those categories will not be produced. Any overlap between `whitelist_categories` and `blacklist_categories` will raise an exception, as each character can only belong to a single class.
>
> - If `whitelist_characters` is specified, then any additional characters in that list will also be produced.
>
> - If `blacklist_characters` is specified, then any characters in that list will be not be produced. Any overlap between `whitelist_characters` and `blacklist_characters` will raise an exception.
>
> The _codepoint arguments must be integers between zero and `sys.max_unicode`. The _characters arguments must be collections of length-one unicode strings, such as a unicode string.
>
> The _categories arguments must be used to specify either the one-letter Unicode major category or the two-letter Unicode general category. For example, (`'Nd'`, `'Lu'`) signifies "Number, decimal digit" and "Letter, uppercase". A single letter ('major category') can be given to match all corresponding categories, for example `'P'` for characters in any punctuation category.
>
> Examples from this strategy shrink towards the codepoint for `'0'`, or the first allowable codepoint after it if `'0'` is excluded.

`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 (excluding surrogate characters). If it is an empty collection this will only generate empty strings.
>
> min_size and max_size have the usual interpretations.
>
> The average_size argument is deprecated. Internal upgrades since Hypothesis 1.x mean we no longer needed this hint to generate useful data.
>
> Examples from this strategy shrink towards shorter strings, and with the characters in the text shrinking as per the alphabet strategy.

`hypothesis.strategies.`**`from_regex`**(*regex*)

> Generates strings that contain a match for the given regex (i.e. ones for which `re.search()` will return a non-None result).
>
> regex may be a pattern or `compiled regex`. Both byte-strings and unicode strings are supported, and will generate examples of the same type.
>
> You can use regex flags such as `re.IGNORECASE`, `re.DOTALL` or `re.UNICODE` to control generation. Flags can be passed either in compiled regex or inside the pattern with a `(?iLmsux)` group.
>
> Some regular expressions are only partly supported - the underlying strategy checks local matching and relies on filtering to resolve context-dependent expressions. Using too many of these constructs may cause health-

check errors as too many examples are filtered out. This mainly includes (positive or negative) lookahead and lookbehind groups.

If you want the generated string to match the whole regex you should use boundary markers. So e.g. `r"\A.`
`\Z"` will return a single character string, while `"."` will return any string, and `r"\A.$"` will return a single character optionally followed by a `"\n"`.

Examples from this strategy shrink towards shorter strings and lower character values.

`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 and max_size have the usual interpretations.

The average_size argument is deprecated. Internal upgrades since Hypothesis 1.x mean we no longer needed this hint to generate useful data.

Examples from this strategy shrink towards smaller strings and lower byte values.

`hypothesis.strategies.`**`randoms`**()
Generates instances of Random (actually a Hypothesis specific RandomWithSeed class which displays what it was initially seeded with)

Examples from this strategy shrink to seeds closer to zero.

`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.

Examples from these strategy shrink to seeds closer to zero.

`hypothesis.strategies.`**`builds`**(**callable_and_args*, ***kwargs*)
Generates values by drawing from `args` and `kwargs` and passing them to the callable (provided as the first positional argument) 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)`.

If the callable has type annotations, they will be used to infer a strategy for required arguments that were not passed to builds. You can also tell builds to infer a strategy for an optional argument by passing the special value `hypothesis.infer` as a keyword argument to builds, instead of a strategy for that argument to the callable.

Examples from this strategy shrink by shrinking the argument values to the callable.

`hypothesis.strategies.`**`from_type`**(*thing*)
Looks up the appropriate search strategy for the given type.

`from_type` is used internally to fill in missing arguments to *builds()* and can be used interactively to explore what strategies are available or to debug type resolution.

You can use *register_type_strategy()* to handle your custom types, or to globally redefine certain strategies - for example excluding NaN from floats, or use timezone-aware instead of naive time and datetime strategies.

The resolution logic may be changed in a future version, but currently tries these four options:

1. If `thing` is in the default lookup mapping or user-registered lookup, return the corresponding strategy. The default lookup covers all types with Hypothesis strategies, including extras where possible.

2. If `thing` is from the `typing` module, return the corresponding strategy (special logic).

3. If `thing` has one or more subtypes in the merged lookup, return the union of the strategies for those types that are not subtypes of other elements in the lookup.

4. Finally, if `thing` has type annotations for all required arguments, it is resolved via *builds()*.

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. min_value and max_value may be anything accepted by the `Fraction` constructor.

If max_denominator is not None then the denominator of any generated values is no greater than max_denominator. Note that max_denominator must be None or a positive integer.

Examples from this strategy shrink towards smaller denominators, then closer to zero.

hypothesis.strategies.**decimals**(*min_value=None,* *max_value=None,* *allow_nan=None,* *allow_infinity=None,* *places=None*)
Generates instances of `decimals.Decimal`, which may be:

- A finite rational number, between `min_value` and `max_value`.

- Not a Number, if `allow_nan` is True. None means "allow NaN, unless `min_value` and `max_value` are not None".

- Positive or negative infinity, if `max_value` and `min_value` respectively are None, and `allow_infinity` is not False. None means "allow infinity, unless excluded by the min and max values".

Note that where floats have one `NaN` value, Decimals have four: signed, and either *quiet* or *signalling*. See the decimal module docs for more information on special values.

If `places` is not None, all finite values drawn from the strategy will have that number of digits after the decimal place.

Examples from this strategy do not have a well defined shrink order but try to maximize human readability when shrinking.

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 may be 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, []]`, and `[[[[True]]]]` are all valid values to be drawn from that strategy.

Examples from this strategy shrink by trying to reduce the amount of recursion and by shrinking according to the shrinking behaviour of base and the result of extend.

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

Examples from this strategy shrink by trying to become closer to the original order of values.

`hypothesis.strategies.`**`datetimes`**(*min_value=datetime.datetime(1,   1,   1,   0,   0),*
                                    max_value=datetime.datetime(9999,   12,   31,   23,   59,
                                    59,   999999),   timezones=none(),   min_datetime=None,
                                    max_datetime=None)

A strategy for generating datetimes, which may be timezone-aware.

This strategy works by drawing a naive datetime between `min_datetime` and `max_datetime`, which must both be naive (have no timezone).

`timezones` must be a strategy that generates `tzinfo` objects (or None, which is valid for naive datetimes). A value drawn from this strategy will be added to a naive datetime, and the resulting tz-aware datetime returned.

Note: tz-aware datetimes from this strategy may be ambiguous or non-existent due to daylight savings, leap seconds, timezone and calendar adjustments, etc. This is intentional, as malformed timestamps are a common source of bugs.

---

hypothesis.extra.pytz.timezones() requires the pytz package, but provides all timezones in the Olsen database. If you want to allow naive datetimes, combine strategies like `none() | timezones()`.

Alternatively, you can create a list of the timezones you wish to allow (e.g. from the standard library, `datetutil`, or `pytz`) and use *sampled_from()*. Ensure that simple values such as None or UTC are at the beginning of the list for proper minimisation.

Examples from this strategy shrink towards midnight on January 1st 2000.

The following arguments have been renamed:

- min_datetime has been renamed to min_value

- max_datetime has been renamed to max_value

Use of the old names has been deprecated and will be removed in a future version of Hypothesis.

`hypothesis.strategies.`**`dates`**(*min_value=datetime.date(1, 1, 1), max_value=datetime.date(9999,*
                              12, 31), min_date=None, max_date=None)

A strategy for dates between `min_date` and `max_date`.

Examples from this strategy shrink towards January 1st 2000.

The following arguments have been renamed:

- max_date has been renamed to max_value

- min_date has been renamed to min_value

Use of the old names has been deprecated and will be removed in a future version of Hypothesis.

`hypothesis.strategies.`**`times`**(*min_value=datetime.time(0, 0), max_value=datetime.time(23, 59,*
                              59, 999999), timezones=none(), min_time=None, max_time=None)

A strategy for times between `min_time` and `max_time`.

The `timezones` argument is handled as for *datetimes()*.

Examples from this strategy shrink towards midnight, with the timezone component shrinking as for the strategy that provided it.

The following arguments have been renamed:

- min_time has been renamed to min_value

- max_time has been renamed to max_value

Use of the old names has been deprecated and will be removed in a future version of Hypothesis.

`hypothesis.strategies.`**`timedeltas`**(*min_value=datetime.timedelta(-999999999)*,
max_value=datetime.timedelta(999999999, 86399, 999999),
min_delta=None, *max_delta=None*)

A strategy for timedeltas between `min_value` and `max_value`.

Examples from this strategy shrink towards zero.

The following arguments have been renamed:

- max_delta has been renamed to max_value

- min_delta has been renamed to min_value

Use of the old names has been deprecated and will be removed in a future version of Hypothesis.

`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.

Examples from this strategy shrink by shrinking the output of each draw call.

`hypothesis.strategies.`**`complex_numbers`**(*min_magnitude=0*, *max_magnitude=None*, *allow_infinity=None*, *allow_nan=None*)

Returns a strategy that generates complex numbers.

This strategy draws complex numbers with constrained magnitudes. The `min_magnitude` and `max_magnitude` parameters should be non-negative `Real` numbers; values of `None` correspond to zero and infinite values respectively.

If `min_magnitude` is positive or `max_magnitude` is finite, it is an error to enable `allow_nan`. If `max_magnitude` is finite, it is an error to enable `allow_infinity`.

The magnitude contraints are respected up to a relative error of (around) floating-point epsilon, due to implementation via the system `sqrt` function.

Examples from this strategy shrink by shrinking their real and imaginary parts, as *floats()*.

If you need to generate complex numbers with particular real and imaginary parts or relationships between parts, consider using *builds(complex, ...) <hypothesis.strategies.builds>* or *@composite <hypothesis.strategies.composite>* respectively.

`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:

```
>>> s = integers()   # or any other strategy
>>> 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")
```

Examples from this strategy shrink as per their base strategy.

`hypothesis.strategies.`**`choices`**()

Strategy that generates a function that behaves like random.choice.

Will note choices made for reproducibility.

Deprecated since version 3.15.0: Use *data()* with *sampled_from()* instead.

Examples from this strategy shrink by making each choice function return an earlier value in the sequence passed to it.

hypothesis.strategies.**uuids**(*version=None*)

Returns a strategy that generates UUIDs.

If the optional version argument is given, value is passed through to UUID and only UUIDs of that version will be generated.

All returned values from this will be unique, so e.g. if you do lists(uuids()) the resulting list will never contain duplicates.

Examples from this strategy don't have any meaningful shrink order.

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.

Examples from this strategy do not shrink (because there is only one).

hypothesis.strategies.**data**()

This isn't really a normal strategy, but instead gives you an object which can be used to draw data interactively from other strategies.

It can only be used within *@given*, not *find()*. This is because the lifetime of the object cannot outlast the test body.

See *the rest of the documentation* for more complete information.

Examples from this strategy do not shrink (because there is only one), but the result of calls to each draw() call shrink as they normally would.

hypothesis.strategies.**register_type_strategy**(*custom_type*, *strategy*)

Add an entry to the global type-to-strategy lookup.

This lookup is used in *builds()* and *@given*.

builds() will be used automatically for classes with type annotations on __init__ , so you only need to register a strategy if one or more arguments need to be more tightly defined than their type-based default, or if you want to supply a strategy for an argument with a default value.

strategy may be a search strategy, or a function that takes a type and returns a strategy (useful for generic types).

hypothesis.strategies.**deferred**(*definition*)

A deferred strategy allows you to write a strategy that references other strategies that have not yet been defined. This allows for the easy definition of recursive and mutually recursive strategies.

The definition argument should be a zero-argument function that returns a strategy. It will be evaluated the first time the strategy is used to produce an example.

Example usage:

```
>>> import hypothesis.strategies as st
>>> x = st.deferred(lambda: st.booleans() | st.tuples(x, x))
>>> x.example()
(((False, (True, True)), (False, True)), (True, True))
>>> x.example()
True
```

Mutual recursion also works fine:

```
>>> a = st.deferred(lambda: st.booleans() | b)
>>> b = st.deferred(lambda: st.tuples(a, a))
>>> a.example()
True
>>> b.example()
(False, (False, ((False, True), False)))
```

Examples from this strategy shrink as they normally would from the strategy returned by the definition.

4.1 Shrinking

When using strategies it is worth thinking about how the data *shrinks*. Shrinking is the process by which Hypothesis tries to produce human readable examples when it finds a failure - it takes a complex example and turns it into a simpler one.

Each strategy defines an order in which it shrinks - you won't usually need to care about this much, but it can be worth being aware of as it can affect what the best way to write your own strategies is.

The exact shrinking behaviour is not a guaranteed part of the API, but it doesn't change that often and when it does it's usually because we think the new way produces nicer examples.

Possibly the most important one to be aware of is *one_of()*, which has a preference for values produced by strategies earlier in its argument list. Most of the others should largely "do the right thing" without you having to think about it.

4.2 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.

4.2.1 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()
[-25527, -24245, -23118, -93, -70, -7, 0, 39, 40, 65, 88, 112, 6189, 9480, 19469,␣
→27256, 32526, 1566924430]
```

Note that many things that you might use mapping for can also be done with *builds()*.

4.2.2 Filtering

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

```
>>> integers().filter(lambda x: x > 11).example()
26126
>>> integers().filter(lambda x: x > 11).example()
23324
```

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):
    ...
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(sorted).filter(lambda x: x[0] < x[1]).example()
[-8543729478746591815, 3760495307320535691]
```

4.2.3 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 each other.

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

```
>>> 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).

4.2.4 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 other problem here is that not all unicode strings display consistently on different machines, so we'll restrict them in our doctest.

The way Hypothesis handles this is with the *recursive()* function 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:

```
>>> from string import printable; from pprint import pprint
>>> json = recursive(none() | booleans() | floats() | text(printable),
... lambda children: lists(children) | dictionaries(text(printable), children))
>>> pprint(json.example())
['dy',
 [None, True, 6.297399055778002e+16, False],
 {'a{h\\:694K~{mY>a1yA:#CmDYb': None},
 '\\kP!4',
 {'#1J1': '',
  'cx.': None,
  "jv'A?qyp_sB\n$62g": [],
  'qgnP': [False, -inf, 'la)']},
 [],
 {}]
>>> pprint(json.example())
{'': None,
 '(Rt)': 1.192092896e-07,
 ',': [],
 '6': 2.2250738585072014e-308,
 'HA=/': [],
 'YU]gy8': inf,
 'l': None,
 'nK': False}
>>> pprint(json.example())
[]
```

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

4.3 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()
([-21904], 0)

>>> list_and_index(booleans())
list_and_index(elements=booleans())
>>> list_and_index(booleans()).example()
([True], 0)
```

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.

4.4 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.

Test functions using the *data()* strategy do not support explicit *@example(...)*s. In this case, the best option is usually to construct your data with *@composite* or the explicit example, and unpack this within the body of the test.

Optionally, you can provide a label to identify values generated by each call to `data.draw()`. These labels can be used to identify values in the output of a falsifying example.

For instance:

```python
@given(data())
def test_draw_sequentially(data):
    x = data.draw(integers(), label='First number')
    y = data.draw(integers(min_value=x), label='Second number')
    assert x < y
```

will produce the output:

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

CHAPTER 5

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.

There are separate pages for *Hypothesis for Django users* and *Hypothesis for the Scientific Stack*.

5.1 hypothesis[pytz]

This module provides `pytz` timezones.

You can use this strategy to make *hypothesis.strategies.datetimes()* and *hypothesis. strategies.times()* produce timezone-aware values.

hypothesis.extra.pytz.**timezones**()
　　Any timezone in the Olsen database, as a pytz tzinfo object.

　　This strategy minimises to UTC, or the smallest possible fixed offset, and is designed for use with *hypothesis.strategies.datetimes()*.

5.2 hypothesis[datetime]

This module provides deprecated time and date related strategies.

It depends on the `pytz` package, which is stable enough that almost any version should be compatible - most updates are for the timezone database.

`hypothesis.extra.datetime.`**`datetimes`**(*allow_naive=None*, *timezones=None*, *min_year=None*, *max_year=None*)

> Return a strategy for generating datetimes.
>
> Deprecated since version 3.9.0: use `hypothesis.strategies.datetimes()` instead.
>
> allow_naive=True will cause the values to sometimes be naive. timezones is the set of permissible timezones. If set to an empty collection all datetimes will be naive. If set to None all timezones available via pytz will be used.
>
> All generated datetimes will be between min_year and max_year, inclusive.

`hypothesis.extra.datetime.`**`dates`**(*min_year=None*, *max_year=None*)

> Return a strategy for generating dates.
>
> Deprecated since version 3.9.0: use `hypothesis.strategies.dates()` instead.
>
> All generated dates will be between min_year and max_year, inclusive.

`hypothesis.extra.datetime.`**`times`**(*allow_naive=None*, *timezones=None*)

> Return a strategy for generating times.
>
> Deprecated since version 3.9.0: use `hypothesis.strategies.times()` instead.
>
> The allow_naive and timezones arguments act the same as the datetimes strategy above.

5.3 hypothesis[fakefactory]

Note: This extra package is deprecated. We strongly recommend using native Hypothesis strategies, which are more effective at both finding and shrinking failing examples for your tests.

The `from_regex()`, `text()` (with some specific alphabet), and `sampled_from()` strategies may be particularly useful.

Faker (previously fake-factory) is a Python package that generates fake data for you. It's great for bootstraping your database, creating good-looking XML documents, stress-testing a database, or anonymizing production data. However, it's not designed for automated testing - data from Hypothesis looks less realistic, but produces minimal bug-triggering examples and uses coverage information to check more cases.

`hypothesis.extra.fakefactory` lets you use Faker generators to parametrize Hypothesis tests. This was only ever meant to ease your transition to Hypothesis, but we've improved Hypothesis enough since then that we no longer recommend using Faker for automated tests under any circumstances.

hypothesis.extra.fakefactory defines a function fake_factory which returns a strategy for producing text data from any Faker 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 Šťastný'
>>> fake_factory('name', locales=['en_GB', 'cs_CZ']).example()
'Harm Sanford'
```

You can use custom Faker providers via the `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'
```

Hypothesis for Django users

Hypothesis offers a number of features specific for Django testing, available in the `hypothesis[django]` *extra*. This is tested against each supported series with mainstream or extended support - if you're still getting security patches, you can test with Hypothesis.

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

I strongly recommend not using `TransactionTestCase` unless you really have to. Because Hypothesis runs this in a loop the performance problems it normally has are significantly exacerbated and your tests will be really slow. If you are using `TransactionTestCase`, you may need to use `@settings(suppress_health_check=[HealthCheck.too_slow])` to avoid *errors due to slow example generation*.

Having set up a test class, you can now pass *@given* a strategy for Django models:

`hypothesis.extra.django.models.`**`models`**(*model*, ***field_strategies*)
> Return a strategy for examples of `model`.

> > **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.

> `model` must be an subclass of `Model`. Strategies for fields may be passed as keyword arguments, for example `is_staff=st.just(False)`.

> Hypothesis can often infer a strategy based the field type and validators - for best results, make sure your validators are derived from Django's and therefore have the known types and attributes. Passing a keyword argument skips inference for that field; pass a strategy or pass `hypothesis.extra.django.models.default_value` to skip inference for that field.

> Foreign keys are not automatically derived. If they're nullable they will default to always being null, otherwise you always have to specify them. For example, examples of a Shop type with a foreign key to Company could

be generated with:

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

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\U000f19
↪'
>>> c.age
-873375803
```

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

Obviously the customer's age is implausible, which is only possible because we have not used (eg)
`MinValueValidator` to set the valid range for this field (or used a `PositiveSmallIntegerField`, which
would only need a maximum value validator).

If you *do* have validators attached, Hypothesis will only generate examples that pass validation. Sometimes that will
mean that we fail a `HealthCheck` because of the filtering, so let's explicitly pass a strategy to skip validation at the
strategy level:

Note: Inference from validators will be much more powerful when issue #1116 is implemented, but there will always
be some edge cases that require you to pass an explicit strategy.

```
>>> from hypothesis.strategies import integers
>>> c = models(Customer, age=integers(min_value=0, max_value=120)).example()
>>> c
<Customer: Customer object>
>>> c.age
5
```

6.1 Tips and tricks

6.1.1 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.

6.1.2 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.

6.1.3 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'
```

Hypothesis for the Scientific Stack

7.1 numpy

Hypothesis offers a number of strategies for NumPy testing, available in the `hypothesis[numpy]` *extra*. It lives in the `hypothesis.extra.numpy` package.

The centerpiece is the `arrays()` strategy, which generates arrays with any dtype, shape, and contents you can specify or give a strategy for. To make this as useful as possible, strategies are provided to generate array shapes and generate all kinds of fixed-size or compound dtypes.

hypothesis.extra.numpy.**arrays**(*dtype*, *shape*, *elements=None*, *fill=None*, *unique=False*)

Returns a strategy for generating `numpy's ndarrays`.

- `dtype` may be any valid input to `numpy.dtype` (this includes `dtype` objects), or a strategy that generates such values.

- `shape` may be an integer >= 0, a tuple of length >= 0 of such integers, or a strategy that generates such values.

- `elements` is a strategy for generating values to put in the array. If it is None a suitable value will be inferred based on the dtype, which may give any legal value (including eg `NaN` for floats). If you have more specific requirements, you should supply your own elements strategy.

- `fill` is a strategy that may be used to generate a single background value for the array. If None, a suitable default will be inferred based on the other arguments. If set to `st.nothing()` then filling behaviour will be disabled entirely and every element will be generated independently.

- `unique` specifies if the elements of the array should all be distinct from one another. Note that in this case multiple NaN values may still be allowed. If fill is also set, the only valid values for it to return are NaN values (anything for which `numpy.isnan()` returns True. So e.g. for complex numbers (nan+1j) is also a valid fill). Note that if unique is set to True the generated values must be hashable.

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)
```

- See *What you can generate and how*.

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

Array values are generated in two parts:

1. Some subset of the coordinates of the array are populated with a value drawn from the elements strategy (or its inferred form).

2. If any coordinates were not assigned in the previous step, a single value is drawn from the fill strategy and is assigned to all remaining places.

You can set fill to *nothing()* if you want to disable this behaviour and draw a value for every element.

If fill is set to None then it will attempt to infer the correct behaviour automatically: If unique is True, no filling will occur by default. Otherwise, if it looks safe to reuse the values of elements across multiple coordinates (this will be the case for any inferred strategy, and for most of the builtins, but is not the case for mutable values or strategies built with flatmap, map, composite, etc) then it will use the elements strategy as the fill, else it will default to having no fill.

Having a fill helps Hypothesis craft high quality examples, but its main importance is when the array generated is large: Hypothesis is primarily designed around testing small examples. If you have arrays with hundreds or more elements, having a fill value is essential if you want your tests to run in reasonable time.

hypothesis.extra.numpy.**array_shapes**(*min_dims=1*, *max_dims=3*, *min_side=1*, *max_side=10*)
Return a strategy for array shapes (tuples of int >= 1).

hypothesis.extra.numpy.**scalar_dtypes**()
Return a strategy that can return any non-flexible scalar dtype.

hypothesis.extra.numpy.**unsigned_integer_dtypes**(*endianness='?'*, *sizes=(8, 16, 32, 64)*)
Return a strategy for unsigned integer dtypes.

endianness may be < for little-endian, > for big-endian, = for native byte order, or ? to allow either byte order. This argument only applies to dtypes of more than one byte.

sizes must be a collection of integer sizes in bits. The default (8, 16, 32, 64) covers the full range of sizes.

hypothesis.extra.numpy.**integer_dtypes**(*endianness='?'*, *sizes=(8, 16, 32, 64)*)
Return a strategy for signed integer dtypes.

endianness and sizes are treated as for *unsigned_integer_dtypes()*.

hypothesis.extra.numpy.**floating_dtypes**(*endianness='?'*, *sizes=(16, 32, 64)*)
Return a strategy for floating-point dtypes.

sizes is the size in bits of floating-point number. Some machines support 96- or 128-bit floats, but these are not generated by default.

Larger floats (96 and 128 bit real parts) are not supported on all platforms and therefore disabled by default. To generate these dtypes, include these values in the sizes argument.

hypothesis.extra.numpy.**complex_number_dtypes**(*endianness='?'*, *sizes=(64, 128)*)
Return a strategy for complex-number dtypes.

sizes is the total size in bits of a complex number, which consists of two floats. Complex halfs (a 16-bit real part) are not supported by numpy and will not be generated by this strategy.

hypothesis.extra.numpy.**datetime64_dtypes**(*max_period='Y'*, *min_period='ns'*, *endianness='?'*)
> Return a strategy for datetime64 dtypes, with various precisions from year to attosecond.

hypothesis.extra.numpy.**timedelta64_dtypes**(*max_period='Y'*, *min_period='ns'*, *endianness='?'*)
> Return a strategy for timedelta64 dtypes, with various precisions from year to attosecond.

hypothesis.extra.numpy.**byte_string_dtypes**(*endianness='?'*, *min_len=0*, *max_len=16*)
> Return a strategy for generating bytestring dtypes, of various lengths and byteorder.

hypothesis.extra.numpy.**unicode_string_dtypes**(*endianness='?'*, *min_len=0*, *max_len=16*)
> Return a strategy for generating unicode string dtypes, of various lengths and byteorder.

hypothesis.extra.numpy.**array_dtypes**(*subtype_strategy=scalar_dtypes()*, *min_size=1*, *max_size=5*, *allow_subarrays=False*)
> Return a strategy for generating array (compound) dtypes, with members drawn from the given subtype strategy.

hypothesis.extra.numpy.**nested_dtypes**(*subtype_strategy=scalar_dtypes()*, *max_leaves=10*, *max_itemsize=None*)
> Return the most-general dtype strategy.
>
> Elements drawn from this strategy may be simple (from the subtype_strategy), or several such values drawn from *array_dtypes()* with allow_subarrays=True. Subdtypes in an array dtype may be nested to any depth, subject to the max_leaves argument.

7.2 pandas

Hypothesis provides strategies for several of the core pandas data types: pandas.Index, pandas.Series and pandas.DataFrame.

The general approach taken by the pandas module is that there are multiple strategies for generating indexes, and all of the other strategies take the number of entries they contain from their index strategy (with sensible defaults). So e.g. a Series is specified by specifying its numpy.dtype (and/or a strategy for generating elements for it).

hypothesis.extra.pandas.**indexes**(*elements=None*, *dtype=None*, *min_size=0*, *max_size=None*, *unique=True*)
> Provides a strategy for producing a pandas.Index.
>
> Arguments:
>
> - elements is a strategy which will be used to generate the individual values of the index. If None, it will be inferred from the dtype. Note: even if the elements strategy produces tuples, the generated value will not be a MultiIndex, but instead be a normal index whose elements are tuples.
>
> - dtype is the dtype of the resulting index. If None, it will be inferred from the elements strategy. At least one of dtype or elements must be provided.
>
> - min_size is the minimum number of elements in the index.
>
> - max_size is the maximum number of elements in the index. If None then it will default to a suitable small size. If you want larger indexes you should pass a max_size explicitly.
>
> - unique specifies whether all of the elements in the resulting index should be distinct.

hypothesis.extra.pandas.**range_indexes**(*min_size=0*, *max_size=None*)
> Provides a strategy which generates an Index whose values are 0, 1, ..., n for some n.
>
> Arguments:

- min_size is the smallest number of elements the index can have.

- max_size is the largest number of elements the index can have. If None it will default to some suitable value based on min_size.

`hypothesis.extra.pandas.`**`series`**(*elements=None*, *dtype=None*, *index=None*, *fill=None*, *unique=False*)

Provides a strategy for producing a `pandas.Series`.

Arguments:

- elements: a strategy that will be used to generate the individual values in the series. If None, we will attempt to infer a suitable default from the dtype.

- dtype: the dtype of the resulting series and may be any value that can be passed to `numpy.dtype`. If None, will use pandas's standard behaviour to infer it from the type of the elements values. Note that if the type of values that comes out of your elements strategy varies, then so will the resulting dtype of the series.

- index: If not None, a strategy for generating indexes for the resulting Series. This can generate either `pandas.Index` objects or any sequence of values (which will be passed to the Index constructor).

  You will probably find it most convenient to use the `indexes()` or `range_indexes()` function to produce values for this argument.

Usage:

```
>>> series(dtype=int).example()
0   -2001747478
1    1153062837
```

class `hypothesis.extra.pandas.`**`column`**(*name=None*, *elements=None*, *dtype=None*, *fill=None*, *unique=False*)

Data object for describing a column in a DataFrame.

Arguments:

- name: the column name, or None to default to the column position. Must be hashable, but can otherwise be any value supported as a pandas column name.

- elements: the strategy for generating values in this column, or None to infer it from the dtype.

- dtype: the dtype of the column, or None to infer it from the element strategy. At least one of dtype or elements must be provided.

- fill: A default value for elements of the column. See `arrays()` for a full explanation.

- unique: If all values in this column should be distinct.

`hypothesis.extra.pandas.`**`columns`**(*names_or_number*, *dtype=None*, *elements=None*, *fill=None*, *unique=False*)

A convenience function for producing a list of `column` objects of the same general shape.

The names_or_number argument is either a sequence of values, the elements of which will be used as the name for individual column objects, or a number, in which case that many unnamed columns will be created. All other arguments are passed through verbatim to create the columns.

`hypothesis.extra.pandas.`**`data_frames`**(*columns=None*, *rows=None*, *index=None*)

Provides a strategy for producing a `pandas.DataFrame`.

Arguments:

- columns: An iterable of `column` objects describing the shape of the generated DataFrame.

- rows: A strategy for generating a row object. Should generate either dicts mapping column names to values or a sequence mapping column position to the value in that position (note that unlike the `pandas.DataFrame` constructor, single values are not allowed here. Passing e.g. an integer is an error, even if there is only one column).

  At least one of rows and columns must be provided. If both are provided then the generated rows will be validated against the columns and an error will be raised if they don't match.

  Caveats on using rows:

  - In general you should prefer using columns to rows, and only use rows if the columns interface is insufficiently flexible to describe what you need - you will get better performance and example quality that way.

  - If you provide rows and not columns, then the shape and dtype of the resulting DataFrame may vary. e.g. if you have a mix of int and float in the values for one column in your row entries, the column will sometimes have an integral dtype and sometimes a float.

- index: If not None, a strategy for generating indexes for the resulting DataFrame. This can generate either `pandas.Index` objects or any sequence of values (which will be passed to the Index constructor).

  You will probably find it most convenient to use the *indexes()* or *range_indexes()* function to produce values for this argument.

Usage:

The expected usage pattern is that you use *column* and *columns()* to specify a fixed shape of the DataFrame you want as follows. For example the following gives a two column data frame:

```
>>> from hypothesis.extra.pandas import column, data_frames
>>> data_frames([
... column('A', dtype=int), column('B', dtype=float)]).example()
            A             B
0  2021915903  1.793898e+232
1  1146643993           inf
2 -2096165693  1.000000e+07
```

If you want the values in different columns to interact in some way you can use the rows argument. For example the following gives a two column DataFrame where the value in the first column is always at most the value in the second:

```
>>> from hypothesis.extra.pandas import column, data_frames
>>> import hypothesis.strategies as st
>>> data_frames(
...     rows=st.tuples(st.floats(allow_nan=False),
...                    st.floats(allow_nan=False)).map(sorted)
... ).example()
              0             1
0  -3.402823e+38  9.007199e+15
1 -1.562796e-298  5.000000e-01
```

You can also combine the two:

```
>>> from hypothesis.extra.pandas import columns, data_frames
>>> import hypothesis.strategies as st
>>> data_frames(
...     columns=columns(["lo", "hi"], dtype=float),
...     rows=st.tuples(st.floats(allow_nan=False),
...                    st.floats(allow_nan=False)).map(sorted)
... ).example()
```

```
         lo            hi
0   9.314723e-49   4.353037e+45
1  -9.999900e-01   1.000000e+07
2 -2.152861e+134 -1.069317e-73
```

(Note that the column dtype must still be specified and will not be inferred from the rows. This restriction may be lifted in future).

Combining rows and columns has the following behaviour:

- The column names and dtypes will be used.

- If the column is required to be unique, this will be enforced.

- Any values missing from the generated rows will be provided using the column's fill.

- Any values in the row not present in the column specification (if dicts are passed, if there are keys with no corresponding column name, if sequences are passed if there are too many items) will result in InvalidArgument being raised.

7.2.1 Supported Versions

There is quite a lot of variation between pandas versions. We only commit to supporting the latest version of pandas, but older minor versions are supported on a "best effort" basis. Hypothesis is currently tested against and confirmed working with Pandas 0.19, 0.20, 0.21, and 0.22.

Releases that are not the latest patch release of their minor version are not tested or officially supported, but will probably also work unless you hit a pandas bug.

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

- Tests that are unlikely to complete in a reasonable amount of time.

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* setting. 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* to False.

class hypothesis.**HealthCheck**

> Arguments for *suppress_health_check*.
>
> Each member of this enum is a type of health check to suppress.
>
> **exception_in_generation = 0**
>
> > Deprecated and no longer does anything. It used to convert errors in data generation into Failed-HealthCheck error.
>
> **data_too_large = 1**
>
> > Check for when the typical size of the examples you are generating exceeds the maximum allowed size too often.
>
> **filter_too_much = 2**
>
> > Check for when the test is filtering out too many examples, either through use of *assume()* or *filter()*, or occasionally for Hypothesis internal reasons.

too_slow = 3

Check for when your data generation is extremely slow and likely to hurt testing.

random_module = 4

Deprecated and no longer does anything. It used to check for whether your tests used the global random module. Now @given tests automatically seed random so this is no longer an error.

return_value = 5

Checks if your tests return a non-None value (which will be ignored and is unlikely to do what you want).

hung_test = 6

Checks if your tests have been running for a very long time.

large_base_example = 7

Checks if the natural example to shrink towards is very large.

not_a_test_method = 8

Checks if @given has been applied to a method of unittest.TestCase.

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.

9.1 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.

The database also records examples that excercise less-used parts of your code, so the database may update even when no failing examples were found.

9.2 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 by setting the `database` setting.

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.

If you have not configured a database and the default location is unusable (e.g. because you do not have read/write permission), Hypothesis will issue a warning and then fall back to an in-memory database.

9.3 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.

9.4 Sharing your example database

Note: If specific examples are important for correctness you should use the `@example` decorator, as the example database may discard entries due to changes in your code or dependencies. For most users, we therefore recommend using the example database locally and possibly persisting it between CI builds, but not tracking it under version control.

The examples database can be shared simply by checking the directory into version control, for example with the following `.gitignore`:

```
# Ignore files cached by Hypothesis...
.hypothesis/*
# except for the examples directory
!.hypothesis/examples/
```

Like everything under `.hypothesis/`, the examples directory will be transparently created on demand. Unlike the other subdirectories, `examples/` is designed to handle merges, deletes, etc if you just add the directory into git, mercurial, or any similar version control system.

Stateful testing

With `@given`, your tests are still something that you mostly write yourself, with Hypothesis providing some data. With Hypothesis's *stateful testing*, Hypothesis instead tries to generate not just data but entire tests. You specify a number of primitive actions that can be combined together, and then Hypothesis will try to find sequences of those actions that result in a failure.

Note: This style of testing is often called *model-based testing*, but in Hypothesis is called *stateful testing* (mostly for historical reasons - the original implementation of this idea in Hypothesis was more closely based on ScalaCheck's stateful testing where the name is more apt). Both of these names are somewhat misleading: You don't really need any sort of formal model of your code to use this, and it can be just as useful for pure APIs that don't involve any state as it is for stateful ones.

It's perhaps best to not take the name of this sort of testing too seriously. Regardless of what you call it, it is a powerful form of testing which is useful for most non-trivial APIs.

Hypothesis has two stateful testing APIs: A high level one, providing what we call *rule based state machines*, and a low level one, providing what we call *generic state machines*.

You probably want to use the rule based state machines - they provide a high level API for describing the sort of actions you want to perform, based on a structured representation of actions. However the generic state machines are more flexible, and are particularly useful if you want the set of currently possible actions to depend primarily on external state.

10.1 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.

The two main ingredients of a rule based state machine are rules and bundles.

A rule is very similar to a normal `@given` based test in that it takes values drawn from strategies and passes them to a user defined test function. The key difference is that where `@given` based tests must be independent, rules can be chained together - a single test run may involve multiple rule invocations, which may interact in various ways.

A Bundle is a named collection of generated values that can be reused by other operations in the test. They are populated with the results of rules, and may be used as arguments to rules, allowing data to flow from one rule to another, and rules to work on the results of previous computations or actions.

The following rule based state machine example is a simplified version of a test for Hypothesis's example database implementation. An example database maps keys to sets of values, and in this test we compare one implementation of it to a simplified in memory model of its behaviour, which just stores the same values in a Python `dict`. The test then runs operations against both the real database and the in-memory representation of it and looks for discrepancies in their behaviour.

```python
import shutil
import tempfile

from collections import defaultdict
import hypothesis.strategies as st
from hypothesis.database import DirectoryBasedExampleDatabase
from hypothesis.stateful import Bundle, RuleBasedStateMachine, rule


class DatabaseComparison(RuleBasedStateMachine):
    def __init__(self):
        super(DatabaseComparison, self).__init__()
        self.tempd = tempfile.mkdtemp()
        self.database = DirectoryBasedExampleDatabase(self.tempd)
        self.model = defaultdict(set)

    keys = Bundle('keys')
    values = Bundle('values')

    @rule(target=keys, k=st.binary())
    def k(self, k):
        return k

    @rule(target=values, v=st.binary())
    def v(self, v):
        return v

    @rule(k=keys, v=values)
    def save(self, k, v):
        self.model[k].add(v)
        self.database.save(k, v)

    @rule(k=keys, v=values)
    def delete(self, k, v):
        self.model[k].discard(v)
        self.database.delete(k, v)

    @rule(k=keys)
    def values_agree(self, k):
        assert set(self.database.fetch(k)) == self.model[k]

    def teardown(self):
        shutil.rmtree(self.tempd)


TestDBComparison = DatabaseComparison.TestCase
```

In this we declare two bundles - one for keys, and one for values. We have two trivial rules which just populate them

with data (`k` and `v`), and three non-trivial rules: `save` saves a value under a key and `delete` removes a value from a key, in both cases also updating the model of what *should* be in the database. `values_agree` then checks that the contents of the database agrees with the model for a particular key.

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

```
TestTrees = DatabaseComparison.TestCase

# Or just run with pytest's unittest support
if __name__ == '__main__':
    unittest.main()
```

This test currently passes, but if we comment out the line where we call `self.model[k].discard(v)`, we would see the following output when run under pytest:

```
AssertionError: assert set() == {b''}

------------ Hypothesis ------------

state = DatabaseComparison()
v1 = state.k(k=b'')
v2 = state.v(v=v1)
state.save(k=v1, v=v2)
state.delete(k=v1, v=v2)
state.values_agree(k=v1)
state.teardown()
```

Note how it's printed out a very short program that will demonstrate the problem. The output from a rule based state machine should generally be pretty close to Python code - if you have custom `repr` implementations that don't return valid Python then it might not be, but most of the time you should just be able to copy and paste the code into a test to reproduce it.

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:

```
DatabaseComparison.settings = settings(max_examples=50, stateful_step_count=100)
```

Which doubles the number of steps each program runs and halves the number of test cases that will be run.

10.2 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.

hypothesis.stateful.**precondition**(*precond*)
> Decorator to apply a precondition for rules in a RuleBasedStateMachine. Specifies a precondition for a rule to be considered as a valid step in the state machine. The given function will be called with the instance of RuleBasedStateMachine and should return True or False. Usually it will need to look at attributes on that instance.
>
> For example:

```
class MyTestMachine(RuleBasedStateMachine):
    state = 1

    @precondition(lambda self: self.state != 0)
    @rule(numerator=integers())
    def divide_with(self, numerator):
        self.state = numerator / self.state
```

This is better than using assume in your rule since more valid rules should be able to be run.

```
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.

10.3 Invariants

Often there are invariants that you want to ensure are met after every step in a process. It would be possible to add these as rules that are run, but they would be run zero or multiple times between other rules. Hypothesis provides a decorator that marks a function to be run after every step.

hypothesis.stateful.**invariant**()

Decorator to apply an invariant for rules in a RuleBasedStateMachine. The decorated function will be run after every rule and can raise an exception to indicate failed invariants.

For example:

```
class MyTestMachine(RuleBasedStateMachine):
    state = 1

    @invariant()
    def is_nonzero(self):
        assert self.state != 0
```

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

class NumberModifier(RuleBasedStateMachine):

    num = 0
```

```
    @rule()
    def add_two(self):
        self.num += 2
        if self.num > 50:
            self.num += 1

    @invariant()
    def divide_with_one(self):
        assert self.num % 2 == 0

NumberTest = NumberModifier.TestCase
```

Invariants can also have *precondition()*s applied to them, in which case they will only be run if the precondition function returns true.

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

10.4 Generic state machines

The class *GenericStateMachine* is the underlying machinery of stateful testing in Hypothesis. Chances are you will want to use the rule based stateful testing for most things, but the generic state machine functionality can be useful e.g. if you want to test things where the set of actions to be taken is more closely tied to the state of the system you are testing.

class hypothesis.stateful.**GenericStateMachine**
    A GenericStateMachine is the basic entry point into Hypothesis's approach to stateful testing.

    The intent is for it to be subclassed to provide state machine descriptions

    The way this is used is that Hypothesis will repeatedly execute something that looks something like:

```
x = MyStatemachineSubclass()
x.check_invariants()
try:
    for _ in range(n_steps):
        x.execute_step(x.steps().example())
        x.check_invariants()
finally:
    x.teardown()
```

    And if this ever produces an error it will shrink it down to a small sequence of example choices demonstrating that.

    steps()
        Return a SearchStrategy instance the defines the available next steps.

    execute_step(*step*)
        Execute a step that has been previously drawn from self.steps()

    teardown()
        Called after a run has finished executing to clean up any necessary state.

        Does nothing by default

    check_invariants()
        Called after initializing and after executing each step.

For example, here we use stateful testing as a sort of link checker, to test hypothesis.works for broken links or links that use HTTP instead of HTTPS.

```python
from hypothesis.stateful import GenericStateMachine
import hypothesis.strategies as st
from requests_html import HTMLSession


class LinkChecker(GenericStateMachine):
    def __init__(self):
        super(LinkChecker, self).__init__()
        self.session = HTMLSession()
        self.result = None

    def steps(self):
        if self.result is None:
            # Always start on the home page
            return st.just("https://hypothesis.works/")
        else:
            return st.sampled_from([
                l
                for l in self.result.html.absolute_links
                # Don't try to crawl to other people's sites
                if l.startswith("https://hypothesis.works") and
                # Avoid Cloudflare's bot protection. We are a bot but we don't
                # care about the info it's hiding.
                '/cdn-cgi/' not in l
            ])

    def execute_step(self, step):
        self.result = self.session.get(step)

        assert self.result.status_code == 200

        for l in self.result.html.absolute_links:
            # All links should be HTTPS
            assert "http://hypothesis.works" not in l


TestLinks = LinkChecker.TestCase
```

Running this (at the time of writing this documentation) produced the following output:

```
AssertionError: assert 'http://hypothesis.works' not in 'http://hypoth...test-
→fixtures/'
'http://hypothesis.works' is contained here:
  http://hypothesis.works/articles/hypothesis-pytest-fixtures/
? +++++++++++++++++++++++++

  ------------ Hypothesis ------------

Step #1: 'https://hypothesis.works/'
Step #2: 'https://hypothesis.works/articles/'
```

10.5 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.

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.

11.1 Python versions

Hypothesis is supported and tested on CPython 2.7 and CPython 3.4+.

Hypothesis also supports PyPy2, and will support PyPy3 when there is a stable release supporting Python 3.4+. Hypothesis does not currently work on Jython, though could feasibly be made to do so. IronPython might work but hasn't been tested. 32-bit and narrow builds should work, though this is currently only tested on Windows.

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.

11.2 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.

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.

11.3 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 we can make it, and this is verified as part of the CI. Note however that `@given` should only be used on tests, not `unittest.TestCase` setup or teardown methods.

- py.test fixtures work in the usual way for tests that have been decorated with `@given` - just avoid passing a strategy for each argument that will be supplied by a fixture. However, each fixture will run once for the whole function, not once per example. Decorating a fixture function is meaningless.

- Nose works fine with hypothesis, and this is 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 - we use it to guide example selection for user code, and Hypothesis has 100% branch coverage in its own tests.

11.4 Optional Packages

The supported versions of optional packages, for strategies in `hypothesis.extra`, are listed in the documentation for that extra. Our general goal is to support all versions that are supported upstream.

11.5 Regularly verifying this

Everything mentioned above as explicitly supported is checked on every commit with Travis, Appveyor, and CircleCI. Our continous delivery pipeline runs all of these checks before publishing each release, so when we say they're supported we really mean it.

11.6 Hypothesis versions

Backwards compatibility is better than backporting fixes, so we use *semantic versioning* and only support the most recent version of Hypothesis. See *Help and Support* for more information.

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).

12.1 How not to sort by a partial order

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

Suppose we've got the following type:

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

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

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

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

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

```python
from functools import total_ordering


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

    def __lt__(self, other):
        if self.value.sorts_before(other.value):
            return True
        if other.value.sorts_before(self.value):
            return False

        return self.value.label < other.value.label


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

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

But now we want to test that it works.

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

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

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

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

First we need to define a strategy for Node:

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

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

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

We can now write a test:

```python
from hypothesis import given

@given(s.lists(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(NodeStrategy).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.

12.2 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 *datetimes()* strategy works.

```
>>> from datetime import timedelta
>>> from hypothesis.extra.pytz import timezones
>>> from hypothesis.strategies import datetimes

>>> # The datetimes strategy is naive by default, so tell it to use timezones
>>> aware_datetimes = datetimes(timezones=timezones())

>>> @given(aware_datetimes, timezones(), 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(aware_datetimes, timezones())
... def test_convert_to_and_fro(dt, tz2):
...     """If we convert to a new timezone and back to the old one
...     this should leave the result unchanged.
...     """
...     tz1 = dt.tzinfo
...     assert dt == dt.astimezone(tz2).astimezone(tz1)

>>> @given(aware_datetimes, 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(aware_datetimes, 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

>>> # And we can check that our tests pass
>>> test_convert_via_intermediary()
>>> test_convert_to_and_fro()
>>> test_adding_an_hour_commutes()
>>> test_adding_a_day_commutes()
```

12.3 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.

12.4 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
```

```python
# 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 each other. 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()
```

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, which is more active than the mailing list.

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

Please note that the Hypothesis code of conduct applies in all Hypothesis community spaces.

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.

Testimonials

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.

15.1 Stripe

At Stripe we use Hypothesis to test every piece of our machine learning model training pipeline (powered by scikit). Before we migrated, our tests were filled with hand-crafted pandas Dataframes that weren't representative at all of our actual very complex data. Because we needed to craft examples for each test, we took the easy way out and lived with extremely low test coverage.

Hypothesis changed all that. Once we had our strategies for generating Dataframes of features it became trivial to slightly customize each strategy for new tests. Our coverage is now close to 90%.

Full-stop, property-based testing is profoundly more powerful - and has caught or prevented far more bugs - than our old style of example-based testing.

15.2 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.

15.3 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!

15.4 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.

15.5 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.

15.6 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).

15.7 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.

15.8 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.

15.9 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.

15.10 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.

15.11 Wellfire Interactive

We've been using Hypothesis in a variety of client projects, from testing Django-related functionality to domain-specific calculations. It both speeds up and simplifies the testing process since there's so much less tedious and error-prone work to do in identifying edge cases. Test coverage is nice but test depth is even nicer, and it's much easier to get meaningful test depth using Hypothesis.

15.12 Cody Kochmann

Hypothesis is being used as the engine for random object generation with my open source function fuzzer battle_tested which maps all behaviors of a function allowing you to minimize the chance of unexpected crashes when running code in production.

With how efficient Hypothesis is at generating the edge cases that cause unexpected behavior occur, battle_tested is able to map out the entire behavior of most functions in less than a few seconds.

Hypothesis truly is a masterpiece. I can't thank you enough for building it.

15.13 Merchise Autrement

Just minutes after our first use of hypothesis we uncovered a subtle bug in one of our most used library. Since then, we have increasingly used hypothesis to improve the quality of our testing in libraries and applications as well.

15.14 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.

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.

You can find hundreds more from the Hypothesis page at libraries.io.

- aur
- argon2_cffi
- attrs
- axelrod
- bidict
- binaryornot
- brotlipy
- chardet
- cmph-cffi
- cryptography
- dbus-signature-pyparsing
- fastnumbers
- flocker
- flownetpy
- funsize
- fusion-index
- hyper-h2
- into-dbus-python

- justbases
- justbytes
- loris
- mariadb-dyncol
- mercurial
- natsort
- pretext
- priority
- PyCEbox
- PyPy
- pyrsistent
- python-humble-utils
- pyudev
- qutebrowser
- RubyMarshal
- Segpy
- simoa
- srt
- tchannel
- vdirsyncer
- wcag-contrast-ratio
- yacluster
- yturl

Projects extending Hypothesis

Hypothesis has been eagerly used and extended by the open source community. This page lists extensions and applications; you can find more or newer packages by searching PyPI or libraries.io.

If there's something missing which you think should be here, let us know!

Note: Being listed on this page does not imply that the Hypothesis maintainers endorse a package.

17.1 External Strategies

Some packages provide strategies directly:

- hypothesis-fspaths - strategy to generate filesystem paths.
- hypothesis-geojson - strategy to generate GeoJson.
- hs-dbus-signature - strategy to generate arbitrary D-Bus signatures.
- hypothesis_sqlalchemy - strategies to generate SQLAlchemy objects.

Others provide a function to infer a strategy from some other schema:

- lollipop-hypothesis - infer strategies from lollipop schemas.
- hypothesis-drf - infer strategies from a djangorestframework serialiser.
- hypothesis-mongoengine - infer strategies from a mongoengine model.
- hypothesis-pb - infer strategies from Protocol Buffer schemas.

17.2 Other Cool Things

swagger-conformance is powered by Hypothesis and pyswagger. Based on a Swagger specification, it can build and

run an entire test suite to check that the implementation matches the spec. The command-line version can test apps written in any language, simply by passing the file or URL path to the schema to check!

libarchimedes makes it easy to use Hypothesis in the Hy language, a Lisp embedded in Python.

battle_tested is a fuzzing tool that will show you how your code can fail - by trying all kinds of inputs and reporting whatever happens.

pytest-subtesthack functions as a workaround for issue #377.

17.3 Writing an Extension

See CONTRIBUTING.rst *for more information.*

New strategies can be added to Hypothesis, or published as an external package on PyPI - either is fine for most strategies. If in doubt, ask!

It's generally much easier to get things working outside, because there's more freedom to experiment and fewer requirements in stability and API style. We're happy to review and help with external packages as well as pull requests!

If you're thinking about writing an extension, please name it `hypothesis-{something}` - a standard prefix makes the community more visible and searching for extensions easier.

On the other hand, being inside gets you access to some deeper implementation features (if you need them) and better long-term guarantees about maintenance. We particularly encourage pull requests for new composable primitives that make implementing other strategies easier, or for widely used types in the standard library. Strategies for other things are also welcome; anything with external dependencies just goes in hypothesis.extra.

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.

18.1 3.55.4 - 2018-04-13

This patch makes a variety of minor improvements to the documentation, and improves a few validation messages for invalid inputs.

18.2 3.55.3 - 2018-04-12

This release updates the URL metadata associated with the PyPI package (again). It has no other user visible effects.

18.3 3.55.2 - 2018-04-11

This release updates the URL metadata associated with the PyPI package. It has no other user visible effects.

18.4 3.55.1 - 2018-04-06

This patch relaxes constraints on the expected values returned by the standard library function `hypot()` and the internal helper function `cathetus()`, this to fix near-exact test-failures on some 32-bit systems.

18.5 3.55.0 - 2018-04-05

This release includes several improvements to the handling of the `database` setting.

- The `database_file` setting was a historical artefact, and you should just use `database` directly.

- The `HYPOTHESIS_DATABASE_FILE` environment variable is deprecated, in favor of `load_profile()` and the `database` setting.

- If you have not configured the example database at all and the default location is not usable (due to e.g. permissions issues), Hypothesis will fall back to an in-memory database. This is not persisted between sessions, but means that the defaults work on read-only filesystems.

18.6 3.54.0 - 2018-04-04

This release improves the `complex_numbers()` strategy, which now supports `min_magnitude` and `max_magnitude` arguments, along with `allow_nan` and `allow_infinity` like for `floats()`.

Thanks to J.J. Green for this feature.

18.7 3.53.0 - 2018-04-01

This release removes support for Django 1.8, which reached end of life on 2018-04-01. You can see Django's release and support schedule on the Django Project website.

18.8 3.52.3 - 2018-04-01

This patch fixes the `min_satisfying_examples` settings documentation, by explaining that example shrinking is tracked at the level of the underlying bytestream rather than the output value.

The output from `find()` in verbose mode has also been adjusted - see *the example session* - to avoid duplicating lines when the example repr is constant, even if the underlying representation has been shrunken.

18.9 3.52.2 - 2018-03-30

This release improves the output of failures with *rule based stateful testing* in two ways:

- The output from it is now usually valid Python code.

- When the same value has two different names because it belongs to two different bundles, it will now display with the name associated with the correct bundle for a rule argument where it is used.

18.10 3.52.1 - 2018-03-29

This release improves the behaviour of *stateful testing* in two ways:

- Previously some runs would run no steps (issue #376). This should no longer happen.
- RuleBasedStateMachine tests which used bundles extensively would often shrink terribly. This should now be significantly improved, though there is likely a lot more room for improvement.

This release also involves a low level change to how ranges of integers are handles which may result in other improvements to shrink quality in some cases.

18.11 3.52.0 - 2018-03-24

This release deprecates use of `@settings(...)` as a decorator, on functions or methods that are not also decorated with `@given`. You can still apply these decorators in any order, though you should only do so once each.

Applying `@given` twice was already deprecated, and applying `@settings(...)` twice is deprecated in this release and will become an error in a future version. Neither could ever be used twice to good effect.

Using `@settings(...)` as the sole decorator on a test is completely pointless, so this common usage error will become an error in a future version of Hypothesis.

18.12 3.51.0 - 2018-03-24

This release deprecates the `average_size` argument to `lists()` and other collection strategies. You should simply delete it wherever it was used in your tests, as it no longer has any effect.

In early versions of Hypothesis, the `average_size` argument was treated as a hint about the distribution of examples from a strategy. Subsequent improvements to the conceptual model and the engine for generating and shrinking examples mean it is more effective to simply describe what constitutes a valid example, and let our internals handle the distribution.

18.13 3.50.3 - 2018-03-24

This patch contains some internal refactoring so that we can run with warnings as errors in CI.

18.14 3.50.2 - 2018-03-20

This has no user-visible changes except one slight formatting change to one docstring, to avoid a deprecation warning.

18.15 3.50.1 - 2018-03-20

This patch fixes an internal error introduced in 3.48.0, where a check for the Django test runner would expose import-time errors in Django configuration (issue #1167).

18.16 3.50.0 - 2018-03-19

This release improves validation of numeric bounds for some strategies.

- *integers()* and *floats()* now raise `InvalidArgument` if passed a `min_value` or `max_value` which is not an instance of `Real`, instead of various internal errors.

- *floats()* now converts its bounding values to the nearest float above or below the min or max bound respectively, instead of just casting to float. The old behaviour was incorrect in that you could generate `float(min_value)`, even when this was less than `min_value` itself (possible with eg. fractions).

- When both bounds are provided to *floats()* but there are no floats in the interval, such as `[(2**54)+1 .. (2**55)-1]`, InvalidArgument is raised.

- *decimals()* gives a more useful error message if passed a string that cannot be converted to `Decimal` in a context where this error is not trapped.

Code that previously **seemed** to work may be explicitly broken if there were no floats between `min_value` and `max_value` (only possible with non-float bounds), or if a bound was not a `Real` number but still allowed in `math.isnan` (some custom classes with a __float__ method).

18.17 3.49.1 - 2018-03-15

This patch fixes our tests for Numpy dtype strategies on big-endian platforms, where the strategy behaved correctly but the test assumed that the native byte order was little-endian.

There is no user impact unless you are running our test suite on big-endian platforms. Thanks to Graham Inggs for reporting issue #1164.

18.18 3.49.0 - 2018-03-12

This release deprecates passing `elements=None` to collection strategies, such as *lists()*.

Requiring `lists(nothing())` or `builds(list)` instead of `lists()` means slightly more typing, but also improves the consistency and discoverability of our API - as well as showing how to compose or construct strategies in ways that still work in more complex situations.

Passing a nonzero max_size to a collection strategy where the elements strategy contains no values is now deprecated, and will be an error in a future version. The equivalent with `elements=None` is already an error.

18.19 3.48.1 - 2018-03-05

This patch will minimize examples that would come out non-minimal in previous versions. Thanks to Kyle Reeve for this patch.

18.20 3.48.0 - 2018-03-05

This release improves some "unhappy paths" when using Hypothesis with the standard library `unittest` module:

- Applying *@given* to a non-test method which is overridden from `unittest.TestCase`, such as `setUp`, raises `a new health check`. (issue #991)

- Using `subTest()` within a test decorated with `@given` would leak intermediate results when tests were run under the `unittest` test runner. Individual reporting of failing subtests is now disabled during a test using `@given`. (issue #1071)

- `@given` is still not a class decorator, but the error message if you try using it on a class has been improved.

As a related improvement, using `django.test.TestCase` with `@given` instead of `hypothesis.extra. django.TestCase` raises an explicit error instead of running all examples in a single database transaction.

18.21 3.47.0 - 2018-03-02

`register_profile` now accepts keyword arguments for specific settings, and the parent settings object is now optional. Using a `name` for a registered profile which is not a string was never suggested, but it is now also deprecated and will eventually be an error.

18.22 3.46.2 - 2018-03-01

This release removes an unnecessary branch from the code, and has no user-visible impact.

18.23 3.46.1 - 2018-03-01

This changes only the formatting of our docstrings and should have no user-visible effects.

18.24 3.46.0 - 2018-02-26

`characters()` has improved docs about what arguments are valid, and additional validation logic to raise a clear error early (instead of e.g. silently ignoring a bad argument). Categories may be specified as the Unicode 'general category' (eg `u'Nd'`), or as the 'major category' (eg `[u'N', u'Lu']` is equivalent to `[u'Nd', u'Nl', u'No', u'Lu']`).

In previous versions, general categories were supported and all other input was silently ignored. Now, major categories are supported in addition to general categories (which may change the behaviour of some existing code), and all other input is deprecated.

18.25 3.45.5 - 2018-02-26

This patch improves strategy inference in `hypothesis.extra.django` to account for some validators in addition to field type - see issue #1116 for ongoing work in this space.

Specifically, if a `CharField` or `TextField` has an attached `RegexValidator`, we now use `from_regex()` instead of `text()` as the underlying strategy. This allows us to generate examples of the default `User` model, closing issue #1112.

18.26 3.45.4 - 2018-02-25

This patch improves some internal debugging information, fixes a typo in a validation error message, and expands the documentation for new contributors.

18.27 3.45.3 - 2018-02-23

This patch may improve example shrinking slightly for some strategies.

18.28 3.45.2 - 2018-02-18

This release makes our docstring style more consistent, thanks to flake8-docstrings. There are no user-visible changes.

18.29 3.45.1 - 2018-02-17

This fixes an indentation issue in docstrings for `datetimes()`, `dates()`, `times()`, and `timedeltas()`.

18.30 3.45.0 - 2018-02-13

This release fixes `builds()` so that `target` can be used as a keyword argument for passing values to the target. The target itself can still be specified as a keyword argument, but that behavior is now deprecated. The target should be provided as the first positional argument.

18.31 3.44.26 - 2018-02-06

This release fixes some formatting issues in the Hypothesis source code. It should have no externally visible effects.

18.32 3.44.25 - 2018-02-05

This release changes the way in which Hypothesis tries to shrink the size of examples. It probably won't have much impact, but might make shrinking faster in some cases. It is unlikely but not impossible that it will change the resulting examples.

18.33 3.44.24 - 2018-01-27

This release fixes dependency information when installing Hypothesis from a binary "wheel" distribution.

- The `install_requires` for enum34 is resolved at install time, rather than at build time (with potentially different results).
- Django has fixed their `python_requires` for versions 2.0.0 onward, simplifying Python2-compatible constraints for downstream projects.

18.34 3.44.23 - 2018-01-24

This release improves shrinking in a class of pathological examples that you are probably never hitting in practice. If you *are* hitting them in practice this should be a significant speed up in shrinking. If you are not, you are very unlikely to notice any difference. You might see a slight slow down and/or slightly better falsifying examples.

18.35 3.44.22 - 2018-01-23

This release fixes a dependency problem. It was possible to install Hypothesis with an old version of attrs, which would throw a `TypeError` as soon as you tried to import hypothesis. Specifically, you need attrs 16.0.0 or newer.

Hypothesis will now require the correct version of attrs when installing.

18.36 3.44.21 - 2018-01-22

This change adds some additional structural information that Hypothesis will use to guide its search.

You mostly shouldn't see much difference from this. The two most likely effects you would notice are:

1. Hypothesis stores slightly more examples in its database for passing tests.
2. Hypothesis *may* find new bugs that it was previously missing, but it probably won't (this is a basic implementation of the feature that is intended to support future work. Although it is useful on its own, it's not *very* useful on its own).

18.37 3.44.20 - 2018-01-21

This is a small refactoring release that changes how Hypothesis tracks some information about the boundary of examples in its internal representation.

You are unlikely to see much difference in behaviour, but memory usage and run time may both go down slightly during normal test execution, and when failing Hypothesis might print its failing example slightly sooner.

18.38 3.44.19 - 2018-01-21

This changes how we compute the default `average_size` for all collection strategies. Previously setting a `max_size` without setting an `average_size` would have the seemingly paradoxical effect of making data generation *slower*, because it would raise the average size from its default. Now setting `max_size` will either leave the default unchanged or lower it from its default.

If you are currently experiencing this problem, this may make your tests substantially faster. If you are not, this will likely have no effect on you.

18.39 3.44.18 - 2018-01-20

This is a small refactoring release that changes how Hypothesis detects when the structure of data generation depends on earlier values generated (e.g. when using *flatmap* or `composite()`). It should not have any observable effect on behaviour.

18.40 3.44.17 - 2018-01-15

This release fixes a typo in internal documentation, and has no user-visible impact.

18.41 3.44.16 - 2018-01-13

This release improves test case reduction for recursive data structures. Hypothesis now guarantees that whenever a strategy calls itself recursively (usually this will happen because you are using `deferred()`), any recursive call may replace the top level value. e.g. given a tree structure, Hypothesis will always try replacing it with a subtree.

Additionally this introduces a new heuristic that may in some circumstances significantly speed up test case reduction - Hypothesis should be better at immediately replacing elements drawn inside another strategy with their minimal possible value.

18.42 3.44.15 - 2018-01-13

`from_type()` can now resolve recursive types such as binary trees (issue #1004). Detection of non-type arguments has also improved, leading to better error messages in many cases involving **forward references**.

18.43 3.44.14 - 2018-01-08

This release fixes a bug in the shrinker that prevented the optimisations in 3.44.6 from working in some cases. It would not have worked correctly when filtered examples were nested (e.g. with a set of integers in some range).

This would not have resulted in any correctness problems, but shrinking may have been slower than it otherwise could be.

18.44 3.44.13 - 2018-01-08

This release changes the average bit length of values drawn from `integers()` to be much smaller. Additionally it changes the shrinking order so that now size is considered before sign - e.g. -1 will be preferred to +10.

The new internal format for integers required some changes to the minimizer to make work well, so you may also see some improvements to example quality in unrelated areas.

18.45 3.44.12 - 2018-01-07

This changes Hypothesis's internal implementation of weighted sampling. This will affect example distribution and quality, but you shouldn't see any other effects.

18.46 3.44.11 - 2018-01-06

This is a change to some internals around how Hypothesis handles avoiding generating duplicate examples and seeking out novel regions of the search space.

You are unlikely to see much difference as a result of it, but it fixes a bug where an internal assertion could theoretically be triggered and has some minor effects on the distribution of examples so could potentially find bugs that have previously been missed.

18.47 3.44.10 - 2018-01-06

This patch avoids creating debug statements when debugging is disabled. Profiling suggests this is a 5-10% performance improvement (issue #1040).

18.48 3.44.9 - 2018-01-06

This patch blacklists null characters (`'\x00'`) in automatically created strategies for Django `CharField` and `TextField`, due to a database issue which was recently fixed upstream (Hypothesis issue #1045).

18.49 3.44.8 - 2018-01-06

This release makes the Hypothesis shrinker slightly less greedy in order to avoid local minima - when it gets stuck, it makes a small attempt to search around the final example it would previously have returned to find a new starting point to shrink from. This should improve example quality in some cases, especially ones where the test data has dependencies among parts of it that make it difficult for Hypothesis to proceed.

18.50 3.44.7 - 2018-01-04

This release adds support for Django 2 in the hypothesis-django extra.

This release drops support for Django 1.10, as it is no longer supported by the Django team.

18.51 3.44.6 - 2018-01-02

This release speeds up test case reduction in many examples by being better at detecting large shrinks it can use to discard redundant parts of its input. This will be particularly noticeable in examples that make use of filtering and for some integer ranges.

18.52 3.44.5 - 2018-01-02

Happy new year!

This is a no-op release that updates the year range on all of the copyright headers in our source to include 2018.

18.53 3.44.4 - 2017-12-23

This release fixes issue #1044, which slowed tests by up to 6% due to broken caching.

18.54 3.44.3 - 2017-12-21

This release improves the shrinker in cases where examples drawn earlier can affect how much data is drawn later (e.g. when you draw a length parameter in a composite and then draw that many elements). Examples found in cases like this should now be much closer to minimal.

18.55 3.44.2 - 2017-12-20

This is a pure refactoring release which changes how Hypothesis manages its set of examples internally. It should have no externally visible effects.

18.56 3.44.1 - 2017-12-18

This release fixes issue #997, in which under some circumstances the body of tests run under Hypothesis would not show up when run under coverage even though the tests were run and the code they called outside of the test file would show up normally.

18.57 3.44.0 - 2017-12-17

This release adds a new feature: The *@reproduce_failure*, designed to make it easy to use Hypothesis's binary format for examples to reproduce a problem locally without having to share your example database between machines.

This also changes when seeds are printed:

- They will no longer be printed for normal falsifying examples, as there are now adequate ways of reproducing those for all cases, so it just contributes noise.
- They will once again be printed when reusing examples from the database, as health check failures should now be more reliable in this scenario so it will almost always work in this case.

This work was funded by Smarkets.

18.58 3.43.1 - 2017-12-17

This release fixes a bug with Hypothesis's database management - examples that were found in the course of shrinking were saved in a way that indicated that they had distinct causes, and so they would all be retried on the start of the next test. The intended behaviour, which is now what is implemented, is that only a bounded subset of these examples would be retried.

18.59 3.43.0 - 2017-12-17

`HypothesisDeprecationWarning` now inherits from `FutureWarning` instead of `DeprecationWarning`, as recommended by **PEP 565** for user-facing warnings (issue #618). If you have not changed the default warnings settings, you will now see each distinct `HypothesisDeprecationWarning` instead of only the first.

18.60 3.42.2 - 2017-12-12

This patch fixes issue #1017, where instances of a list or tuple subtype used as an argument to a strategy would be coerced to tuple.

18.61 3.42.1 - 2017-12-10

This release has some internal cleanup, which makes reading the code more pleasant and may shrink large examples slightly faster.

18.62 3.42.0 - 2017-12-09

This release deprecates *hypothesis[fakefactory]*, which was designed as a transition strategy but does not support example shrinking or coverage-guided discovery.

18.63 3.41.0 - 2017-12-06

`sampled_from()` can now sample from one-dimensional numpy ndarrays. Sampling from multi-dimensional ndarrays still results in a deprecation warning. Thanks to Charlie Tanksley for this patch.

18.64 3.40.1 - 2017-12-04

This release makes two changes:

- It makes the calculation of some of the metadata that Hypothesis uses for shrinking occur lazily. This should speed up performance of test case generation a bit because it no longer calculates information it doesn't need.

- It improves the shrinker for certain classes of nested examples. e.g. when shrinking lists of lists, the shrinker is now able to concatenate two adjacent lists together into a single list. As a result of this change, shrinking may get somewhat slower when the minimal example found is large.

18.65 3.40.0 - 2017-12-02

This release improves how various ways of seeding Hypothesis interact with the example database:

- Using the example database with `seed()` is now deprecated. You should set `database=None` if you are doing that. This will only warn if you actually load examples from the database while using @seed.

- The `derandomize` will behave the same way as @seed.

- Using `--hypothesis-seed` will disable use of the database.

- If a test used examples from the database, it will not suggest using a seed to reproduce it, because that won't work.

This work was funded by Smarkets.

18.66 3.39.0 - 2017-12-01

This release adds a new health check that checks if the smallest "natural" possible example of your test case is very large - this will tend to cause Hypothesis to generate bad examples and be quite slow.

This work was funded by Smarkets.

18.67 3.38.9 - 2017-11-29

This is a documentation release to improve the documentation of shrinking behaviour for Hypothesis's strategies.

18.68 3.38.8 - 2017-11-29

This release improves the performance of `characters()` when using `blacklist_characters` and `from_regex()` when using negative character classes.

The problems this fixes were found in the course of work funded by Smarkets.

18.69 3.38.7 - 2017-11-29

This is a patch release for `from_regex()`, which had a bug in handling of the `re.VERBOSE` flag (issue #992). Flags are now handled correctly when parsing regex.

18.70 3.38.6 - 2017-11-28

This patch changes a few byte-string literals from double to single quotes, thanks to an update in unify. There are no user-visible changes.

18.71 3.38.5 - 2017-11-23

This fixes the repr of strategies using lambda that are defined inside decorators to include the lambda source.

This would mostly have been visible when using the *statistics* functionality - lambdas used for e.g. filtering would have shown up with a `<unknown>` as their body. This can still happen, but it should happen less often now.

18.72 3.38.4 - 2017-11-22

This release updates the reported *statistics* so that they show approximately what fraction of your test run time is spent in data generation (as opposed to test execution).

This work was funded by Smarkets.

18.73 3.38.3 - 2017-11-21

This is a documentation release, which ensures code examples are up to date by running them as doctests in CI (issue #711).

18.74 3.38.2 - 2017-11-21

This release changes the behaviour of the `deadline` setting when used with `data()`: Time spent inside calls to `data.draw` will no longer be counted towards the deadline time.

As a side effect of some refactoring required for this work, the way flaky tests are handled has changed slightly. You are unlikely to see much difference from this, but some error messages will have changed.

This work was funded by Smarkets.

18.75 3.38.1 - 2017-11-21

This patch has a variety of non-user-visible refactorings, removing various minor warts ranging from indirect imports to typos in comments.

18.76 3.38.0 - 2017-11-18

This release overhauls *the health check system* in a variety of small ways. It adds no new features, but is nevertheless a minor release because it changes which tests are likely to fail health checks.

The most noticeable effect is that some tests that used to fail health checks will now pass, and some that used to pass will fail. These should all be improvements in accuracy. In particular:

- New failures will usually be because they are now taking into account things like use of `data()` and `assume()` inside the test body.
- New failures *may* also be because for some classes of example the way data generation performance was measured was artificially faster than real data generation (for most examples that are hitting performance health checks the opposite should be the case).
- Tests that used to fail health checks and now pass do so because the health check system used to run in a way that was subtly different than the main Hypothesis data generation and lacked some of its support for e.g. large examples.

If your data generation is especially slow, you may also see your tests get somewhat faster, as there is no longer a separate health check phase. This will be particularly noticeable when rerunning test failures.

This work was funded by Smarkets.

18.77 3.37.0 - 2017-11-12

This is a deprecation release for some health check related features.

The following are now deprecated:

- Passing *exception_in_generation* to *suppress_health_check*. This no longer does anything even when passed - All errors that occur during data generation will now be immediately reraised rather than going through the health check mechanism.

- Passing *random_module* to *suppress_health_check*. This hasn't done anything for a long time, but was never explicitly deprecated. Hypothesis always seeds the random module when running @given tests, so this is no longer an error and suppressing it doesn't do anything.

- Passing non-*HealthCheck* values in *suppress_health_check*. This was previously allowed but never did anything useful.

In addition, passing a non-iterable value as *suppress_health_check* will now raise an error immediately (it would never have worked correctly, but it would previously have failed later). Some validation error messages have also been updated.

This work was funded by Smarkets.

18.78 3.36.1 - 2017-11-10

This is a yak shaving release, mostly concerned with our own tests.

While `getfullargspec()` was documented as deprecated in Python 3.5, it never actually emitted a warning. Our code to silence this (nonexistent) warning has therefore been removed.

We now run our tests with `DeprecationWarning` as an error, and made some minor changes to our own tests as a result. This required similar upstream updates to coverage and execnet (a test-time dependency via pytest-xdist).

There is no user-visible change in Hypothesis itself, but we encourage you to consider enabling deprecations as errors in your own tests.

18.79 3.36.0 - 2017-11-06

This release adds a setting to the public API, and does some internal cleanup:

- The *derandomize* setting is now documented (issue #890)

- Removed - and disallowed - all 'bare excepts' in Hypothesis (issue #953)

- Documented the *strict* setting as deprecated, and updated the build so our docs always match deprecations in the code.

18.80 3.35.0 - 2017-11-06

This minor release supports constraining *uuids()* to generate a particular version of `UUID` (issue #721).

Thanks to Dion Misic for this feature.

18.81 3.34.1 - 2017-11-02

This patch updates the documentation to suggest *builds(callable)* instead of *just(callable())*.

18.82 3.34.0 - 2017-11-02

Hypothesis now emits deprecation warnings if you apply *@given* more than once to a target.

Applying *@given* repeatedly wraps the target multiple times. Each wrapper will search the space of of possible parameters separately. This is equivalent but will be much more inefficient than doing it with a single call to *@given*.

For example, instead of @given(booleans()) @given(integers()), you could write @given(booleans(), integers())

18.83 3.33.1 - 2017-11-02

This is a bugfix release:

- *builds()* would try to infer a strategy for required positional arguments of the target from type hints, even if they had been given to *builds()* as positional arguments (issue #946). Now it only infers missing required arguments.

- An internal introspection function wrongly reported self as a required argument for bound methods, which might also have affected *builds()*. Now it knows better.

18.84 3.33.0 - 2017-10-16

This release supports strategy inference for more field types in Django models() - you can now omit an argument for Date, Time, Duration, Slug, IP Address, and UUID fields. (issue #642)

Strategy generation for fields with grouped choices now selects choices from each group, instead of selecting from the group names.

18.85 3.32.2 - 2017-10-15

This patch removes the mergedb tool, introduced in Hypothesis 1.7.1 on an experimental basis. It has never actually worked, and the new *Hypothesis example database* is designed to make such a tool unnecessary.

18.86 3.32.1 - 2017-10-13

This patch has two improvements for strategies based on enumerations.

- *from_type()* now handles enumerations correctly, delegating to *sampled_from()*. Previously it noted that Enum.__init__ has no required arguments and therefore delegated to *builds()*, which would subsequently fail.

- When sampling from an enum.Flag, we also generate combinations of members. Eg for Flag('Permissions', 'READ, WRITE, EXECUTE') we can now generate, Permissions. READ, Permissions.READ|WRITE, and so on.

18.87 3.32.0 - 2017-10-09

This changes the default value of `use_coverage=True` to True when running on pypy (it was already True on CPython).

It was previously set to False because we expected it to be too slow, but recent benchmarking shows that actually performance of the feature on pypy is fairly acceptable - sometimes it's slower than on CPython, sometimes it's faster, but it's generally within a factor of two either way.

18.88 3.31.6 - 2017-10-08

This patch improves the quality of strategies inferred from Numpy dtypes:

- Integer dtypes generated examples with the upper half of their (non-sign) bits set to zero. The inferred strategies can now produce any representable integer.
- Fixed-width unicode- and byte-string dtypes now cap the internal example length, which should improve example and shrink quality.
- Numpy arrays can only store fixed-size strings internally, and allow shorter strings by right-padding them with null bytes. Inferred string strategies no longer generate such values, as they can never be retrieved from an array. This improves shrinking performance by skipping useless values.

This has already been useful in Hypothesis - we found an overflow bug in our Pandas support, and as a result `indexes()` and `range_indexes()` now check that `min_size` and `max_size` are at least zero.

18.89 3.31.5 - 2017-10-08

This release fixes a performance problem in tests where `use_coverage` is set to True.

Tests experience a slow-down proportionate to the amount of code they cover. This is still the case, but the factor is now low enough that it should be unnoticeable. Previously it was large and became much larger in 3.28.4.

18.90 3.31.4 - 2017-10-08

`from_type()` failed with a very confusing error if passed a `NewType()` (issue #901). These psudeo-types are now unwrapped correctly, and strategy inference works as expected.

18.91 3.31.3 - 2017-10-06

This release makes some small optimisations to our use of coverage that should reduce constant per-example overhead. This is probably only noticeable on examples where the test itself is quite fast. On no-op tests that don't test anything you may see up to a fourfold speed increase (which is still significantly slower than without coverage). On more realistic tests the speed up is likely to be less than that.

18.92 3.31.2 - 2017-09-30

This release fixes some formatting and small typos/grammar issues in the documentation, specifically the page docs/settings.rst, and the inline docs for the various settings.

18.93 3.31.1 - 2017-09-30

This release improves the handling of deadlines so that they act better with the shrinking process. This fixes issue #892.

This involves two changes:

1. The deadline is raised during the initial generation and shrinking, and then lowered to the set value for final replay. This restricts our attention to examples which exceed the deadline by a more significant margin, which increases their reliability.

2. When despite the above a test still becomes flaky because it is significantly faster on rerun than it was on its first run, the error message is now more explicit about the nature of this problem, and includes both the initial test run time and the new test run time.

In addition, this release also clarifies the documentation of the deadline setting slightly to be more explicit about where it applies.

This work was funded by Smarkets.

18.94 3.31.0 - 2017-09-29

This release blocks installation of Hypothesis on Python 3.3, which **reached its end of life date on 2017-09-29**.

This should not be of interest to anyone but downstream maintainers - if you are affected, migrate to a secure version of Python as soon as possible or at least seek commercial support.

18.95 3.30.4 - 2017-09-27

This release makes several changes:

1. It significantly improves Hypothesis's ability to use coverage information to find interesting examples.

2. It reduces the default `max_examples` setting from 200 to 100. This takes advantage of the improved algorithm meaning fewer examples are typically needed to get the same testing and is sufficiently better at covering interesting behaviour, and offsets some of the performance problems of running under coverage.

3. Hypothesis will always try to start its testing with an example that is near minimized.

The new algorithm for 1 also makes some changes to Hypothesis's low level data generation which apply even with coverage turned off. They generally reduce the total amount of data generated, which should improve test performance somewhat. Between this and 3 you should see a noticeable reduction in test runtime (how much so depends on your tests and how much example size affects their performance. On our benchmarks, where data generation dominates, we saw up to a factor of two performance improvement, but it's unlikely to be that large.

18.96 3.30.3 - 2017-09-25

This release fixes some formatting and small typos/grammar issues in the documentation, specifically the page docs/details.rst, and some inline docs linked from there.

18.97 3.30.2 - 2017-09-24

This release changes Hypothesis's caching approach for functions in `hypothesis.strategies`. Previously it would have cached extremely aggressively and cache entries would never be evicted. Now it adopts a least-frequently used, least recently used key invalidation policy, and is somewhat more conservative about which strategies it caches.

Workloads which create strategies based on dynamic values, e.g. by using *flatmap* or `composite()`, will use significantly less memory.

18.98 3.30.1 - 2017-09-22

This release fixes a bug where when running with `use_coverage=True` inside an existing running instance of coverage, Hypothesis would frequently put files that the coveragerc excluded in the report for the enclosing coverage.

18.99 3.30.0 - 2017-09-20

This release introduces two new features:

- When a test fails, either with a health check failure or a falsifying example, Hypothesis will print out a seed that led to that failure, if the test is not already running with a fixed seed. You can then recreate that failure using either the `@seed` decorator or (if you are running pytest) with `--hypothesis-seed`.
- pytest users can specify a seed to use for `@given` based tests by passing the `--hypothesis-seed` command line argument.

This work was funded by Smarkets.

18.100 3.29.0 - 2017-09-19

This release makes Hypothesis coverage aware. Hypothesis now runs all test bodies under coverage, and uses this information to guide its testing.

The `use_coverage` setting can be used to disable this behaviour if you want to test code that is sensitive to coverage being enabled (either because of performance or interaction with the trace function).

The main benefits of this feature are:

- Hypothesis now observes when examples it discovers cover particular lines or branches and stores them in the database for later.
- Hypothesis will make some use of this information to guide its exploration of the search space and improve the examples it finds (this is currently used only very lightly and will likely improve significantly in future releases).

This also has the following side-effects:

- Hypothesis now has an install time dependency on the coverage package.

- Tests that are already running Hypothesis under coverage will likely get faster.

- Tests that are not running under coverage now run their test bodies under coverage by default.

This feature is only partially supported under pypy. It is significantly slower than on CPython and is turned off by default as a result, but it should still work correctly if you want to use it.

18.101 3.28.3 - 2017-09-18

This release is an internal change that affects how Hypothesis handles calculating certain properties of strategies.

The primary effect of this is that it fixes a bug where use of `deferred()` could sometimes trigger an internal assertion error. However the fix for this bug involved some moderately deep changes to how Hypothesis handles certain constructs so you may notice some additional knock-on effects.

In particular the way Hypothesis handles drawing data from strategies that cannot generate any values has changed to bail out sooner than it previously did. This may speed up certain tests, but it is unlikely to make much of a difference in practice for tests that were not already failing with Unsatisfiable.

18.102 3.28.2 - 2017-09-18

This is a patch release that fixes a bug in the *hypothesis.extra.pandas* documentation where it incorrectly referred to *column()* instead of *columns()*.

18.103 3.28.1 - 2017-09-16

This is a refactoring release. It moves a number of internal uses of `namedtuple()` over to using attrs based classes, and removes a couple of internal namedtuple classes that were no longer in use.

It should have no user visible impact.

18.104 3.28.0 - 2017-09-15

This release adds support for testing pandas via the *hypothesis.extra.pandas* module.

It also adds a dependency on attrs.

This work was funded by Stripe.

18.105 3.27.1 - 2017-09-14

This release fixes some formatting and broken cross-references in the documentation, which includes editing docstrings - and thus a patch release.

18.106 3.27.0 - 2017-09-13

This release introduces a `deadline` setting to Hypothesis.

When set this turns slow tests into errors. By default it is unset but will warn if you exceed 200ms, which will become the default value in a future release.

This work was funded by Smarkets.

18.107 3.26.0 - 2017-09-12

Hypothesis now emits deprecation warnings if you are using the legacy SQLite example database format, or the tool for merging them. These were already documented as deprecated, so this doesn't change their deprecation status, only that we warn about it.

18.108 3.25.1 - 2017-09-12

This release fixes a bug with generating numpy datetime and timedelta types: When inferring the strategy from the dtype, datetime and timedelta dtypes with sub-second precision would always produce examples with one second resolution. Inferring a strategy from a time dtype will now always produce example with the same precision.

18.109 3.25.0 - 2017-09-12

This release changes how Hypothesis shrinks and replays examples to take into account that it can encounter new bugs while shrinking the bug it originally found. Previously it would end up replacing the originally found bug with the new bug and show you only that one. Now it is (often) able to recognise when two bugs are distinct and when it finds more than one will show both.

18.110 3.24.2 - 2017-09-11

This release removes the (purely internal and no longer useful) `strategy_test_suite` function and the corresponding strategytests module.

18.111 3.24.1 - 2017-09-06

This release improves the reduction of examples involving floating point numbers to produce more human readable examples.

It also has some general purpose changes to the way the minimizer works internally, which may see some improvement in quality and slow down of test case reduction in cases that have nothing to do with floating point numbers.

18.112 3.24.0 - 2017-09-05

Hypothesis now emits deprecation warnings if you use `some_strategy.example()` inside a test function or strategy definition (this was never intended to be supported, but is sufficiently widespread that it warrants a deprecation path).

18.113 3.23.3 - 2017-09-05

This is a bugfix release for `decimals()` with the `places` argument.

- No longer fails health checks (issue #725, due to internal filtering)

- Specifying a `min_value` and `max_value` without any decimals with `places` places between them gives a more useful error message.

- Works for any valid arguments, regardless of the decimal precision context.

18.114 3.23.2 - 2017-09-01

This is a small refactoring release that removes a now-unused parameter to an internal API. It shouldn't have any user visible effect.

18.115 3.23.1 - 2017-09-01

Hypothesis no longer propagates the dynamic scope of settings into strategy definitions.

This release is a small change to something that was never part of the public API and you will almost certainly not notice any effect unless you're doing something surprising, but for example the following code will now give a different answer in some circumstances:

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

CURRENT_SETTINGS = st.builds(lambda: settings.default)
```

(We don't actually encourage you writing code like this)

Previously this would have generated the settings that were in effect at the point of definition of `CURRENT_SETTINGS`. Now it will generate the settings that are used for the current test.

It is very unlikely to be significant enough to be visible, but you may also notice a small performance improvement.

18.116 3.23.0 - 2017-08-31

This release adds a `unique` argument to `arrays()` which behaves the same ways as the corresponding one for `lists()`, requiring all of the elements in the generated array to be distinct.

18.117 3.22.2 - 2017-08-29

This release fixes an issue where Hypothesis would raise a `TypeError` when using the datetime-related strategies if running with `PYTHONOPTIMIZE=2`. This bug was introduced in v3.20.0. (See issue #822)

18.118 3.22.1 - 2017-08-28

Hypothesis now transparently handles problems with an internal unicode cache, including file truncation or read-only filesystems (issue #767). Thanks to Sam Hames for the patch.

18.119 3.22.0 - 2017-08-26

This release provides what should be a substantial performance improvement to numpy arrays generated using *provided numpy support*, and adds a new `fill_value` argument to *arrays()* to control this behaviour.

This work was funded by Stripe.

18.120 3.21.3 - 2017-08-26

This release fixes some extremely specific circumstances that probably have never occurred in the wild where users of `deferred()` might have seen a RuntimeError from too much recursion, usually in cases where no valid example could have been generated anyway.

18.121 3.21.2 - 2017-08-25

This release fixes some minor bugs in argument validation:

- *hypothesis.extra.numpy* dtype strategies would raise an internal error instead of an InvalidArgument exception when passed an invalid endianness specification.
- *fractions()* would raise an internal error instead of an InvalidArgument if passed `float("nan")` as one of its bounds.
- The error message for passing `float("nan")` as a bound to various strategies has been improved.
- Various bound arguments will now raise InvalidArgument in cases where they would previously have raised an internal TypeError or ValueError from the relevant conversion function.
- *streaming()* would not have emitted a deprecation warning when called with an invalid argument.

18.122 3.21.1 - 2017-08-24

This release fixes a bug where test failures that were the result of an *@example* would print an extra stack trace before re-raising the exception.

18.123 3.21.0 - 2017-08-23

This release deprecates Hypothesis's strict mode, which turned Hypothesis's deprecation warnings into errors. Similar functionality can be achieved by using `simplefilter('error', HypothesisDeprecationWarning)`.

18.124 3.20.0 - 2017-08-22

This release renames the relevant arguments on the `datetimes()`, `dates()`, `times()`, and `timedeltas()` strategies to `min_value` and `max_value`, to make them consistent with the other strategies in the module.

The old argument names are still supported but will emit a deprecation warning when used explicitly as keyword arguments. Arguments passed positionally will go to the new argument names and are not deprecated.

18.125 3.19.3 - 2017-08-22

This release provides a major overhaul to the internals of how Hypothesis handles shrinking.

This should mostly be visible in terms of getting better examples for tests which make heavy use of `composite()`, `data()` or *flatmap* where the data drawn depends a lot on previous choices, especially where size parameters are affected. Previously Hypothesis would have struggled to reliably produce good examples here. Now it should do much better. Performance should also be better for examples with a non-zero `min_size`.

You may see slight changes to example generation (e.g. improved example diversity) as a result of related changes to internals, but they are unlikely to be significant enough to notice.

18.126 3.19.2 - 2017-08-21

This release fixes two bugs in `hypothesis.extra.numpy`:

- `unicode_string_dtypes()` didn't work at all due to an incorrect dtype specifier. Now it does.
- Various impossible conditions would have been accepted but would error when they fail to produced any example. Now they raise an explicit InvalidArgument error.

18.127 3.19.1 - 2017-08-21

This is a bugfix release for issue #739, where bounds for `fractions()` or floating-point `decimals()` were not properly converted to integers before passing them to the integers strategy. This excluded some values that should have been possible, and could trigger internal errors if the bounds lay between adjacent integers.

You can now bound `fractions()` with two arbitrarily close fractions.

It is now an explicit error to supply a min_value, max_value, and max_denominator to `fractions()` where the value bounds do not include a fraction with denominator at most max_denominator.

18.128 3.19.0 - 2017-08-20

This release adds the `from_regex()` strategy, which generates strings that contain a match of a regular expression.

Thanks to Maxim Kulkin for creating the hypothesis-regex package and then helping to upstream it! (issue #662)

18.129 3.18.5 - 2017-08-18

This is a bugfix release for `integers()`. Previously the strategy would hit an internal assertion if passed non-integer bounds for `min_value` and `max_value` that had no integers between them. The strategy now raises InvalidArgument instead.

18.130 3.18.4 - 2017-08-18

Release to fix a bug where mocks can be used as test runners under certain conditions. Specifically, if a mock is injected into a test via pytest fixtures or patch decorators, and that mock is the first argument in the list, hypothesis will think it represents self and turns the mock into a test runner. If this happens, the affected test always passes because the mock is executed instead of the test body. Sometimes, it will also fail health checks.

Fixes issue #491 and a section of issue #198. Thanks to Ben Peterson for this bug fix.

18.131 3.18.3 - 2017-08-17

This release should improve the performance of some tests which experienced a slow down as a result of the 3.13.0 release.

Tests most likely to benefit from this are ones that make extensive use of `min_size` parameters, but others may see some improvement as well.

18.132 3.18.2 - 2017-08-16

This release fixes a bug introduced in 3.18.0. If the arguments `whitelist_characters` and `blacklist_characters` to `characters()` both contained elements, then an `InvalidArgument` exception would be raised.

Thanks to Zac Hatfield-Dodds for reporting and fixing this.

18.133 3.18.1 - 2017-08-14

This is a bug fix release to fix issue #780, where `sets()` and similar would trigger health check errors if their element strategy could only produce one element (e.g. if it was `just()`).

18.134 3.18.0 - 2017-08-13

This is a feature release:

- `characters()` now accepts `whitelist_characters`, particular characters which will be added to those it produces. (issue #668)

- A bug fix for the internal function `_union_interval_lists()`, and a rename to `_union_intervals()`. It now correctly handles all cases where intervals overlap, and it always returns the result as a tuple for tuples.

Thanks to Alex Willmer for these.

18.135 3.17.0 - 2017-08-07

This release documents *the previously undocumented phases feature*, making it part of the public API. It also updates how the example database is used. Principally:

- A `Phases.reuse` argument will now correctly control whether examples from the database are run (it previously did exactly the wrong thing and controlled whether examples would be *saved*).

- Hypothesis will no longer try to rerun *all* previously failing examples. Instead it will replay the smallest previously failing example and a selection of other examples that are likely to trigger any other bugs that will found. This prevents a previous failure from dominating your tests unnecessarily.

- As a result of the previous change, Hypothesis will be slower about clearing out old examples from the database that are no longer failing (because it can only clear out ones that it actually runs).

18.136 3.16.1 - 2017-08-07

This release makes an implementation change to how Hypothesis handles certain internal constructs.

The main effect you should see is improvement to the behaviour and performance of collection types, especially ones with a `min_size` parameter. Many cases that would previously fail due to being unable to generate enough valid examples will now succeed, and other cases should run slightly faster.

18.137 3.16.0 - 2017-08-04

This release introduces a deprecation of the timeout feature. This results in the following changes:

- Creating a settings object with an explicit timeout will emit a deprecation warning.

- If your test stops because it hits the timeout (and has not found a bug) then it will emit a deprecation warning.

- There is a new value `unlimited` which you can import from hypothesis. `settings(timeout=unlimited)` will *not* cause a deprecation warning.

- There is a new health check, `hung_test`, which will trigger after a test has been running for five minutes if it is not suppressed.

18.138 3.15.0 - 2017-08-04

This release deprecates two strategies, *choices()* and *streaming()*.

Both of these are somewhat confusing to use and are entirely redundant since the introduction of the *data()* strategy for interactive drawing in tests, and their use should be replaced with direct use of *data()* instead.

18.139 3.14.2 - 2017-08-03

This fixes a bug where Hypothesis would not work correctly on Python 2.7 if you had the `typing` module backport installed.

18.140 3.14.1 - 2017-08-02

This raises the maximum depth at which Hypothesis starts cutting off data generation to a more reasonable value which it is harder to hit by accident.

This resolves (issue #751), in which some examples which previously worked would start timing out, but it will also likely improve the data generation quality for complex data types.

18.141 3.14.0 - 2017-07-23

Hypothesis now understands inline type annotations (issue #293):

- If the target of `builds()` has type annotations, a default strategy for missing required arguments is selected based on the type. Type-based strategy selection will only override a default if you pass `hypothesis.infer` as a keyword argument.

- If `@given` wraps a function with type annotations, you can pass `infer` as a keyword argument and the appropriate strategy will be substituted.

- You can check what strategy will be inferred for a type with the new `from_type()` function.

- `register_type_strategy()` teaches Hypothesis which strategy to infer for custom or unknown types. You can provide a strategy, or for more complex cases a function which takes the type and returns a strategy.

18.142 3.13.1 - 2017-07-20

This is a bug fix release for issue #514 - Hypothesis would continue running examples after a `SkipTest` exception was raised, including printing a falsifying example. Skip exceptions from the standard `unittest` module, and `pytest`, `nose`, or `unittest2` modules now abort the test immediately without printing output.

18.143 3.13.0 - 2017-07-16

This release has two major aspects to it: The first is the introduction of `deferred()`, which allows more natural definition of recursive (including mutually recursive) strategies.

The second is a number of engine changes designed to support this sort of strategy better. These should have a knock-on effect of also improving the performance of any existing strategies that currently generate a lot of data or involve heavy nesting by reducing their typical example size.

18.144 3.12.0 - 2017-07-07

This release makes some major internal changes to how Hypothesis represents data internally, as a prelude to some major engine changes that should improve data quality. There are no API changes, but it's a significant enough internal change that a minor version bump seemed warranted.

User facing impact should be fairly mild, but includes:

- All existing examples in the database will probably be invalidated. Hypothesis handles this automatically, so you don't need to do anything, but if you see all your examples disappear that's why.
- Almost all data distributions have changed significantly. Possibly for the better, possibly for the worse. This may result in new bugs being found, but it may also result in Hypothesis being unable to find bugs it previously did.
- Data generation may be somewhat faster if your existing bottleneck was in draw_bytes (which is often the case for large examples).
- Shrinking will probably be slower, possibly significantly.

If you notice any effects you consider to be a significant regression, please open an issue about them.

18.145 3.11.6 - 2017-06-19

This release involves no functionality changes, but is the first to ship wheels as well as an sdist.

18.146 3.11.5 - 2017-06-18

This release provides a performance improvement to shrinking. For cases where there is some non-trivial "boundary" value (e.g. the bug happens for all values greater than some other value), shrinking should now be substantially faster. Other types of bug will likely see improvements too.

This may also result in some changes to the quality of the final examples - it may sometimes be better, but is more likely to get slightly worse in some edge cases. If you see any examples where this happens in practice, please report them.

18.147 3.11.4 - 2017-06-17

This is a bugfix release: Hypothesis now prints explicit examples when running in verbose mode. (issue #313)

18.148 3.11.3 - 2017-06-11

This is a bugfix release: Hypothesis no longer emits a warning if you try to use `sampled_from()` with `collections.OrderedDict`. (issue #688)

18.149 3.11.2 - 2017-06-10

This is a documentation release. Several outdated snippets have been updated or removed, and many cross-references are now hyperlinks.

18.150 3.11.1 - 2017-05-28

This is a minor ergonomics release. Tracebacks shown by pytest no longer include Hypothesis internals for test functions decorated with *@given*.

18.151 3.11.0 - 2017-05-23

This is a feature release, adding datetime-related strategies to the core strategies.

`timezones()` allows you to sample pytz timezones from the Olsen database. Use directly in a recipe for tz-aware datetimes, or compose with *none()* to allow a mix of aware and naive output.

The new *dates()*, *times()*, *datetimes()*, and *timedeltas()* strategies are all constrained by objects of their type. This means that you can generate dates bounded by a single day (i.e. a single date), or datetimes constrained to the microsecond.

times() and *datetimes()* take an optional `timezones=` argument, which defaults to *none()* for naive times. You can use our extra strategy based on pytz, or roll your own timezones strategy with dateutil or even the standard library.

The old `dates`, `times`, and `datetimes` strategies in `hypothesis.extra.datetimes` are deprecated in favor of the new core strategies, which are more flexible and have no dependencies.

18.152 3.10.0 - 2017-05-22

Hypothesis now uses `inspect.getfullargspec()` internally. On Python 2, there are no visible changes.

On Python 3 *@given* and *@composite* now preserve **PEP 3107** annotations on the decorated function. Keyword-only arguments are now either handled correctly (e.g. *@composite*), or caught in validation instead of silently discarded or raising an unrelated error later (e.g. *@given*).

18.153 3.9.1 - 2017-05-22

This is a bugfix release: the default field mapping for a DateTimeField in the Django extra now respects the `USE_TZ` setting when choosing a strategy.

18.154 3.9.0 - 2017-05-19

This is feature release, expanding the capabilities of the *decimals()* strategy.

- The new (optional) `places` argument allows you to generate decimals with a certain number of places (e.g. cents, thousandths, satoshis).

- If allow_infinity is None, setting min_bound no longer excludes positive infinity and setting max_value no longer excludes negative infinity.

- All of `NaN`, `-Nan`, `sNaN`, and `-sNaN` may now be drawn if allow_nan is True, or if allow_nan is None and min_value or max_value is None.

- min_value and max_value may be given as decimal strings, e.g. `"1.234"`.

18.155 3.8.5 - 2017-05-16

Hypothesis now imports `sqlite3` when a SQLite database is used, rather than at module load, improving compatibility with Python implementations compiled without SQLite support (such as BSD or Jython).

18.156 3.8.4 - 2017-05-16

This is a compatibility bugfix release. `sampled_from` no longer raises a deprecation warning when sampling from an `Enum`, as all enums have a reliable iteration order.

18.157 3.8.3 - 2017-05-09

This release removes a version check for older versions of pytest when using the Hypothesis pytest plugin. The pytest plugin will now run unconditionally on all versions of pytest. This breaks compatibility with any version of pytest prior to 2.7.0 (which is more than two years old).

The primary reason for this change is that the version check was a frequent source of breakage when pytest change their versioning scheme. If you are not working on pytest itself and are not running a very old version of it, this release probably doesn't affect you.

18.158 3.8.2 - 2017-04-26

This is a code reorganisation release that moves some internal test helpers out of the main source tree so as to not have changes to them trigger releases in future.

18.159 3.8.1 - 2017-04-26

This is a documentation release. Almost all code examples are now doctests checked in CI, eliminating stale examples.

18.160 3.8.0 - 2017-04-23

This is a feature release, adding the *iterables()* strategy, equivalent to `lists(...).map(iter)` but with a much more useful repr. You can use this strategy to check that code doesn't accidentally depend on sequence properties such as indexing support or repeated iteration.

18.161 3.7.4 - 2017-04-22

This is a bug fix release for a single bug:

- In 3.7.3, using *@example* and a pytest fixture in the same test could cause the test to fail to fill the arguments, and throw a TypeError.

18.162 3.7.3 - 2017-04-21

This release should include no user visible changes and is purely a refactoring release. This modularises the behaviour of the core `given()` function, breaking it up into smaller and more accessible parts, but its actual behaviour should remain unchanged.

18.163 3.7.2 - 2017-04-21

This reverts an undocumented change in 3.7.1 which broke installation on debian stable: The specifier for the hypothesis[django] extra_requires had introduced a wild card, which was not supported on the default version of pip.

18.164 3.7.1 - 2017-04-21

This is a bug fix and internal improvements release.

- In particular Hypothesis now tracks a tree of where it has already explored. This allows it to avoid some classes of duplicate examples, and significantly improves the performance of shrinking failing examples by allowing it to skip some shrinks that it can determine can't possibly work.

- Hypothesis will no longer seed the global random arbitrarily unless you have asked it to using `random_module()`

- Shrinking would previously have not worked correctly in some special cases on Python 2, and would have resulted in suboptimal examples.

18.165 3.7.0 - 2017-03-20

This is a feature release.

New features:

- Rule based stateful testing now has an `@invariant` decorator that specifies methods that are run after init and after every step, allowing you to encode properties that should be true at all times. Thanks to Tom Prince for this feature.

- The `decimals()` strategy now supports `allow_nan` and `allow_infinity` flags.

- There are *significantly more strategies available for numpy*, including for generating arbitrary data types. Thanks to Zac Hatfield Dodds for this feature.

- When using the `data()` strategy you can now add a label as an argument to `draw()`, which will be printed along with the value when an example fails. Thanks to Peter Inglesby for this feature.

Bug fixes:

- Bug fix: `composite()` now preserves functions' docstrings.

- The build is now reproducible and doesn't depend on the path you build it from. Thanks to Chris Lamb for this feature.

- numpy strategies for the void data type did not work correctly. Thanks to Zac Hatfield Dodds for this fix.

There have also been a number of performance optimizations:

- The *permutations()* strategy is now significantly faster to use for large lists (the underlying algorithm has gone from O(n^2) to O(n)).

- Shrinking of failing test cases should have got significantly faster in some circumstances where it was previously struggling for a long time.

- Example generation now involves less indirection, which results in a small speedup in some cases (small enough that you won't really notice it except in pathological cases).

18.166 3.6.1 - 2016-12-20

This release fixes a dependency problem and makes some small behind the scenes improvements.

- The fake-factory dependency was renamed to faker. If you were depending on it through hypothesis[django] or hypothesis[fake-factory] without pinning it yourself then it would have failed to install properly. This release changes it so that hypothesis[fakefactory] (which can now also be installed as hypothesis[faker]) will install the renamed faker package instead.

- This release also removed the dependency of hypothesis[django] on hypothesis[fakefactory] - it was only being used for emails. These now use a custom strategy that isn't from fakefactory. As a result you should also see performance improvements of tests which generated User objects or other things with email fields, as well as better shrinking of email addresses.

- The distribution of code using nested calls to *one_of()* or the | operator for combining strategies has been improved, as branches are now flattened to give a more uniform distribution.

- Examples using *composite()* or .flatmap should now shrink better. In particular this will affect things which work by first generating a length and then generating that many items, which have historically not shrunk very well.

18.167 3.6.0 - 2016-10-31

This release reverts Hypothesis to its old pretty printing of lambda functions based on attempting to extract the source code rather than decompile the bytecode. This is unfortunately slightly inferior in some cases and may result in you occasionally seeing things like lambda x: <unknown> in statistics reports and strategy reprs.

This removes the dependencies on uncompyle6, xdis and spark-parser.

The reason for this is that the new functionality was based on uncompyle6, which turns out to introduce a hidden GPLed dependency - it in turn depended on xdis, and although the library was licensed under the MIT license, it contained some GPL licensed source code and thus should have been released under the GPL.

My interpretation is that Hypothesis itself was never in violation of the GPL (because the license it is under, the Mozilla Public License v2, is fully compatible with being included in a GPL licensed work), but I have not consulted a lawyer on the subject. Regardless of the answer to this question, adding a GPLed dependency will likely cause a lot of users of Hypothesis to inadvertently be in violation of the GPL.

As a result, if you are running Hypothesis 3.5.x you really should upgrade to this release immediately.

18.168 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).

18.169 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.

18.170 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 not 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).

18.171 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 **PEP 3107** 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).

18.172 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.

18.173 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.

18.174 3.4.0 - 2016-05-27

This release is entirely provided by Lucas Wiman:

Strategies constructed by `models()` 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.

18.175 3.3.0 - 2016-05-27

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

18.176 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 `random_module()` this will continue to work and will always display the seed.

18.177 3.1.3 - 2016-05-01

Single bug fix release

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

18.178 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.

18.179 3.1.1 - 2016-04-29

Minor bug fix release.

- Fix concurrency issue when running tests that use `text()` from multiple processes at once (issue #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.

18.180 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).

18.181 3.0.5 - 2016-02-25

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

18.182 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).

18.183 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.

18.184 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.

18.185 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.

18.186 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.

18.187 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.

18.188 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.

18.189 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.

18.190 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.

18.191 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.

18.192 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 caused the test to fail but would cause e.g. degraded performance or confusing results.

18.193 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.

18.194 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.

18.195 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

18.196 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).

18.197 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__.

18.198 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.

18.199 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.

18.200 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).

18.201 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).

18.202 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

18.203 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.

18.204 1.10.6 - 2015-08-26

Fix support for fixtures on Django 1.7.

18.205 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.

18.206 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.

18.207 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.

18.208 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.

18.209 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.

18.210 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.

18.211 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.

18.212 1.8.3 - 2015-07-20

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

18.213 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.

18.214 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.

18.215 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.

18.216 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.

18.217 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.

18.218 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.

18.219 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.

18.220 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.

18.221 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.

18.222 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.

18.223 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.

18.224 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.

18.225 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.

18.226 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

18.227 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.

18.228 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.

18.229 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.

18.230 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.

18.231 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.

18.232 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.

18.233 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].

18.234 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)

18.235 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.

18.236 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.

18.237 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.

18.238 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.

18.239 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...

18.240 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.

18.241 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.

18.242 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.

18.243 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.

18.244 0.2.2 - 2015-01-08

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

18.245 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.

18.246 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

18.247 0.1.4 - 2013-12-14

- Make verification runs time bounded with a configurable timeout

18.248 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

18.249 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.

18.250 0.1.1 - 2013-03-24

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

18.251 0.1.0 - 2013-03-23

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

18.252 0.0.5 - 2013-03-13

- No changes except trying to fix packaging

18.253 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

18.254 0.0.3 - 2013-03-13

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

18.255 0.0.2 - 2013-03-12

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

18.256 0.0.1 - 2013-03-10

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

Ongoing Hypothesis Development

Hypothesis development is 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, the maintenance team takes an active role in shepherding pull requests and helping people write a new feature (see CONTRIBUTING.rst for details and pull request #154 for an example of how the process goes). This isn't "patches welcome", it's "we will help you write a patch".

19.1 Release Policy

Hypothesis releases follow semantic versioning.

We maintain backwards-compatibility wherever possible, and use deprecation warnings to mark features that have been superseded by a newer alternative. If you want to detect this, you can `upgrade warnings to errors in the usual ways`.

We use continuous deployment to ensure that you can always use our newest and shiniest features - every change to the source tree is automatically built and published on PyPI as soon as it's merged onto master, after code review and passing our extensive test suite.

19.2 Project Roadmap

Hypothesis does not have a long-term release plan. However some visibility into our plans for future *compatibility* may be useful:

- We value compatibility, and maintain it as far as practical. This generally excludes things which are end-of-life upstream, or have an unstable API.

- We would like to drop Python 2 support when it reaches end of life in 2020. Ongoing support is likely to depend on commercial funding.

- We intend to support PyPy3 as soon as it supports a recent enough version of Python 3. See issue #602.

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.

To see the versions of Python, optional dependencies, test runners, and operating systems Hypothesis supports (meaning incompatibility is treated as a bug), see *Compatibility*.

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.

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

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

21.1 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 and should not be considered complete releases. Requests to include additional files in them will not be granted. Their absence is not a bug.

21.2 Dependencies

21.2.1 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
- CPython 3.6.x

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

21.2.2 Other Python libraries

Hypothesis has *mandatory* dependencies on the following libraries:

- attrs

- coverage

- enum34 is required on Python 2.7

Hypothesis has *optional* dependencies on the following libraries:

- pytz (almost any version should work)

- Faker, version 0.7 or later

- Django, all supported versions

- numpy, 1.10 or later (earlier versions will probably work fine)

- pandas, 1.8 or later

- py.test (2.8.0 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.

21.3 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 2.8.0 is strongly encouraged, but it may work with earlier versions (however py.test specific logic is disabled before 2.8.0).

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

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

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

21.4 Examples

- arch linux
- fedora
- gentoo

Reproducing Failures

One of the things that is often concerning for people using randomized testing like Hypothesis is the question of how to reproduce failing test cases.

Fortunately Hypothesis has a number of features in support of this. The one you will use most commonly when developing locally is *the example database*, which means that you shouldn't have to think about the problem at all for local use - test failures will just automatically reproduce without you having to do anything.

The example database is perfectly suitable for sharing between machines, but there currently aren't very good work flows for that, so Hypothesis provides a number of ways to make examples reproducible by adding them to the source code of your tests. This is particularly useful when e.g. you are trying to run an example that has failed on your CI, or otherwise share them between machines.

22.1 Providing explicit examples

You can explicitly ask Hypothesis to try a particular example, using

hypothesis.**example**(**args*, ***kwargs*)
> A decorator which ensures a specific example is always tested.

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 either of the following styles will work as expected:

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


from unittest import TestCase
```

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

As with @given, 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.

22.2 Reproducing a test run with `@seed`

hypothesis.**seed**(*seed*)
> seed: Start the test execution from a specific seed.

> May be any hashable object. No exact meaning for seed is provided other than that for a fixed seed value Hypothesis will try the same actions (insofar as it can given external sources of non- determinism. e.g. timing and hash randomization).

> Overrides the derandomize setting, which is designed to enable deterministic builds rather than reproducing observed failures.

When a test fails unexpectedly, usually due to a health check failure, Hypothesis will print out a seed that led to that failure, if the test is not already running with a fixed seed. You can then recreate that failure using either the @seed decorator or (if you are running pytest) with --hypothesis-seed.

22.3 Reproducing an example with with `@reproduce_failure`

Hypothesis has an opaque binary representation that it uses for all examples it generates. This representation is not intended to be stable across versions or with respect to changes in the test, but can be used to to reproduce failures with the @reproduce_example decorator.

hypothesis.**reproduce_failure**(*version*, *blob*)
> Run the example that corresponds to this data blob in order to reproduce a failure.

> A test with this decorator *always* runs only one example and always fails. If the provided example does not cause a failure, or is in some way invalid for this test, then this will fail with a DidNotReproduce error.

> This decorator is not intended to be a permanent addition to your test suite. It's simply some code you can add to ease reproduction of a problem in the event that you don't have access to the test database. Because of this, *no* compatibility guarantees are made between different versions of Hypothesis - its API may change arbitrarily from version to version.

The intent is that you should never write this decorator by hand, but it is instead provided by Hypothesis. When a test fails with a falsifying example, Hypothesis may print out a suggestion to use @reproduce_failure on the test to recreate the problem as follows:

```
>>> from hypothesis import settings, given, PrintSettings
>>> import hypothesis.strategies as st
>>> @given(st.floats())
... @settings(print_blob=PrintSettings.ALWAYS)
... def test(f):
...     assert f == f
```

```
...
>>> try:
...      test()
... except AssertionError:
...      pass
Falsifying example: test(f=nan)

You can reproduce this example by temporarily adding @reproduce_failure(..., b'AAAA//
↪AAAAAAAAEA') as a decorator on your test case
```

Adding the suggested decorator to the test should reproduce the failure (as long as everything else is the same - changing the versions of Python or anything else involved, might of course affect the behaviour of the test! Note that changing the version of Hypothesis will result in a different error - each @reproduce_failure invocation is specific to a Hypothesis version).

When to do this is controlled by the print_blob setting, which may be one of the following values:

class hypothesis.**PrintSettings**

Flags to determine whether or not to print a detailed example blob to use with *reproduce_failure()* for failing test cases.

NEVER = 0

Never print a blob.

INFER = 1

Make an educated guess as to whether it would be appropriate to print the blob.

The current rules are that this will print if both:

1. The output from Hypothesis appears to be unsuitable for use with *example()*.

2. The output is not too long.

ALWAYS = 2

Always print a blob on failure.

Python Module Index

h

Index