<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Quick start guide</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Details and advanced features</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Settings</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>What you can generate and how</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>First-party extensions</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>Ghostwriting tests for you</td>
<td>49</td>
</tr>
<tr>
<td>7</td>
<td>Hypothesis for Django users</td>
<td>53</td>
</tr>
<tr>
<td>8</td>
<td>Hypothesis for the scientific stack</td>
<td>57</td>
</tr>
<tr>
<td>9</td>
<td>Health checks</td>
<td>71</td>
</tr>
<tr>
<td>10</td>
<td>The Hypothesis example database</td>
<td>73</td>
</tr>
<tr>
<td>11</td>
<td>Stateful testing</td>
<td>77</td>
</tr>
<tr>
<td>12</td>
<td>Compatibility</td>
<td>85</td>
</tr>
<tr>
<td>13</td>
<td>Some more examples</td>
<td>87</td>
</tr>
<tr>
<td>14</td>
<td>Community</td>
<td>95</td>
</tr>
<tr>
<td>15</td>
<td>The purpose of Hypothesis</td>
<td>97</td>
</tr>
<tr>
<td>16</td>
<td>Testimonials</td>
<td>99</td>
</tr>
<tr>
<td>17</td>
<td>Open source projects using Hypothesis</td>
<td>103</td>
</tr>
<tr>
<td>18</td>
<td>Projects extending Hypothesis</td>
<td>105</td>
</tr>
<tr>
<td>19</td>
<td>Changelog</td>
<td>109</td>
</tr>
<tr>
<td>20</td>
<td>Ongoing Hypothesis development</td>
<td>259</td>
</tr>
<tr>
<td>21</td>
<td>Help and support</td>
<td>261</td>
</tr>
<tr>
<td>22</td>
<td>Packaging guidelines</td>
<td>263</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>23 Reproducing failures</td>
<td>265</td>
<td></td>
</tr>
<tr>
<td>Python Module Index</td>
<td>269</td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>271</td>
<td></td>
</tr>
</tbody>
</table>
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 arbitrary 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.
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
                entry = (character, count)
                lst.append(entry)
    return lst

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

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

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

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

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

(For this example we’ll just let pytest discover and run the test. We’ll cover other ways you could have run it later).

The text function returns what Hypothesis calls a search strategy. An object with methods that describe how to generate
and simplify certain kinds of values. The @given decorator then takes our test function and turns it into a parametrized
one which, when called, will run the test function over a wide range of matching data from that strategy.

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

Falsifying example: test_decode_inverts_encode(s='')

UnboundLocalError: local variable 'character' referenced before assignment

Hypothesis correctly points out that this code is simply wrong if called on an empty string.

If we fix that by just adding the following code to the beginning of the function then Hypothesis tells us the code is
correct (by doing nothing as you’d expect a passing test to).

if not input_string:
    return []

If we wanted to make sure this example was always checked we could add it in explicitly by using the @example
decorator:

from hypothesis import example, given, strategies as st

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

This can be useful to show other developers (or your future self) what kinds of data are valid inputs, or to ensure that
particular edge cases such as '' are tested every time. It’s also great for regression tests because although Hypothesis
will remember failing examples, we don’t recommend distributing that database.

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:

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

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

def encode(input_string):
    count = 1

(continues on next page)
```
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
    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 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
```

You can install the dependencies for optional extensions with e.g. `pip install hypothesis[pandas,django]`.

If you want to install directly from the source code (e.g. because you want to make changes and install the changed version), check out the instructions in CONTRIBUTING.rst.

### 1.3 Running tests

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

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

We could also have done this as a unittest.TestCase:

```
import unittest
```

(continues on next page)
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, 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
```

(continues on next page)
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 arbitrary 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 a value with a `__repr__()` method 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
hypothesis.note(value)
```

Report this value for the minimal failing example.

```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(f"Shuffle: {ls2!r}")
...     assert ls == ls2
... try:
...     test_shuffle_is_noop()
... except AssertionError:
...     print("ls != ls2")

Falsifying example: test_shuffle_is_noop(ls=[0, 1], r=RandomWithSeed(1))
Shuffle: [1, 0]
ls != ls2
```

The note is printed for the minimal failing example 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:

- during generate phase (0.06 seconds):
  - Typical runtimes: < 1ms, ~ 47% in data generation
  - 100 passing examples, 0 failing examples, 0 invalid examples
- 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:
  - during generate phase (0.08 seconds):
    - Typical runtimes: < 1ms, ~ 57% in data generation
    - 100 passing examples, 0 failing examples, 12 invalid examples
    - Events:
      * 51.79%, Retried draw from integers().filter(lambda x: x % 2 == 0) to satisfy filter
      * 10.71%, Aborted test because unable to satisfy integers().filter(lambda x: x % 2 == 0)
  - Stopped because settings.max_examples=100
```

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

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

@given(st.integers().filter(lambda x: x % 2 == 0))
def test_even_integers(i):
    event(f"i mod 3 = {i%3}")

You will then see output like:

test_even_integers:

- during generate phase (0.09 seconds):
  - Typical runtimes: < 1ms, ~ 59% in data generation
  - 100 passing examples, 0 failing examples, 32 invalid examples
  - Events:
    * 54.55%, Retried draw from integers().filter(lambda x: x % 2 == 0) to satisfy...
    * 31.06%, i mod 3 = 2
    * 28.79%, i mod 3 = 0
    * 24.24%, Aborted test because unable to satisfy integers().filter(lambda x: x %...
    * 2 == 0)
    * 15.91%, i mod 3 = 1
  - Stopped because settings.max_examples=100

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:

```
@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 let’s block off this particular example:

```python
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:

```python
@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:

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

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

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

Adding `assume(all(x > 0 for x in xs))` and it passes: 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) > 1)
    assume(all(x > 0 for x in xs))
    print(xs)
    assert sum(xs) > 0

In: test_sum_is_positive()

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, but not a mixture of both.

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()
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`. Check the notes on framework compatibility to see how this affects other testing libraries you may be using.

### 2.6 Targeted example generation

Targeted property-based testing combines the advantages of both search-based and property-based testing. Instead of being completely random, T-PBT uses a search-based component to guide the input generation towards values that have a higher probability of falsifying a property. This explores the input space more effectively and requires fewer tests to find a bug or achieve a high confidence in the system being tested than random PBT. (Löschler and Sagonas)

This is not always a good idea - for example calculating the search metric might take time better spent running more uniformly-random test cases - but Hypothesis has experimental support for targeted PBT you may wish to try.

```python
hypothesis.target(observation, *, label='')
```

Calling this function with an `int` or `float` observation gives it feedback with which to guide our search for inputs that will cause an error, in addition to all the usual heuristics. Observations must always be finite.

Hypothesis will try to maximize the observed value over several examples; almost any metric will work so long as it makes sense to increase it. For example, `-abs(error)` is a metric that increases as `error` approaches zero.

Example metrics:

- Number of elements in a collection, or tasks in a queue
- Mean or maximum runtime of a task (or both, if you use `label`)
- Compression ratio for data (perhaps per-algorithm or per-level)
- Number of steps taken by a state machine

The optional `label` argument can be used to distinguish between and therefore separately optimise distinct observations, such as the mean and standard deviation of a dataset. It is an error to call `target()` with any label more than once per test case.

Note: The more examples you run, the better this technique works.

As a rule of thumb, the targeting effect is noticeable above `max_examples=1000`, and immediately obvious by around ten thousand examples per `label` used by your test.

---

Test statistics include the best score seen for each label, which can help avoid the threshold problem when the minimal example shrinks right down to the threshold of failure (issue #2180).
We recommend that users also skim the papers introducing targeted PBT; from ISSTA 2017 and ICST 2018. For the curious, the initial implementation in Hypothesis uses hill-climbing search via a mutating fuzzer, with some tactics inspired by simulated annealing to avoid getting stuck and endlessly mutating a local maximum.

### 2.7 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
class TestRunTwice(Testcase):
    def execute_example(self, f):
        return f()()
```

(continues on next page)
result = f()
if callable(result):
    result = result()
return result

An alternative hook is provided for use by test runner extensions such as pytest-trio, which cannot use the execute_example method. This is not recommended for end-users - it is better to write a complete test function directly, perhaps by using a decorator to perform the same transformation before applying @given.

@given(x=integers())
@pytest.mark.trio
async def test(x):
    ...

# Illustrative code, inside the pytest-trio plugin
test.hypothesis.inner_test = lambda x: trio.run(test, x)

For authors of test runners however, assigning to the inner_test attribute of the hypothesis attribute of the test will replace the interior test.

Note: The new inner_test must accept and pass through all the *args and **kwargs expected by the original test.

If the end user has also specified a custom executor using the execute_example method, it - and all other execution-time logic - will be applied to the new inner test assigned by the test runner.

### 2.8 Making random code deterministic

While Hypothesis’ example generation can be used for nondeterministic tests, debugging anything nondeterministic is usually a very frustrating exercise. To make things worse, our example shrinking relies on the same input causing the same failure each time - though we show the un-shrunk failure and a decent error message if it doesn’t.

By default, Hypothesis will handle the global random and numpy.random random number generators for you, and you can register others:

```python
hypothesis.register_random(r)
```

Register the given Random instance for management by Hypothesis.

You can pass random.Random instances (or other objects with seed, getstate, and setstate methods) to register_random(r) to have their states seeded and restored in the same way as the global PRNGs from the random and numpy.random modules.

All global PRNGs, from e.g. simulation or scheduling frameworks, should be registered to prevent flaky tests. Hypothesis will ensure that the PRNG state is consistent for all test runs, or reproducibly varied if you choose to use the random_module() strategy.
2.9 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 `signature()`). 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.

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

`hypothesis.infer`

`@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.

```python
@given(a=infer)
def test(a: int):
  pass

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

2.9.1 Limitations

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 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.

2.10 Type annotations in Hypothesis

If you install Hypothesis and use `mypy` 0.590+, or another PEP 561-compatible tool, the type checker should automatically pick up our type hints.

**Note:** Hypothesis' type hints may make breaking changes between minor releases.

Upstream tools and conventions about type hints remain in flux - for example the `typing` module itself is provisional, and Mypy has not yet reached version 1.0 - and we plan to support the latest version of this ecosystem, as well as older versions where practical.
We may also find more precise ways to describe the type of various interfaces, or change their type and runtime behaviour together in a way which is otherwise backwards-compatible. We often omit type hints for deprecated features or arguments, as an additional form of warning.

There are known issues inferring the type of examples generated by `deferred()`, `recursive()`, `one_of()`, `dictionaries()`, and `fixed_dictionaries()`. We will fix these, and require correspondingly newer versions of Mypy for type hinting, as the ecosystem improves.

### 2.10.1 Writing downstream type hints

Projects that provide Hypothesis strategies and use type hints may wish to annotate their strategies too. This is a supported use-case, again on a best-effort provisional basis. For example:

```python
def foo_strategy() -> SearchStrategy[Foo]:
...
```

```python
class hypothesis.strategies.SearchStrategy

SearchStrategy is the type of all strategy objects. It is a generic type, and covariant in the type of the examples it creates. For example:

- `integers()` is of type `SearchStrategy[int].`
- `lists(integers())` is of type `SearchStrategy[List[int]].`
- `SearchStrategy[Dog]` is a subtype of `SearchStrategy[Animal]` if `Dog` is a subtype of `Animal` (as seems likely).
```

**Warning:** `SearchStrategy` should only be used in type hints. Please do not inherit from, compare to, or otherwise use it in any way outside of type hints. The only supported way to construct objects of this type is to use the functions provided by the `hypothesis.strategies` module!

### 2.11 The Hypothesis pytest plugin

Hypothesis includes a tiny plugin to improve integration with pytest, which is activated by default (but does not affect other test runners). It aims to improve the integration between Hypothesis and Pytest by providing extra information and convenient access to config options.

- `pytest --hypothesis-show-statistics` can be used to display test and data generation statistics.
- `pytest --hypothesis-profile=<profile name>` can be used to load a settings profile. `pytest --hypothesis-verbosity=<level name>` can be used to override the current verbosity level.
- `pytest --hypothesis-seed=<an int>` can be used to reproduce a failure with a particular seed.
- `pytest --hypothesis-explain` can be used to temporarily enable the explain phase.

Finally, all tests that are defined with Hypothesis automatically have `@pytest.mark.hypothesis` applied to them. See here for information on working with markers.

**Note:** Pytest will load the plugin automatically if Hypothesis is installed. You don’t need to do anything at all to use it.
2.12 Use with external fuzzers

Tip:
Want an integrated workflow for your team’s local tests, CI, and continuous fuzzing? Use HypoFuzz to fuzz your whole test suite, and find more bugs without more tests!

Sometimes, you might want to point a traditional fuzzer such as python-afl, pythonfuzz, or Google’s atheris (for Python and native extensions) at your code. Wouldn’t it be nice if you could use any of your @given tests as fuzz targets, instead of converting bytestrings into your objects by hand?

```python
@given(st.text())
def test_foo(s):
    ...

# This is a traditional fuzz target - call it with a bytestring,
# or a binary IO object, and it runs the test once.
fuzz_target = test_foo.hypothesis.fuzz_one_input

# For example:
fuzz_target(b'\x00\x00\x00\x00\x00\x00\x00\x00')
fuzz_target(io.BytesIO(...))
```

Depending on the input to fuzz_one_input, one of three things will happen:

- If the bytestring was invalid, for example because it was too short or failed a filter or `assume()` too many times, fuzz_one_input returns None.
- If the bytestring was valid and the test passed, fuzz_one_input returns a canonicalised and pruned buffer which will replay that test case. This is provided as an option to improve the performance of mutating fuzzers, but can safely be ignored.
- If the test failed, i.e. raised an exception, fuzz_one_input will add the pruned buffer to the Hypothesis example database and then re-raise that exception. All you need to do to reproduce, minimize, and de-duplicate all the failures found via fuzzing is run your test suite!

Note that the interpretation of both input and output bytestrings is specific to the exact version of Hypothesis you are using and the strategies given to the test, just like the example database and @reproduce_failure decorator.

2.12.1 Interaction with settings

fuzz_one_input uses just enough of Hypothesis’ internals to drive your test function with a fuzzer-provided bytestring, and most settings therefore have no effect in this mode. We recommend running your tests the usual way before fuzzing to get the benefits of healthchecks, as well as afterwards to replay, shrink, deduplicate, and report whatever errors were discovered.

- The database setting is used by fuzzing mode - adding failures to the database to be replayed when you next run your tests is our preferred reporting mechanism and response to the ‘fuzzer taming’ problem.
- The verbosity and stateful_step_count settings work as usual.

The deadline, derandomize, max_examples, phases, print_blob, report_multiple_bugs, and suppress_health_check settings do not affect fuzzing mode.
2.13 Thread-Safety Policy

As discussed in issue #2719, Hypothesis is not truly thread-safe and that’s unlikely to change in the future. This policy therefore describes what you can expect if you use Hypothesis with multiple threads.

**Running tests in multiple processes**, e.g. with `pytest -n auto`, is fully supported and we test this regularly in CI - thanks to process isolation, we only need to ensure that `DirectoryBasedExampleDatabase` can’t tread on its own toes too badly. If you find a bug here we will fix it ASAP.

**Running separate tests in multiple threads** is not something we design or test for, and is not formally supported. That said, anecdotally it does mostly work and we would like it to keep working - we accept reasonable patches and low-priority bug reports. The main risks here are global state, shared caches, and cached strategies.

**Using multiple threads within a single test**, or running a single test simultaneously in multiple threads, makes it pretty easy to trigger internal errors. We usually accept patches for such issues unless readability or single-thread performance suffer.

Hypothesis assumes that tests are single-threaded, or do a sufficiently-good job of pretending to be single-threaded. Tests that use helper threads internally should be OK, but the user must be careful to ensure that test outcomes are still deterministic. In particular it counts as nondeterministic if helper-thread timing changes the sequence of dynamic draws using e.g. the `data()`.

Interacting with any Hypothesis APIs from helper threads might do weird/bad things, so avoid that too - we rely on thread-local variables in a few places, and haven’t explicitly tested/audited how they respond to cross-thread API calls. While `data()` and equivalents are the most obvious danger, other APIs might also be subtly affected.
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:

```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

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

### 3.1 Available settings

```python
class hypothesis.settings(parent=None, *, max_examples=not_set, derandomize=not_set, database=not_set, verbosity=not_set, phases=not_set, stateful_step_count=not_set, report_multiple_bugs=not_set, suppress_health_check=not_set, deadline=not_set, print_blob=not_set)
```

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

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

**database**

An instance of `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.
See the example database documentation for a list of built-in example database implementations, and how to define custom implementations.

default value: (dynamically calculated)

deadline
If set, a duration (as timedelta, or integer or float number of milliseconds) 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.

default value: timedelta(milliseconds=200)

derandomize
If True, seed Hypothesis’ random number generator using a hash of the test function, so that every run will test the same set of examples until you update Hypothesis, Python, or the test function.

This allows you to check for regressions and look for bugs using separate settings profiles - for example running quick deterministic tests on every commit, and a longer non-deterministic nightly testing run.

default value: False

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

The default value is chosen to suit a workflow where the test will be part of a suite that is regularly executed locally or on a CI server, balancing total running time against the chance of missing a bug.

If you are writing one-off tests, running tens of thousands of examples is quite reasonable as Hypothesis may miss uncommon bugs with default settings. For very complex code, we have observed Hypothesis finding novel bugs after several million examples while testing SymPy. If you are running more than 100k examples for a test, consider using our integration for coverage-guided fuzzing - it really shines when given minutes or hours to run.

default value: 100

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

default value: (Phase.explicit, Phase.reuse, Phase.generate, Phase.target, Phase.shrink)

print_blob
If set to True, Hypothesis will print code for failing examples that can be used with @reproduce_failure to reproduce the failing example. The default is True if the CI or TF_BUILD env vars are set, False otherwise.

default value: (dynamically calculated)

report_multiple_bugs
Because Hypothesis runs the test many times, it can sometimes find multiple bugs in a single run. Reporting all of them at once is usually very useful, but replacing the exceptions can occasionally clash with debuggers.

If disabled, only the exception with the smallest minimal example is raised.

default value: True

stateful_step_count
Number of steps to run a stateful program for before giving up on it breaking.

default value: 50
**suppress_health_check**
A list of `HealthCheck` items to disable.

default value: ()

**verbosity**
Control the verbosity level of Hypothesis messages

default value: `Verbosity.normal`

### 3.1.1 Controlling what runs

Hypothesis divides tests into 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
4. Mutating examples for *targeted property-based testing*.
5. Attempting to shrink an example found in previous phases (other than phase 1 - explicit examples cannot be shrunk). This turns potentially large and complicated examples which may be hard to read into smaller and simpler ones.
6. Attempting to explain the cause of the failure, by identifying suspicious lines of code (e.g. the earliest lines which are never run on passing inputs, and always run on failures). This relies on `sys.settrace()`, and is therefore automatically disabled on PyPy or if you are using `coverage` or a debugger. If there are no clearly suspicious lines of code, *we refuse the temptation to guess*.

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:

```python
class hypothesis.Phase

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.target controls whether examples will be mutated for targeting.
5. Phase.shrink controls whether examples will be shrunk.
6. Phase.explain controls whether Hypothesis attempts to explain test failures.
```

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.

```python
>>> from hypothesis import find, settings, Verbosity
>>> from hypothesis.strategies import lists, integers
>>> @given(lists(integers()))
... @settings(verbosity=Verbosity.verbose)
... def f(x):
...     assert not any(x)
... f()
Trying example: []
Falsifying example: [-1198601713, -67, 116, -29578]
Shrunk example to [-1198601713]
Shrunk example to [-1198601600]
Shrunk example to [-1191228800]
Shrunk example to [-8421504]
Shrunk example to [-32896]
Shrunk example to [-128]
Shrunk example to [64]
Shrunk example to [32]
Shrunk example to [16]
Shrunk example to [8]
Shrunk example to [4]
Shrunk example to [3]
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.

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:

```python
>>> from hypothesis import settings
>>> settings().max_examples
100
>>> settings(max_examples=10).max_examples
10
```

You can also pass a ‘parent’ settings object as the first argument, and any settings you do not specify as keyword arguments will be copied from the parent settings:

```python
>>> parent = settings(max_examples=10)
>>> child = settings(parent, deadline=None)
>>> parent.max_examples == child.max_examples == 10
True
```
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.

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.

```
static settings.register_profile(name, parent=None, **kwargs)
    Registers a collection of values to be used as a settings profile.

    Settings profiles can be loaded by name - for example, you might create a ‘fast’ profile which runs fewer examples, keep the ‘default’ profile, and create a ‘ci’ profile that increases the number of examples and uses a different database to store failures.

    The arguments to this method are exactly as for settings: optional parent settings, and keyword arguments for each setting that will be set differently to parent (or settings.default, if parent is None).

static settings.get_profile(name)
    Return the profile with the given name.

static settings.load_profile(name)
    Loads in the settings defined in the profile provided.

    If the profile does not exist, InvalidArgument will be raised. Any setting not defined in the profile will be the library defined default for that setting.

Loading a profile changes the default settings but will not change the behaviour 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.
Optional, 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.

```python
>>> 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("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`.

```
$ pytest tests --hypothesis-profile <profile-name>
```
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.

### 4.1 Core strategies

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

**hypothesis.strategies.binary(*, min_size=0, max_size=None)**

Generates bytes.

The generated bytes will have a length of at least min_size and at most max_size. If max_size is None there is no upper limit.

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

**hypothesis.strategies.booleans()**

Returns a strategy which generates instances of bool.

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

**hypothesis.strategies.builds(target, /, *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.

If the callable is a class defined with attrs, missing required arguments will be inferred from the attribute on a best-effort basis, e.g. by checking attrs standard validators. Dataclasses are handled natively by the inference from type hints.

Examples from this strategy shrink by shrinking the argument values to the callable.
hypothesis.strategies.characters(*, whitelist_categories=None, blacklist_categories=None, blacklist_characters=None, min_codepoint=None, max_codepoint=None, whitelist_characters=None)

Generates characters, length-one strings, following specified filtering rules.

- When no filtering rules are specified, any character can be produced.
- If `min_codepoint` or `max_codepoint` is specified, 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 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.maxunicode`. 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.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; a value of `None` corresponds an infinite upper bound.

If `min_magnitude` is nonzero 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 constraints 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, ...)` or `@composite` respectively.

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.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.

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.

```python
class hypothesis.strategies.DataObject(data)
    This type only exists so that you can write type hints for tests using the data() strategy. Do not use it directly!
```

```python
hypothesis.strategies.dates(min_value=datetime.date.min, max_value=datetime.date.max)
    A strategy for dates between min_value and max_value.
    Examples from this strategy shrink towards January 1st 2000.
```

```python
hypothesis.strategies.datetimes(min_value=datetime.datetime.min, max_value=datetime.datetime.max, *,
    timezones=none(), allow_imaginary=True)
    A strategy for generating datetimes, which may be timezone-aware.
    This strategy works by drawing a naive datetime between min_value and max_value, which must both be naive (have no timezone).
    timezones must be a strategy that generates either None, for naive datetimes, or tzinfo objects for ‘aware’ datetimes. You can construct your own, though we recommend using the dateutil package and hypothesis.extra.dateutil.timezones() strategy, and also provide hypothesis.extra.pytz.timezones().
    You may pass allow_imaginary=False to filter out “imaginary” datetimes which did not (or will not) occur due to daylight savings, leap seconds, timezone and calendar adjustments, etc. Imaginary datetimes are allowed by default, because malformed timestamps are a common source of bugs.
    Examples from this strategy shrink towards midnight on January 1st 2000, local time.
```

```python
hypothesis.strategies.decimals(min_value=None, max_value=None, *
    allow_nan=None, allow_infinity=None, places=None)
    Generates instances of decimal.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.
```

```python
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:
```
```
```
```
```
```
Mutual recursion also works fine:

```python
>>> 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.

```python
hypothesis.strategies.dictionaries(keys, values, *, dict_class=<class 'dict'>, min_size=0,
    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.

```python
class hypothesis.strategies.DrawFn
```

This type only exists so that you can write type hints for functions decorated with `@composite`.

```python
@composite
def list_and_index(draw: DrawFn) -> Tuple[int, str]:
    i = draw(integers())  # type inferred as 'int'
    s = draw(text())      # type inferred as 'str'
```

```python
hypothesis.strategies.emails()
```

A strategy for generating email addresses as unicode strings. The address format is specified in RFC 5322#section-3.4.1. Values shrink towards shorter local-parts and host domains.

This strategy is useful for generating “user data” for tests, as mishandling of email addresses is a common source of bugs.

```python
hypothesis.strategies.fixed_dictionaries(mapping, *, optional=None)
```

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, in iteration order, with the corresponding values drawn from `mapping[key]`.

If `optional` is passed, the generated value `may or may not` contain each key from `optional` and a value drawn from the corresponding strategy. Generated values may contain optional keys in an arbitrary order.

Examples from this strategy shrink by shrinking each individual value in the generated dictionary, and omitting optional key-value pairs.

```python
hypothesis.strategies.floats(min_value=None, max_value=None, *, allow_nan=None,
    allow_infinity=None, width=64, exclude_min=False, exclude_max=False)
```

Returns a strategy which generates floats.

- If `min_value` is not None, all values will be `>= min_value` (or `> min_value` if `exclude_min`).
- If `max_value` is not None, all values will be `<= max_value` (or `< max_value` if `exclude_max`).
• 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.

The width argument specifies the maximum number of bits of precision required to represent the generated float. Valid values are 16, 32, or 64. Passing width=32 will still use the builtin 64-bit float class, but always for values which can be exactly represented as a 32-bit float.

The exclude_min and exclude_max argument can be used to generate numbers from open or half-open intervals, by excluding the respective endpoints. Excluding either signed zero will also exclude the other. Attempting to exclude an endpoint which is None will raise an error; use allow_infinity=False to generate finite floats. You can however use e.g. min_value=-math.inf, exclude_min=True to exclude only one infinite endpoint.

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.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.from_regex(regex, *, fullmatch=False)
```

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 or re.DOTALL 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". Alternatively, passing fullmatch=True will ensure that the whole string is a match, as if you had used the \A and \Z markers.

Examples from this strategy shrink towards shorter strings and lower character values, with exact behaviour that may depend on the pattern.

```
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.

4.1. Core strategies
The resolution logic may be changed in a future version, but currently tries these five 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, and is not an abstract class, it is resolved via `builds()`.
5. Because abstract types cannot be instantiated, we treat abstract types as the union of their concrete subclasses. Note that this lookup works via inheritance but not via `register`, so you may still need to use `register_type_strategy()`.

There is a valuable recipe for leveraging `from_type()` to generate “everything except” values from a specified type. I.e.

```python
def everything_except(excluded_types):
    return 
    from_type(type)
    .flatmap(from_type)
    .filter(lambda x: not isinstance(x, excluded_types))
```

For example, `everything_except(int)` returns a strategy that can generate anything that `from_type()` can ever generate, except for instances of `int`, and excluding instances of types added via `register_type_strategy()`.

This is useful when writing tests which check that invalid input is rejected in a certain way.

### `hypothesis.strategies.frozensets(elements, *, min_size=0, max_size=None)`

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

### `hypothesis.strategies.functions(*, like=lambda : ..., returns=none(), pure=False)`

A strategy for functions, which can be used in callbacks.

The generated functions will mimic the interface of `like`, which must be a callable (including a class, method, or function). The return value for the function is drawn from the `returns` argument, which must be a strategy.

If `pure=True`, all arguments passed to the generated function must be hashable, and if passed identical arguments the original return value will be returned again - not regenerated, so beware mutable values.

If `pure=False`, generated functions do not validate their arguments, and may return a different value if called again with the same arguments.

Generated functions can only be called within the scope of the `@given` which created them. This strategy does not support `.example()`.

### `hypothesis.strategies.integers(min_value=None, max_value=None)`

Returns a strategy which generates integers.

If `min_value` is not `None` then all values will be $\geq \text{min\_value}$. If `max_value` is not `None` then all values will be $\leq \text{max\_value}$.

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.ip_addresses(*, v=None, network=None)`

Generate IP addresses - $v=4$ for `IPv4Addresses`, $v=6$ for `IPv6Addresses`, or leave unspecified to allow both versions.
network may be an IPv4Network or IPv6Network, or a string representing a network such as "127.0.0.0/24" or "2001:db8::/32". As well as generating addresses within a particular routable network, this can be used to generate addresses from a reserved range listed in the IANA registries.

If you pass both v and network, they must be for the same version.

hypothesis.strategies.iterables(elements, *, min_size=0, 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.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.lists(elements, *, min_size=0, 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, only the empty list will be drawn.

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 callable or tuple of callables 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]).

If unique_by is a tuple of callables the uniqueness will be respective to each callable.

For example, the following will produce two columns of integers with both columns being unique respectively.

```
>>> twoints = st.tuples(st.integers(), st.integers())
>>> st.lists(twoints, unique_by=(lambda x: x[0], lambda x: x[1]))
```

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

hypothesis.strategies.none()

Return a strategy which only generates None.

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

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.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. \( x = \text{st.deferred}(\lambda: \text{st.none()} | \text{st.tuples}(x, x)) \) will shrink well, but \( x = \text{st.deferred}(\lambda: \text{st.tuples}(x, x) | \text{st.none()} \) will shrink very badly indeed.

**hypothesis.strategies.permutations(values)**

Return a strategy which returns permutations of the ordered collection `values`.

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

**hypothesis.strategies.random_module()**

The Hypothesis engine handles PRNG state for the stdlib and Numpy random modules internally, always seeding them to zero and restoring the previous state after the test.

If having a fixed seed would unacceptably weaken your tests, and you cannot use a `random.Random` instance provided by `randoms()`, this strategy calls `random.seed()` with an arbitrary integer and passes you an opaque object whose repr displays the seed value for debugging. If `numpy.random` is available, that state is also managed.

Examples from these strategy shrink to seeds closer to zero.

**hypothesis.strategies.randoms(*, note_method_calls=False, use_true_random=False)**

Generates instances of `random.Random`. The generated Random instances are of a special HypothesisRandom subclass.

- If `note_method_calls` is set to True, Hypothesis will print the randomly drawn values in any falsifying test case. This can be helpful for debugging the behaviour of randomized algorithms.
- If `use_true_random` is set to True then values will be drawn from their usual distribution, otherwise they will actually be Hypothesis generated values (and will be shrunk accordingly for any failing test case). Setting `use_true_random=False` will tend to expose bugs that would occur with very low probability when it is set to True, and this flag should only be set to True when your code relies on the distribution of values for correctness.

**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 \( S = \text{extend}(\text{base} | S) \). That is, values may be drawn from base, or from any strategy reachable by mixing applications of \( | \) and extend.

An example may clarify: `recursive(boolean(), 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.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 Documentation, Release 6.25.0

Note that you may not register a parametrised generic type (such as `MyCollection[int]`) directly, because the resolution logic does not handle this case correctly. Instead, you may register a function for `MyCollection` and inspect the type parameters within that function.

```python
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).

```python
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.

It is an error to sample from an empty sequence, because returning `nothing()` makes it too easy to silently drop parts of compound strategies. If you need that behaviour, use `sampled_from(seq) if seq else nothing()`.

```python
hypothesis.strategies.sets(elements, *, min_size=0, 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.

```python
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:

```python
>>> 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:

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

Examples from this strategy shrink as per their base strategy.

```python
hypothesis.strategies.slices(size)
```

Generates slices that will select indices up to the supplied size

Generated slices will have start and stop indices that range from -size to size - 1 and will step in the appropriate direction. Slices should only produce an empty selection if the start and end are the same.

Examples from this strategy shrink toward 0 and smaller values

4.1. Core strategies
hypothesis.strategies.text(alphabet=characters(blacklist_categories=('Cs',)), *, min_size=0, max_size=None)

Generates strings with characters drawn from alphabet, which should be a collection of length one strings or a strategy generating such strings.

The default alphabet strategy can generate the full unicode range but excludes surrogate characters because they are invalid in the UTF-8 encoding. You can use characters() without arguments to find surrogate-related bugs such as bpo-34454.

min_size and max_size have the usual interpretations. Note that Python measures string length by counting codepoints: U+00C5 Å is a single character, while U+0041 U+030A A is two - the A, and a combining ring above.

Examples from this strategy shrink towards shorter strings, and with the characters in the text shrinking as per the alphabet strategy. This strategy does not normalize() examples, so generated strings may be in any or none of the 'normal forms'.

hypothesis.strategies.timedeltas(min_value=datetime.timedelta.min, max_value=datetime.timedelta.max)

A strategy for timedeltas between min_value and max_value.

Examples from this strategy shrink towards zero.

hypothesis.strategies.times(min_value=datetime.time.min, max_value=datetime.time.max, *, timezones=none())

A strategy for times between min_value and max_value.

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.

hypothesis.strategies.timezone_keys(*, allow_prefix=True)

A strategy for IANA timezone names.

As well as timezone names like "UTC", "Australia/Sydney", or "America/New_York", this strategy can generate:

- Aliases such as "Antarctica/McMurdo", which links to "Pacific/Auckland".
- Deprecated names such as "Antarctica/South_Pole", which also links to "Pacific/Auckland". Note that most but not all deprecated timezone names are also aliases.
- Timezone names with the "posix/" or "right/" prefixes, unless allow_prefix=False.

These strings are provided separately from Tzinfo objects - such as ZoneInfo instances from the timezones() strategy - to facilitate testing of timezone logic without needing workarounds to access non-canonical names.

Note: The zoneinfo module is new in Python 3.9, so you will need to install the backports.zoneinfo module on earlier versions, and the importlib_resources backport on Python 3.6.

pip install hypothesis[zoneinfo] will install these conditional dependencies if and only if they are needed.

On Windows, you may need to access IANA timezone data via the tzdata package. For non-IANA timezones, such as Windows-native names or GNU TZ strings, we recommend using sampled_from() with the dateutil package, e.g. dateutil.tz.tzwin.list().

hypothesis.strategies.timezone_keys(*, no_cache=False)

A strategy for zoneinfo.ZoneInfo objects.
If `no_cache=True`, the generated instances are constructed using `ZoneInfo.no_cache` instead of the usual constructor. This may change the semantics of your datetimes in surprising ways, so only use it if you know that you need to!

**Note:** The `zoneinfo` module is new in Python 3.9, so you will need to install the `backports.zoneinfo` module on earlier versions, and the `importlib_resources` backport on Python 3.6.

```shell
pip install hypothesis[zoneinfo]
```

will install these conditional dependencies if and only if they are needed.

```python
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.

```python
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.

### 4.2 Provisional strategies

This module contains various provisional APIs and strategies.

It is intended for internal use, to ease code reuse, and is not stable. Point releases may move or break the contents at any time!

Internet strategies should conform to [RFC 3986](https://tools.ietf.org/html/rfc3986) or the authoritative definitions it links to. If not, report the bug!

```python
hypothesis.provisional.domains(*, max_length=255, max_element_length=63)
```


```python
hypothesis.provisional.urls()
```


### 4.3 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.

### 4.4 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.

#### 4.4.1 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.**

```python
>>> 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()`, and if you find yourself indexing into a tuple within `.map()` it’s probably time to use that instead.

#### 4.4.2 Filtering

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

**e.g.**

```python
>>> 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:

```python
>>> integers().filter(lambda x: False).example()
Traceback (most recent call last):
  ... hypothesis.errors.Unsatisfiable: Could not find any valid examples in 20 tries
```

In general you should try to use `filter` only to avoid corner cases that you don’t want rather than attempting to cut out a large chunk of the search space.

A technique that often works well here is to use map to first transform the data and then use `filter` to remove things that didn’t work out. So for example if you wanted pairs of integers `(x, y)` such that `x < y` you could do the following:

```python
>>> tuples(integers(), integers()).map(sorted).filter(lambda x: x[0] < x[1]).example()
[-8543729478746591815, 3760495307320535691]
```
4.4.3 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:

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

4.4.4 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()` strategy 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:

```python
>>> 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())
```

(continues on next page)
Hypothesis Documentation, Release 6.25.0

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:

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

### 4.5 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 converting a function that returns one example into a function that returns a strategy that produces such examples - which you can pass to `@given`, modify with `.map` or `.filter`, and generally use like any other strategy.

It does this by giving you a special function `draw` as the first argument, which can be used just like the corresponding method of the `data()` strategy within a test. In fact, the implementation is almost the same - but defining a strategy with `@composite` makes code reuse easier, and usually improves the display of failing examples.

For example, the following gives you a list and an index into it:

```python
>>> @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.

```python
>>> list_and_index()
list_and_index()
```

(continues on next page)
>>> list_and_index(boolean())
list_and_index(elements=boolean())

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:

```python
@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.

Take care that your function can cope with adversarial draws, or explicitly rejects them using the `.filter()` method or `assume()` - our mutation and shrinking logic can do some strange things, and a naive implementation might lead to serious performance problems. For example:

```python
@composite
def reimplementing_sets_strategy(draw, elements=st.integers(), size=5):
    # The bad way: if Hypothesis keeps generating e.g. zero,
    # we'll keep looping for a very long time.
    result = set()
    while len(result) < size:
        result.add(draw(elements))
    # The good way: use a filter, so Hypothesis can tell what's valid!
    for _ in range(size):
        result.add(draw(elements.filter(lambda x: x not in result)))
    return result
```

If `@composite` is used to decorate a method or classmethod, the `draw` argument must come before `self` or `cls`. While we therefore recommend writing strategies as standalone functions and using the `register_type_strategy()` function to associate them with a class, methods are supported and the `@composite` decorator may be applied either before or after `@classmethod` or `@staticmethod`. See issue #2578 and pull request #2634 for more details.

### 4.6 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.

This is similar to `@composite`, but even more powerful as it allows you to mix test code with example generation. The downside of this power is that `data()` is incompatible with explicit `@example(...)s` - and the mixed code is often harder to debug when something goes wrong.

If you need values that are affected by previous draws but which `don’t` depend on the execution of your test, stick to the simpler `@composite`. 

---

4.6. Drawing interactively in tests 43
```python
@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.

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
```
Hypothesis has minimal dependencies, to maximise compatibility and make installing Hypothesis as easy as possible.

Our integrations with specific packages are therefore provided by extra modules that need their individual dependencies 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[cli]

```
$ hypothesis --help
Usage: hypothesis [OPTIONS] COMMAND [ARGS]...

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

Commands:
  codemod  'hypothesis codemod` refactors deprecated or inefficient code.
  fuzz      [hypofuzz] runs tests with an adaptive coverage-guided fuzzer.
  write     'hypothesis write` writes property-based tests for you!
```

This module requires the click package, and provides Hypothesis’ command-line interface, for e.g. ‘ghostwriting’ tests via the terminal. It’s also where HypoFuzz adds the hypothesis fuzz command (learn more about that here).
5.2 hypothesis[codemods]

This module provides codemods based on the LibCST library, which can both detect and automatically fix issues with code that uses Hypothesis, including upgrading from deprecated features to our recommended style.

You can run the codemods via our CLI:

```
$ hypothesis codemod --help
Usage: hypothesis codemod [OPTIONS] PATH...

  `hypothesis codemod` refactors deprecated or inefficient code.

  It adapts `python -m libcst.tool`, removing many features and config options which are rarely relevant for this purpose. If you need more control, we encourage you to use the libcst CLI directly; if not this one is easier.

  PATH is the file(s) or directories of files to format in place, or "-" to read from stdin and write to stdout.

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

Alternatively you can use `python -m libcst.tool`, which offers more control at the cost of additional configuration (adding `hypothesis.extra` to the modules list in `.libcst.codemod.yaml`) and some issues on Windows.

```
alternativetool = hypothesis.extra.codemods.refactor(code)

This is a convenience function for testing internal code that uses dpcontracts, to automatically filter out arguments that would be rejected by the public interface before triggering a contract error.
```

5.3 hypothesis[dpcontracts]

This module provides tools for working with the dpcontracts library, because combining contracts and property-based testing works really well.

It requires dpcontracts >= 0.4.

---

**Tip:** For new projects, we recommend using either deal or icontact and icontact-hypothesis over dpcontracts. They're generally more powerful tools for design-by-contract programming, and have substantially nicer Hypothesis integration too!

```
alternativetool = hypothesis.extra.dpcontracts.fulfill(contract_func)

This can be used as builds(fulfill(func), ...) or in the body of the test e.g. assert fulfill(func)(*args).
```
5.4 hypothesis[lark]

This extra can be used to generate strings matching any context-free grammar, using the Lark parser library.

It currently only supports Lark’s native EBNF syntax, but we plan to extend this to support other common syntaxes such as ANTLR and RFC 5234 ABNF. Lark already supports loading grammars from nearley.js, so you may not have to write your own at all.

Note that as Lark is at version 0.x, this module may break API compatibility in minor releases if supporting the latest version of Lark would otherwise be infeasible. We may also be quite aggressive in bumping the minimum version of Lark, unless someone volunteers to either fund or do the maintenance.

```
from_lark(grammar, *, start=None, explicit=None)
```

A strategy for strings accepted by the given context-free grammar.

grammar must be a Lark object, which wraps an EBNF specification. The Lark EBNF grammar reference can be found here.

from_lark will automatically generate strings matching the nonterminal start symbol in the grammar, which was supplied as an argument to the Lark class. To generate strings matching a different symbol, including terminals, you can override this by passing the start argument to from_lark. Note that Lark may remove unreachable productions when the grammar is compiled, so you should probably pass the same value for start to both.

Currently from_lark does not support grammars that need custom lexing. Any lexers will be ignored, and any undefined terminals from the use of %declare will result in generation errors. To define strategies for such terminals, pass a dictionary mapping their name to a corresponding strategy as the explicit argument.

The hypothesis project includes a strategy for Python source, based on a grammar and careful post-processing. Example grammars, which may provide a useful starting point for your tests, can be found in the Lark repository and in this third-party collection.

5.5 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.

```
timezones()
```

Any timezone in the Olsen database, as a pytz tzinfo object.

This strategy minimises to UTC, or the timezone with the smallest offset from UTC as of 2000-01-01, and is designed for use with hypothesis.strategies.datetimes().

5.6 hypothesis[dateutil]

This module provides dateutil timezones.

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

```
timezones()
```

Any timezone from dateutil.

This strategy minimises to UTC, or the timezone with the smallest offset from UTC as of 2000-01-01, and is designed for use with datetimes().
Note that the timezones generated by the strategy may vary depending on the configuration of your machine. See the dateutil documentation for more information.
Writing tests with Hypothesis frees you from the tedium of deciding on and writing out specific inputs to test. Now, the `hypothesis.extra.ghostwriter` module can write your test functions for you too!

The idea is to provide an easy way to start property-based testing, and a seamless transition to more complex test code - because ghostwritten tests are source code that you could have written for yourself.

So just pick a function you'd like tested, and feed it to one of the functions below. They follow imports, use but do not require type annotations, and generally do their best to write you a useful test. You can also use our command-line interface:

```
$ hypothesis write --help
Usage: hypothesis write [OPTIONS] FUNC...

'hypothesis write' writes property-based tests for you!

Type annotations are helpful but not required for our advanced introspection and templating logic. Try running the examples below to see how it works:

hyposthesis write gzip
hyposthesis write numpy.matmul
hyposthesis write re.compile --except re.error
hyposthesis write --equivalent ast.literal_eval eval
hyposthesis write --roundtrip json.dumps json.loads
hyposthesis write --style=unittest --idempotent sorted
hyposthesis write --binary-op operator.add
```

Options:
- `--roundtrip` start by testing write/read or encode/decode!
- `--equivalent` very useful when optimising or refactoring code
- `--idempotent`
- `--binary-op`
- `--style [pytest|unittest]` pytest-style function, or unittest-style method?
- `-e, --except OBJ_NAME` dotted name of exception(s) to ignore
- `-h, --help` Show this message and exit.

**Note:** The ghostwriter requires black, but the generated code only requires Hypothesis itself.

**Note:** Legal questions? While the ghostwriter fragments and logic is under the MPL-2.0 license like the rest of Hypothesis, the output from the ghostwriter is made available under the Creative Commons Zero (CC0) public domain
dedication, so you can use it without any restrictions.

hypothesis.extra.ghostwriter.magic(*modules_or_functions, except_=(), style='pytest')

Guess which ghostwriters to use, for a module or collection of functions.

As for all ghostwriters, the except_ argument should be an Exception or tuple of exceptions, and style may be either "pytest" to write test functions or "unittest" to write test methods and TestCase.

After finding the public functions attached to any modules, the magic ghostwriter looks for pairs of functions to pass to roundtrip(), then checks for binary_operation() and ufunc() functions, and any others are passed to fuzz().

For example, try `hypothesis write gzip` on the command line!

hypothesis.extra.ghostwriter.fuzz(func, *, except_=(), style='pytest')

Write source code for a property-based test of func.

The resulting test checks that valid input only leads to expected exceptions. For example:

```python
from re import compile, error
from hypothesis.extra import ghostwriter
ghostwriter.fuzz(compile, except_=error)
```

Gives:

```python
# This test code was written by the `hypothesis.extra.ghostwriter` module
# and is provided under the Creative Commons Zero public domain dedication.
import re

from hypothesis import given, reject, strategies as st

# TODO: replace st.nothing() with an appropriate strategy

given(pattern=st.nothing(), flags=st.just(0))
def test_fuzz_compile(pattern, flags):
    try:
        re.compile(pattern=pattern, flags=flags)
    except re.error:
        reject()
```

Note that it includes all the required imports. Because the pattern parameter doesn’t have annotations or a default argument, you’ll need to specify a strategy - for example `text()` or `binary()`. After that, you have a test!

hypothesis.extra.ghostwriter.idempotent(func, *, except_=(), style='pytest')

Write source code for a property-based test of func.

The resulting test checks that if you call func on it’s own output, the result does not change. For example:

```python
from typing import Sequence
from hypothesis.extra import ghostwriter
```

(continues on next page)
```python
def timsort(seq: Sequence[int]) -> Sequence[int]:
    return sorted(seq)
```

ghostwriter.idempotent(timsort)

Gives:

```python
# This test code was written by the `hypothesis.extra.ghostwriter` module
# and is provided under the Creative Commons Zero public domain dedication.

from hypothesis import given, strategies as st

given(seq=st.one_of(st.binary(), st.binary().map(bytearray), st.lists(st.integers())))

def test_idempotent_timsort(seq):
    result = timsort(seq=seq)
    repeat = timsort(seq=result)
    assert result == repeat, (result, repeat)
```

hypothesis.extra.ghostwriter.roundtrip(*funcs, except_=(), style='pytest')

Write source code for a property-based test of `funcs`.

The resulting test checks that if you call the first function, pass the result to the second (and so on), the final result is equal to the first input argument.

This is a very powerful property to test, especially when the config options are varied along with the object to round-trip. For example, try ghostwriting a test for `json.dumps()` - would you have thought of all that?

hypothesis.write --roundtrip json.dumps json.loads

hypothesis.extra.ghostwriter.equivalent(*funcs, except_=(), style='pytest')

Write source code for a property-based test of `funcs`.

The resulting test checks that calling each of the functions gives the same result. This can be used as a classic 'oracle', such as testing a fast sorting algorithm against the `sorted()` built-in, or for differential testing where none of the compared functions are fully trusted but any difference indicates a bug (e.g. running a function on different numbers of threads, or simply multiple times).

The functions should have reasonably similar signatures, as only the common parameters will be passed the same arguments - any other parameters will be allowed to vary.

hypothesis.extra.ghostwriter.binary_operation(func, *, associative=True, commutative=True, identity=infer, distributes_over=None, except_=(), style='pytest')

Write property tests for the binary operation `func`.

While binary operations are not particularly common, they have such nice properties to test that it seems a shame not to demonstrate them with a ghostwriter. For an operator `f`, test that:

- if associative, `f(a, f(b, c)) == f(f(a, b), c)`
- if commutative, `f(a, b) == f(b, a)`
- if identity is not None, `f(a, identity) == a`
- if distributes_over is +, `f(a, b) + f(a, c) == f(a, b+c)`
For example:

```python
ghostwriter.binary_operation(  
    operator.mul,  
    identity=1,  
    distributes_over=operator.add,  
    style="unittest",  
)
```

`hypothesis.extra.ghostwriter.ufunc(func, *, except_=(), style='pytest')`

Write a property-based test for the array ufunc `func`.

The resulting test checks that your ufunc or gufunc has the expected broadcasting and dtype casting behaviour. You will probably want to add extra assertions, but as with the other ghostwriters this gives you a great place to start.

```python
hypothesis.write numpy.matmul
```

Chapter 6.  Ghostwriting tests for you
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.

```python
class hypothesis.extra.django.TestCase
```

Using it is quite straightforward: All you need to do is subclass `hypothesis.extra.django.TestCase` or `hypothesis.extra.django.TransactionTestCase` or `LiveServerTestCase` or `StaticLiveServerTestCase` 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.

```python
class hypothesis.extra.django.TransactionTestCase
```

```python
class hypothesis.extra.django.LiveServerTestCase
```

```python
class hypothesis.extra.django.StaticLiveServerTestCase
```

We recommend avoiding `TransactionTestCase` unless you really have to run each test case in a database transaction. 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:

```python
hypothesis.extra.django.from_model(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)`. In order to support models with fields named “model”, this is a positional-only parameter.

Hypothesis can often infer a strategy based the field type and validators, and will attempt to do so for any required fields. No strategy will be inferred for an `AutoField`, nullable field, foreign key, or field for which a keyword argument is passed to `from_model()`. For example, a Shop type with a foreign key to Company could be generated with:

```python
shop_strategy = from_model(Shop, company=from_model(Company))
```
Like for `builds()`, you can pass `infer` as a keyword argument to infer a strategy for a field which has a default value instead of using the default.

For example, using the trivial Django project we have for testing:

```python
>>> from hypothesis.extra.django import from_model
>>> from toystore.models import Customer
>>> c = from_model(Customer).example()
>>> c
<Customer: Customer object>
>>> c.email
'jaime.urbina@gmail.com'
>>> c.name
'\U00109d3d\U000e07be\U000165f8\U0003fabf\U000c12cd\U000f1910\U00059f12\U000519b0\n\U0003fabf\U000f1910\U000423fb\U000423fb\U00059f12\U000e07be\U000c12cd\U000e07be\n\U000519b0\U000165f8\U0003fabf\U0007bc31'
>>> c.age
-873375803
```

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

Obviously the customer’s age is implausible, 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.
```

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

Hypothesis can often infer a strategy based the field type and validators, and will attempt to do so for any required fields. No strategy will be inferred for a disabled field or field for which a keyword argument is passed to `from_form()`.

This function uses the fields of an unbound `form` instance to determine field strategies, any keyword arguments needed to instantiate the unbound `form` instance can be passed into `from_form()` as a dict with the keyword `form_kwargs`. E.g.:

```python
shop_strategy = from_form(Shop, form_kwargs={"company_id": 5})
```
Like for `builds()`, you can pass `infer` as a keyword argument to infer a strategy for a field which has a default value instead of using the default.

## 7.1 Tips and tricks

### 7.1.1 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:

```python
>>> from toystore.models import CustomishField, Customish
>>> from_model(Customish).example()
```

```
hypothesis.errors.InvalidArgument: Missing arguments for mandatory field
customish for model Customish
```

```python
>>> from hypothesis.extra.django import register_field_strategy
>>> from hypothesis.strategies import just

>>> register_field_strategy(CustomishField, just("hi"))

>>> x = from_model(Customish).example()

>>> x.customish
"hi"
```

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

`hypothesis.extra.django.register_field_strategy(field_type, strategy)`

Add an entry to the global field-to-strategy lookup used by `from_field()`.

- `field_type` must be a subtype of `django.db.models.Field` or `django.forms.Field`, which must not already be registered.
- `strategy` must be a `SearchStrategy`.

`hypothesis.extra.django.from_field(field)`

Return a strategy for values that fit the given field.

This function is used by `from_form()` and `from_model()` for any fields that require a value, or for which you passed `hypothesis.infer`.

It’s pretty similar to the core `from_type()` function, with a subtle but important difference: `from_field` takes a `Field instance`, rather than a `Field subtype`, so that it has access to instance attributes such as string length and validators.

### 7.1.2 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:

```python
from hypothesis.strategies import just, lists

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

company_with_shops_strategy = from_model(Company).flatmap(generate_with_shops)
```
Let's 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 `from_model(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:

```python
from_model(Company).flatmap(lambda c: lists(from_model(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.

### 7.1.3 Generating primary key values

If your model includes a custom primary key that you want to generate using a strategy (rather than a default auto-increment primary key) then Hypothesis has to deal with the possibility of a duplicate primary key.

If a model strategy generates a value for the primary key field, Hypothesis will create the model instance with `update_or_create()`, overwriting any existing instance in the database for this test case with the same primary key.

### 7.1.4 On the subject of `MultiValueField`

Django forms feature the `MultiValueField` which allows for several fields to be combined under a single named field, the default example of this is the `SplitDateTimeField`

```python
class CustomerForm(forms.Form):
    name = forms.CharField()
    birth_date_time = forms.SplitDateTimeField()
```

`from_form` supports `MultiValueField` subclasses directly, however if you want to define your own strategy be forewarned that Django binds data for a `MultiValueField` in a peculiar way. Specifically each sub-field is expected to have its own entry in data addressed by the field name (e.g. `birth_date_time`) and the index of the sub-field within the `MultiValueField`, so form data for the example above might look like this:

```json
{
    "name": "Samuel John",
    "birth_date_time_0": "2018-05-19",  # the date, as the first sub-field
    "birth_date_time_1": "15:18:00",    # the time, as the second sub-field
}
```

Thus, if you want to define your own strategies for such a field you must address your sub-fields appropriately:

```python
from_form(CustomerForm, birth_date_time_0=just("2018-05-19"))
```
8.1 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.from_dtype(dtype, *, alphabet=None, min_size=0, max_size=None, min_value=None, max_value=None, allow_nan=None, allow_infinity=None, exclude_min=None, exclude_max=None)

Creates a strategy which can generate any value of the given dtype.

Compatible **kwargs are passed to the inferred strategy function for integers, floats, and strings. This allows you to customise the min and max values, control the length or contents of strings, or exclude non-finite numbers. This is particularly useful when kwargs are passed through from arrays() which allow a variety of numeric dtypes, as it seamlessly handles the width or representable bounds for you. See issue #2552 for more detail.

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

Returns a strategy for generating numpy.ndarray.

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

• shape may be an integer >= 0, a tuple 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 a mapping, it will be passed as **kwargs to from_dtype()

• 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 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:

```python
>>> 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,  0.19778793])
```

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 is the smallest length that the generated shape can possess.

* max_dims is the largest length that the generated shape can possess, defaulting to min_dims + 2.

* min_side is the smallest size that a dimension can possess.

* max_side is the largest size that a dimension can possess, defaulting to min_side + 5.

**hypothesis.extra.numpy.scalar_dtypes()

Return a strategy that can return any non-flexible scalar dtype.

**hypothesis.extra.numpy.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 and sizes are treated as for *unsigned_integer_dtypes()*. 

**hypothesis.extra.numpy.floating_dtypes(**

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.

```python
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 halves (a 16-bit real part) are not supported by numpy and will not be generated by this strategy.

```python
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.

```python
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.

```python
hypothesis.extra.numpy.byte_string_dtypes(*, endianness='?', min_len=1, max_len=16)
```

Return a strategy for generating bytestring dtypes, of various lengths and byteorder.

While Hypothesis’ string strategies can generate empty strings, string dtypes with length 0 indicate that size is still to be determined, so the minimum length for string dtypes is 1.

```python
hypothesis.extra.numpy.unicode_string_dtypes(*, endianness='?', min_len=1, max_len=16)
```

Return a strategy for generating unicode string dtypes, of various lengths and byteorder.

While Hypothesis’ string strategies can generate empty strings, string dtypes with length 0 indicate that size is still to be determined, so the minimum length for string dtypes is 1.

```python
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.

```python
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.

```python
hypothesis.extra.numpy.valid_tuple_axes(ndim, *, min_size=0, max_size=None)
```

Return a strategy for generating permissible tuple-values for the axis argument for a numpy sequential function (e.g. `numpy.sum()`), given an array of the specified dimensionality.

All tuples will have a length >= min_size and <= max_size. The default value for max_size is ndim.

Examples from this strategy shrink towards an empty tuple, which render most sequential functions as no-ops.

The following are some examples drawn from this strategy.

```python
>>> [valid_tuple_axes(3).example() for i in range(4)]
[(-3, 1), (0, 1, -1), (0, 2), (0, -2, 2)]
```

valid_tuple_axes can be joined with other strategies to generate any type of valid axis object, i.e. integers, tuples, and None:

```python
any_axis_strategy = none() | integers(-ndim, ndim - 1) | valid_tuple_axes(ndim)
```
hypothesis.extra.numpy.broadcastable_shapes(shape, *, min_dims=0, max_dims=None, min_side=1, max_side=None)

Return a strategy for shapes that are broadcast-compatible with the provided shape.

Examples from this strategy shrink towards a shape with length min_dims. The size of an aligned dimension shrinks towards size 1. The size of an unaligned dimension shrink towards min_side.

- **shape** is a tuple of integers.
- **min_dims** is the smallest length that the generated shape can possess.
- **max_dims** is the largest length that the generated shape can possess, defaulting to max(len(shape), min_dims) + 2.
- **min_side** is the smallest size that an unaligned dimension can possess.
- **max_side** is the largest size that an unaligned dimension can possess.

The following are some examples drawn from this strategy.

```python
>>> [broadcastable_shapes(shape=(2, 3)).example() for i in range(5)]
[(1, 3), (), (2, 3), (2, 1), (4, 1, 3), (3, )]
```

hypothesis.extra.numpy.mutually_broadcastable_shapes(*, num_shapes=not_set, signature=not_set, base_shape=(), min_dims=0, max_dims=None, min_side=1, max_side=None)

Return a strategy for a specified number of shapes N that are mutually-broadcastable with one another and with the provided base shape.

- **num_shapes** is the number of mutually broadcast-compatible shapes to generate.
- **base_shape** is the shape against which all generated shapes can broadcast. The default shape is empty, which corresponds to a scalar and thus does not constrain broadcasting at all.
- **shape** is a tuple of integers.
- **min_dims** is the smallest length that the generated shape can possess.
- **max_dims** is the largest length that the generated shape can possess, defaulting to max(len(shape), min_dims) + 2.
- **min_side** is the smallest size that an unaligned dimension can possess.
- **max_side** is the largest size that an unaligned dimension can possess, defaulting to 2 plus the size of the largest aligned dimension.

The strategy will generate a `typing.NamedTuple` containing:

- **input_shapes** as a tuple of the N generated shapes.
- **result_shape** as the resulting shape produced by broadcasting the N shapes with the base shape.

The following are some examples drawn from this strategy.

```python
>>> # Draw three shapes where each shape is broadcast-compatible with (2, 3)
... strat = mutually_broadcastable_shapes(num_shapes=3, base_shape=(2, 3))
>>> for _ in range(5):
...     print(strat.example())
BroadcastableShapes(input_shapes=((4, 1, 3), (4, 2, 3), ()), result_shape=(4, 2, 3))
BroadcastableShapes(input_shapes=((3,), (1, 3), (2, 3)), result_shape=(2, 3))
BroadcastableShapes(input_shapes=((,), (), ()), result_shape=(2, 3))
```
Use with Generalised Universal Function signatures

A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features. A generalised ufunc operates on sub-arrays rather than elements, based on the “signature” of the function. Compare e.g. `numpy.add()` (ufunc) to `numpy.matmul()` (gufunc).

To generate shapes for a gufunc, you can pass the signature argument instead of num_shapes. This must be a gufunc signature string; which you can write by hand or access as e.g. `np.matmul.signature` on generalised ufuncs.

In this case, the side arguments are applied to the ‘core dimensions’ as well, ignoring any frozen dimensions. base_shape and the dims arguments are applied to the ‘loop dimensions’, and if necessary, the dimensionality of each shape is silently capped to respect the 32-dimension limit.

The generated result_shape is the real result shape of applying the gufunc to arrays of the generated input_shapes, even where this is different to broadcasting the loop dimensions.

gufunc-compatible shapes shrink their loop dimensions as above, towards omitting optional core dimensions, and smaller-size core dimensions.

```python
>>> # np.matmul.signature == "(m?,n),(n,p?)->(m?,p?)"
>>> for _ in range(3):
...    mutually_broadcastable_shapes(signature=np.matmul.signature).example()
BroadcastableShapes(input_shapes=((2,), (2,)), result_shape=())
BroadcastableShapes(input_shapes=((3, 4, 2), (1, 2)), result_shape=(3, 4))
BroadcastableShapes(input_shapes=((4, 2), (1, 2, 3)), result_shape=(4, 3))
```

hypothesis.extra.numpy.basic_indices(shape, *, min_dims=0, max_dims=None, allow_newaxis=False, allow_ellipsis=True)

Return a strategy for basic indexes of arrays with the specified shape, which may include dimensions of size zero.

It generates tuples containing some mix of integers, slice objects, ... (an Ellipsis), and None. When a length-one tuple would be generated, this strategy may instead return the element which will index the first axis, e.g. 5 instead of (5,).

- shape is the shape of the array that will be indexed, as a tuple of positive integers. This must be at least two-dimensional for a tuple to be a valid index; for one-dimensional arrays use slices() instead.
- min_dims is the minimum dimensionality of the resulting array from use of the generated index. When min_dims == 0, scalars and zero-dimensional arrays are both allowed.
- max_dims is the the maximum dimensionality of the resulting array, defaulting to len(shape) if not allow_newaxis else max(len(shape), min_dims) + 2.
- allow_newaxis specifies whether None is allowed in the index.
- allow_ellipsis specifies whether ... is allowed in the index.

hypothesis.extra.numpy.integer_array_indices(shape, *, result_shape=array_shapes(), dtype='int')

Return a search strategy for tuples of integer-arrays that, when used to index into an array of shape shape, given an array whose shape was drawn from result_shape.

Examples from this strategy shrink towards the tuple of index-arrays:
len(shape) * (np.zeros(drawn_result_shape, dtype), )

- shape a tuple of integers that indicates the shape of the array, whose indices are being generated.
- result_shape a strategy for generating tuples of integers, which describe the shape of the resulting index arrays. The default is array_shapes(). The shape drawn from this strategy determines the shape of the array that will be produced when the corresponding example from integer_array_indices is used as an index.
- dtype the integer data type of the generated index-arrays. Negative integer indices can be generated if a signed integer type is specified.

Recall that an array can be indexed using a tuple of integer-arrays to access its members in an arbitrary order, producing an array with an arbitrary shape. For example:

```py
>>> from numpy import array
>>> x = array([-0, -1, -2, -3, -4])
>>> ind = (array([[4, 0], [0, 1]],),)  # a tuple containing a 2D integer-array
>>> x[ind]  # the resulting array is commensurate with the indexing array(s)
array([[-4, 0],
       [0, -1]])
```

Note that this strategy does not accommodate all variations of so-called ‘advanced indexing’, as prescribed by NumPy’s nomenclature. Combinations of basic and advanced indexes are too complex to usefully define in a standard strategy; we leave application-specific strategies to the user. Advanced-boolean indexing can be defined as arrays(shape=..., dtype=bool), and is similarly left to the user.

### 8.2 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, we 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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))
  0  -3.402823e+38  9.007199e+15
  1  -1.562796e-298  5.000000e-01
```

You can also combine the two:

```python
>>> from hypothesis.extra.pandas import columns, data_frames
>>> import hypothesis.strategies as st
>>> data_frames(...
    ...   columns=[column('value', dtype=st.floats(allow_nan=False)),
    ...             column('other_value', dtype=st.floats(allow_nan=False))])
  value    other_value
0  -3.402823e+38   9.007199e+15
1  -1.562796e-298  5.000000e-01
```
columns=columns(["lo", "hi"], dtype=float),
rows=st.tuples(st.floats(allow_nan=False),
            st.floats(allow_nan=False)).map(sorted)
).example()

<table>
<thead>
<tr>
<th>lo</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9.314723e-49</td>
</tr>
<tr>
<td>1</td>
<td>-9.999900e-01</td>
</tr>
<tr>
<td>2</td>
<td>-2.152861e+134</td>
</tr>
<tr>
<td>3</td>
<td>-1.069317e-73</td>
</tr>
</tbody>
</table>

(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.

### 8.2.1 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 every Pandas minor version from 0.25 through to 1.1.

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.

### 8.3 Array API

**Tip:** The Array API standard is not yet finalised, so this module will make breaking changes if necessary to support newer versions of the standard.

Hypothesis offers strategies for Array API adopting libraries in the hypothesis.extra.array_api package. See issue #3037 for more details. If you want to test with CuPy, Dask, JAX, MXNet, PyTorch, TensorFlow, or Xarray - or just numpy.array_api - this is the extension for you!

hypothesis.extra.array_api.make_strategies_namespace(xp)

Creates a strategies namespace for the given array module.

- **xp** is the Array API library to automatically pass to the namespaced methods.

A types.SimpleNamespace is returned which contains all the strategy methods in this module but without requiring the xp argument. Creating and using a strategies namespace for NumPy’s Array API implementation would go like this:
>>> from numpy import array_api as xp
>>> xps = make_strategies_namespace(xp)
>>> x = xps.arrays(xp.int8, (2, 3)).example()
>>> x
Array([[-8, 6, 3],
       [-6, 4, 6]], dtype=int8)
>>> x.__array_namespace__() is xp
True

The resulting namespace contains all our familiar strategies like `arrays()` and `from_dtype()`, but based on the Array API standard semantics and returning objects from the `xp` module:

```python
def xps.from_dtype(dtype, *, min_value=None, max_value=None, allow_nan=None, allow_infinity=None, exclude_min=None, exclude_max=None):
    """Return a strategy for any value of the given dtype. Values generated are of the Python scalar which is promotable to dtype, where the values do not exceed its bounds."
    """  
    dtype may be a dtype object or the string name of a valid dtype.

    Compatible **kwargs are passed to the inferred strategy function for integers and floats. This allows you to customise the min and max values, and exclude non-finite numbers. This is particularly useful when kwargs are passed through from `arrays()`, as it seamlessly handles the width or other representable bounds for you.

```python

def xps.arrays(dtype, shape, *, elements=None, fill=None, unique=False):
    """Returns a strategy for arrays."
    """
    dtype may be a valid dtype object or name, or a strategy that generates such values.
    
    shape may be an integer >= 0, a tuple of such integers, or a strategy that generates such values.
    
    elements is a strategy for values to put in the array. If None then a suitable value will be inferred based on the dtype, which may give any legal value (including e.g. NaN for floats). If a mapping, it will be passed as **kwargs to `from_dtype()` when inferring based on the dtype.
    
    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 `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; if fill is also set, the only valid values for fill to return are NaN values.

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

```python
>>> from numpy import array_api as xp
>>> xps.arrays(xp, xp.int8, (2, 3)).example()
Array([[-8, 6, 3],
       [-6, 4, 6]], dtype=int8)

Specifying element boundaries by a dict of the kwargs to pass to `from_dtype()` will ensure dtype bounds will be respected.

```python

```python
>>> xps.arrays(xp, xp.int8, 3, elements={"min_value": 10}).example()
Array([125, 13, 79], dtype=int8)

Refer to What you can generate and how for passing your own elements strategy.
Array values are generated in two parts:

1. A single value is drawn from the fill strategy and is used to create a filled array.
2. Some subset of the coordinates of the array are populated with a value drawn from the elements strategy (or its inferred form).

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

By default `arrays` will attempt to infer the correct fill behaviour: if `unique` is also `True`, no filling will occur. 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.

```python
>>> xps.arrays(xp, xp.float32, 3, elements=floats(0, 1, width=32)).example()
Array([ 0.88974794, 0.77387938, 0.19778794], dtype=float32)
```

### xps.array_shapes(*, min_dims=1, max_dims=None, min_side=1, max_side=None)

Return a strategy for array shapes (tuples of int >= 1).

- `min_dims` is the smallest length that the generated shape can possess.
- `max_dims` is the largest length that the generated shape can possess, defaulting to `min_dims + 2`.
- `min_side` is the smallest size that a dimension can possess.
- `max_side` is the largest size that a dimension can possess, defaulting to `min_side + 5`.

### xps.scalar_dtypes()

Return a strategy for all valid dtype objects.

### xps.boolean_dtypes()

Return a strategy for just the boolean dtype object.

### xps.numeric_dtypes()

Return a strategy for all numeric dtype objects.

### xps.integer_dtypes(*, sizes=(8, 16, 32, 64))

Return a strategy for signed integer dtype objects.

- `sizes` contains the signed integer sizes in bits, defaulting to (8, 16, 32, 64) which covers all valid sizes.

### xps.unsigned_integer_dtypes(*, sizes=(8, 16, 32, 64))

Return a strategy for unsigned integer dtype objects.

- `sizes` contains the unsigned integer sizes in bits, defaulting to (8, 16, 32, 64) which covers all valid sizes.

### xps.floating_dtypes(*, sizes=(32, 64))

Return a strategy for floating-point dtype objects.

- `sizes` contains the floating-point sizes in bits, defaulting to (32, 64) which covers all valid sizes.

### xps.valid_tuple_axes(ndim, *, min_size=0, max_size=None)

Return a strategy for permissible tuple-values for the `axis` argument in Array API sequential methods e.g. `sum`, given the specified dimensionality.

- All tuples will have a length >= `min_size` and <= `max_size`. The default value for `max_size` is `ndim`.

Examples from this strategy shrink towards an empty tuple, which render most sequential functions as no-ops.
The following are some examples drawn from this strategy.

```python
>>> [valid_tuple_axes(3).example() for i in range(4)]
[(-3, 1), (0, 1, -1), (0, 2), (0, -2, 2)]
```

valid_tuple_axes can be joined with other strategies to generate any type of valid axis object, i.e. integers, tuples, and None:

```python
goingal = none() | integers(-ndim, ndim - 1) | valid_tuple_axes(ndim)
```

xps.broadcastable_shapes(shape, *, min_dims=0, max_dims=None, min_side=1, max_side=None)

Return a strategy for shapes that are broadcast-compatible with the provided shape.

Examples from this strategy shrink towards a shape with length min_dims. The size of an aligned dimension shrinks towards size 1. The size of an unaligned dimension shrink towards min_side.

- shape is a tuple of integers.
- min_dims is the smallest length that the generated shape can possess.
- max_dims is the largest length that the generated shape can possess, defaulting to max(len(shape), min_dims) + 2.
- min_side is the smallest size that an unaligned dimension can possess.
- max_side is the largest size that an unaligned dimension can possess, defaulting to 2 plus the size of the largest aligned dimension.

The following are some examples drawn from this strategy.

```python
>>> [broadcastable_shapes(shape=(2, 3)).example() for i in range(5)]
[(1, 3), (), (2, 3), (2, 1), (4, 1, 3), (3, )]
```

xps.mutually_broadcastable_shapes(num_shapes, *, base_shape=(), min_dims=0, max_dims=None, min_side=1, max_side=None)

Return a strategy for a specified number of shapes N that are mutually-broadcastable with one another and with the provided base shape.

- num_shapes is the number of mutually broadcast-compatible shapes to generate.
- base_shape is the shape against which all generated shapes can broadcast. The default shape is empty, which corresponds to a scalar and thus does not constrain broadcasting at all.
- shape is a tuple of integers.
- min_dims is the smallest length that the generated shape can possess.
- max_dims is the largest length that the generated shape can possess, defaulting to max(len(shape), min_dims) + 2.
- min_side is the smallest size that an unaligned dimension can possess.
- max_side is the largest size that an unaligned dimension can possess, defaulting to 2 plus the size of the largest aligned dimension.

The strategy will generate a typing.NamedTuple containing:

- input_shapes as a tuple of the N generated shapes.
- result_shape as the resulting shape produced by broadcasting the N shapes with the base shape.

The following are some examples drawn from this strategy.
```python
>>> # Draw three shapes where each shape is broadcast-compatible with (2, 3)
... strat = mutually_broadcastable_shapes(num_shapes=3, base_shape=(2, 3))
>>> for _ in range(5):
...     print(strat.example())
BroadcastableShapes(input_shapes=((4, 1, 3), (4, 2, 3), ()), result_shape=(4, 2, 3))
BroadcastableShapes(input_shapes=((3,), (1, 3), (2, 3)), result_shape=(2, 3))
BroadcastableShapes(input_shapes=((1, 2, 3), (3,), ()), result_shape=(1, 2, 3))
```

xps.indices(shape, *, min_dims=0, max_dims=None, allow_ellipsis=True)

Return a strategy for valid indices of arrays with the specified shape, which may include dimensions of size zero.

It generates tuples containing some mix of integers, slice objects, and ... (an Ellipsis). When a length-one tuple would be generated, this strategy may instead return the element which will index the first axis, e.g. 5 instead of (5,).

- **shape** is the shape of the array that will be indexed, as a tuple of integers >= 0. This must be at least two-dimensional for a tuple to be a valid index; for one-dimensional arrays use slices() instead.
- **min_dims** is the minimum dimensionality of the resulting array from use of the generated index.
- **max_dims** is the the maximum dimensionality of the resulting array, defaulting to len(shape).
- **allow_ellipsis** specifies whether ... is allowed in the index.
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. Using a value of `HealthCheck.all()` will disable all health checks.

```python
class hypothesis.HealthCheck(value)

Arguments for suppress_health_check.

Each member of this enum is a type of health check to suppress.

data_too_large = 1
Checks if too many examples are aborted for being too large.

This is measured by the number of random choices that Hypothesis makes in order to generate something, not the size of the generated object. For example, choosing a 100MB object from a predefined list would take only a few bits, while generating 10KB of JSON from scratch might trigger this health check.

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.

return_value = 5
Checks if your tests return a non-None value (which will be ignored and is unlikely to do what you want).

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 defined by `unittest.TestCase` (i.e. not a test).```
function_scoped_fixture = 9

Check if @given has been applied to a test with a pytest function-scoped fixture. Function-scoped fixtures run once for the whole function, not once per example, and this is usually not what you want.

Because of this limitation, tests that need to need to set up or reset state for every example need to do so manually within the test itself, typically using an appropriate context manager.

Suppress this health check only in the rare case that you are using a function-scoped fixture that does not need to be reset between individual examples, but for some reason you cannot use a wider fixture scope (e.g. session scope, module scope, class scope).

This check requires the Hypothesis pytest plugin, which is enabled by default when running Hypothesis inside pytest.

9.1 Deprecations

We also use a range of custom exception and warning types, so you can see exactly where an error came from - or turn only our warnings into errors.

class hypothesis.errors.HypothesisDeprecationWarning
    A deprecation warning issued by Hypothesis.
    Actually inherits from FutureWarning, because DeprecationWarning is hidden by the default warnings filter.
    You can configure the Python warnings to handle these warnings differently to others, either turning them into errors or suppressing them entirely. Obviously we would prefer the former!

Deprecated features will be continue to emit warnings for at least six months, and then be removed in the following major release. Note however that not all warnings are subject to this grace period; sometimes we strengthen validation by adding a warning and these may become errors immediately at a major release.
CHAPTER
TEN

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.

10.1 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 exercise less-used parts of your code, so the database may update even when no failing examples were found.

10.2 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 can never be the case that e.g. changing the strategy that generates an argument gives you data from the old strategy.

10.3 ExampleDatabase implementations

Hypothesis’ default `database` setting creates a `DirectoryBasedExampleDatabase` in your current working directory, under `.hypothesis/examples`. If this location is unusable, e.g. because you do not have read or write permissions, Hypothesis will emit a warning and fall back to an `InMemoryExampleDatabase`.

Hypothesis provides the following `ExampleDatabase` implementations:

```python
class hypothesis.database.InMemoryExampleDatabase
    A non-persistent example database, implemented in terms of a dict of sets.

This can be useful if you call a test function several times in a single session, or for testing other database implementations, but because it does not persist between runs we do not recommend it for general use.

class hypothesis.database.DirectoryBasedExampleDatabase(path)
    Use a directory to store Hypothesis examples as files.
```
Each test corresponds to a directory, and each example to a file within that directory. While the contents are fairly opaque, a `DirectoryBasedExampleDatabase` can be shared by checking the directory into version control, for example with the following `.gitignore`:

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

Note however that this only makes sense if you also pin to an exact version of Hypothesis, and we would usually recommend implementing a shared database with a network datastore - see `ExampleDatabase`, and the `MultiplexedDatabase` helper.

```python
class hypothesis.database.ReadOnlyDatabase(db)
A wrapper to make the given database read-only.

The implementation passes through `fetch`, and turns `save`, `delete`, and `move` into silent no-ops.

Note that this disables Hypothesis’ automatic discarding of stale examples. It is designed to allow local machines to access a shared database (e.g. from CI servers), without propagating changes back from a local or in-development branch.
```

```python
class hypothesis.database.MultiplexedDatabase(*dbs)
A wrapper around multiple databases.

Each `save`, `fetch`, `move`, or `delete` operation will be run against all of the wrapped databases. `fetch` does not yield duplicate values, even if the same value is present in two or more of the wrapped databases.

This combines well with a `ReadOnlyDatabase`, as follows:

```python
local = DirectoryBasedExampleDatabase("/tmp/hypothesis/examples/")
shared = CustomNetworkDatabase()

settings.register_profile("ci", database=shared)
settings.register_profile(
    "dev", database=MultiplexedDatabase(local, ReadOnlyDatabase(shared))
)
settings.load_profile("ci" if os.environ.get("CI") else "dev")
```

So your CI system or fuzzing runs can populate a central shared database; while local runs on development machines can reproduce any failures from CI but will only cache their own failures locally and cannot remove examples from the shared database.

```python
class hypothesis.extra.redis.RedisExampleDatabase(redis, *, expire_after=datetime.timedelta(days=8),
    key_prefix=b'\hypothesis-example:')
Store Hypothesis examples as sets in the given `redis.Redis` datastore.

This is particularly useful for shared databases, as per the recipe for a `MultiplexedDatabase`.
```

**Note:** If a test has not been run for `expire_after`, those examples will be allowed to expire. The default time-to-live persists examples between weekly runs.
10.4 Defining your own ExampleDatabase

You can define your `ExampleDatabase`, for example to use a shared datastore, with just a few methods:

```python
class hypothesis.database.ExampleDatabase(*args, **kwargs)
    An abstract base class for storing examples in Hypothesis' internal format.
    An ExampleDatabase maps each `bytes` key to many distinct `bytes` values, like a `Mapping[bytes, AbstractSet[bytes]]`.

    abstract save(key, value)
        Save value under key.
        If this value is already present for this key, silently do nothing.

    abstract fetch(key)
        Return an iterable over all values matching this key.

    abstract delete(key, value)
        Remove this value from this key.
        If this value is not present, silently do nothing.

    move(src, dest, value)
        Move value from key src to key dest. Equivalent to delete(src, value) followed by save(src, value), but may have a more efficient implementation.
        Note that value will be inserted at dest regardless of whether it is currently present at src.
```
STATEFUL TESTING

With \texttt{@given}, your tests are still something that you mostly write yourself, with Hypothesis providing some data. With Hypothesis's \textit{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.

\textbf{Tip:} Before reading this reference documentation, we recommend reading \textit{How not to Die Hard with Hypothesis} and \textit{An Introduction to Rule-Based Stateful Testing}, in that order. The implementation details will make more sense once you've seen them used in practice, and know \textit{why} each method or decorator is available.

\textbf{Note:} This style of testing is often called \textit{model-based testing}, but in Hypothesis is called \textit{stateful testing} (mostly for historical reasons - the original implementation of this idea in Hypothesis was more closely based on ScalaCheck's \textit{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.

\section*{11.1 You may not need state machines}

The basic idea of stateful testing is to make Hypothesis choose actions as well as values for your test, and state machines are a great declarative way to do just that.

For simpler cases though, you might not need them at all - a standard test with \texttt{@given} might be enough, since you can use \texttt{data()} in branches or loops. In fact, that’s how the state machine explorer works internally. For more complex workloads though, where a higher level API comes into it’s own, keep reading!
11.2 Rule-based state machines

```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().__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 add_key(self, k):
            return k

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

A RuleBasedStateMachine gives you a structured way to define state machines.

The idea is that a state machine carries a bunch of types of data divided into Bundles, and has a set of rules which may read data from bundles (or just from normal strategies) and push data onto bundles. At any given point a random applicable rule will be executed.

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.

Rules can take normal strategies as arguments, or a specific kind of strategy called a Bundle. 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.

You can think of each value that gets added to any Bundle as being assigned to a new variable. Drawing a value from the bundle strategy means choosing one of the corresponding variables and using that value, and `consumes()` as a del statement for that variable. If you can replace use of Bundles with instance attributes of the class that is often simpler, but often Bundles are strictly more powerful.

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.
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:

TestDBComparison = DatabaseComparison.TestCase

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()
var1 = state.add_key(b''
var2 = state.add_value(v=var1)
state.save(k=var1, v=var2)
state.delete(k=var1, v=var2)
state.values_agree(k=var1)
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.

11.2. Rule-based state machines

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

```python
DatabaseComparison.TestCase.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.

### 11.3 Rules

As said earlier, rules are the most common feature used in RuleBasedStateMachine. They are defined by applying the `rule()` decorator on a function. Note that RuleBasedStateMachine must have at least one rule defined and that a single function cannot be used to define multiple rules (this to avoid having multiple rules doing the same things). Due to the stateful execution method, rules generally cannot take arguments from other sources such as fixtures or `pytest.mark.parametrize` - consider providing them via a strategy such as `sampled_from()` instead.

```python
hypothesis.stateful.rule(*targets=(), target=None, **kwargs)
```

Decorator for RuleBasedStateMachine. Any Bundle present in `target` or `targets` will define where the end result of this function should go. If both are empty then the end result will be discarded.

- `target` must be a Bundle, or if the result should go to multiple bundles you can pass a tuple of them as the `targets` argument. It is invalid to use both arguments for a single rule. If the result should go to exactly one of several bundles, define a separate rule for each case.
- `kwargs` then define the arguments that will be passed to the function invocation. If their value is a Bundle, or if it is `consumes(b)` where `b` is a Bundle, then values that have previously been produced for that bundle will be provided. If `consumes` is used, the value will also be removed from the bundle.
- Any other `kwargs` should be strategies and values from them will be provided.

```python
hypothesis.stateful.consumes(bundle)
```

When introducing a rule in a RuleBasedStateMachine, this function can be used to mark bundles from which each value used in a step with the given rule should be removed. This function returns a strategy object that can be manipulated and combined like any other.

- For example, a rule declared with
  ```python
  @rule(value1=b1, value2=consumes(b2), value3=lists(consumes(b3)))
  ```
  will consume a value from Bundle `b2` and several values from Bundle `b3` to populate `value2` and `value3` each time it is executed.

```python
hypothesis.stateful.multiple(*args)
```

This function can be used to pass multiple results to the target(s) of a rule. Just use `return multiple(result1, result2, ...) in your rule.

- It is also possible to use `return multiple()` with no arguments in order to end a rule without passing any result.
11.4 Initializes

Initializes are a special case of rules that are guaranteed to be run at most once at the beginning of a run (i.e. before any normal rule is called). Note if multiple initialize rules are defined, they may be called in any order, and that order will vary from run to run.

Initializes are typically useful to populate bundles:

```python
hypothesis.stateful.initialize(*, targets=None, target=None, **kwargs)

Decorator for RuleBasedStateMachine.

An initialize decorator behaves like a rule, but all @initialize() decorated methods will be called before any @rule() decorated methods, in an arbitrary order. Each @initialize() method will be called exactly once per run, unless one raises an exception - after which only the .teardown() method will be run. @initialize() methods may not have preconditions.
```

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

name_strategy = st.text(min_size=1).filter(lambda x: '/' not in x)

class NumberModifier(RuleBasedStateMachine):
    folders = Bundle("folders")
    files = Bundle("files")

    @initialize(target=folders)
    def init_folders(self):
        return "/"

    @rule(target=folders, name=name_strategy)
    def create_folder(self, parent, name):
        return f"{parent}/{name}"

    @rule(target=files, name=name_strategy)
    def create_file(self, parent, name):
        return f"{parent}/{name}"
```

11.5 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.

```python
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, which is more efficient than using `assume()` within the rule. The `precond` 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:
If multiple preconditions are applied to a single rule, it is only considered a valid step when all of them return True. Preconditions may be applied to invariants as well as rules.

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.

### 11.6 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.

```python
hypothesis.stateful.invariant(*, check_during_init=False)
```

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:

```python
class MyTestMachine(RuleBasedStateMachine):
    state = 1
    
    @invariant()
    def is_nonzero(self):
        assert self.state != 0
```

By default, invariants are only checked after all `@initialize()` rules have been run. Pass `check_during_init=True` for invariants which can also be checked during initialization.
import RuleBasedStateMachine, invariant, rule

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.

## 11.7 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 RuleBasedStateMachine and an optional settings parameter and does the same as the class based runTest provided.

This is not recommended as it bypasses some important internal functions, including reporting of statistics such as runtimes and event() calls. It was originally added to support custom __init__ methods, but you can now use initialize() rules instead.
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.

### 12.1 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.

Documented APIs will not break except between major version bumps. All APIs mentioned in this documentation are public unless explicitly noted as provisional, in which case they may be changed in minor releases. Undocumented attributes, modules, and behaviour may include breaking changes in patch releases.

### 12.2 Python versions

Hypothesis is supported and tested on CPython 3.6+, i.e. all versions of CPython with upstream support, Hypothesis also supports the latest PyPy for Python 3.6. 32-bit builds of CPython also work, though they are 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.

### 12.3 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.
12.4 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, or ask the framework maintainer to support our hooks for inserting such a wrapper later.

In terms of what’s actually known to work:

- Hypothesis integrates as smoothly with pytest and unittest as we can make it, and this is verified as part of the CI.
- pytest 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 with @given is meaningless.
- The unittest.mock.patch() decorator works with @given, but we recommend using it as a context manager within the decorated test to ensure that the mock is per-test-case and avoid poor interactions with Pytest fixtures.
- 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 once per test rather than once per example, which is not what you want.
- Coverage works out of the box with Hypothesis; our own test suite has 100% branch coverage.

12.5 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.

12.6 Regularly verifying this

Everything mentioned above as explicitly supported is checked on every commit with GitHub Actions. Our continuous delivery pipeline runs all of these checks before publishing each release, so when we say they’re supported we really mean it.
This is a collection of examples of how to use Hypothesis in interesting ways. It’s small for now but will grow over time.

All of these examples are designed to be run under pytest, and nose should work too.

## 13.1 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:
    def __init__(self, label, value):
        self.label = label
        self.value = tuple(value)

    def __repr__(self):
        return f"Node({self.label!r}, {self.value!r})"

    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:
    def __init__(self, node):
```

(continues on next page)
self.value = node

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

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

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

But now we want to test that it works.

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

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

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

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

First we need to define a strategy for Node:

import hypothesis.strategies as st

NodeStrategy = st.builds(Node, st.integers(), st.lists(st.booleans(), max_size=10))

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

We can now write a test:

from hypothesis import given

given(st.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,))]

88 Chapter 13. Some more examples
The reason for this is that because False is not a prefix of (True, True) nor vice versa, sorting things the first two nodes are equal because they have equal labels. This makes the whole order non-transitive and produces basically nonsense results.

But this is pretty unsatisfying. It only works because they have the same label. Perhaps we actually wanted our labels to be unique. Let’s change the test to do that.

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

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

```python
@given(st.lists(NodeStrategy).map(deduplicate_nodes_by_label))
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:

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

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

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

```python
def sort_nodes(xs):
    for i in range(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.
13.2 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.

```python
from datetime import timedelta

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

timedeltas = st.timedeltas()

@given(aware_datetimes, timedeltas, timedeltas)
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, timedeltas)
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, timedeltas)
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, timedeltas)
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
```
13.3 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! The main trick is to decide how we want to represent the result of an election - for this example, we’ll use a list of “votes”, where each vote is a list of candidates in the voters preferred order. Without further ado, here is the code:

```python
from collections import Counter

from hypothesis import given
from hypothesis.strategies import lists, permutations

# We need at least three candidates and at least three voters to have a paradox; anything less can only lead to victories or at worst ties.
@given(lists(permutations(['A', 'B', 'C']), min_size=3))
def test_elections_are_transitive(election):
    all_candidates = {'A', 'B', 'C'}

    # 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 = {}
    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:

```


Which does indeed do the job: The majority (votes 0 and 1) prefer B to C, the majority (votes 0 and 2) prefer A to B and the majority (votes 1 and 2) prefer C to A. This is in fact basically the canonical example of the voting paradox.
13.4 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 arbitrary 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 arbitrary 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. The schemathesis package provides an excellent example of this!

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

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):
    (continues on next page)
@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,
    )

    # Let's 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()
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. If you post a question on Stack Overflow, please use the python-hypothesis tag!

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

If you would like to cite Hypothesis, please consider our suggested citation.

If you like repo badges, we suggest the following badge, which you can add with reStructuredText or Markdown, respectively:

```plaintext
.. image:: https://img.shields.io/badge/hypothesis-tested-brightgreen.svg
   :alt: Tested with Hypothesis
   :target: https://hypothesis.readthedocs.io
```

Finally, we have a beautiful logo which appears online, and often on stickers:

As well as being beautiful, dragonflies actively hunt down bugs for a living! You can find the images and a usage guide in the brand directory on GitHub, or find us at conferences where we often have stickers and sometimes other swag.
CHAPTER
FIFTEEN

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.
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.

16.1 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.

16.2 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.

16.3 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!
16.4 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.

16.5 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.

16.6 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).

16.7 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.

16.8 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.
16.9 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.

16.10 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, pytest can handle example-based testing, property-based testing is increasingly far more important than example-base testing, and Hypothesis fits the bill.

16.11 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.

16.12 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.

16.13 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.

16.14 Florian Kromer

At Roboception GmbH I use Hypothesis to implement fully automated stateless and stateful reliability tests for the 3D sensor rc_visard and robotic software components.

Thank you very much for creating the (probably) most powerful property-based testing framework.
16.15 Reposist Power

With a micro-service architecture, testing between services is made easy using Hypothesis in integration testing. Ensuring everything is running smoothly is vital to help maintain a secure network of Virtual Power Plants.

It allows us to find potential bugs and edge cases with relative ease and minimal overhead. As our architecture relies on services communicating effectively, Hypothesis allows us to strictly test for the kind of data which moves around our services, particularly our backend Python applications.

16.16 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, and thousands on GitHub. Hypothesis has over 100,000 downloads per week, and was used by more than 4% of Python users surveyed by the PSF in 2020.

- aur
- argon2_cffi
- attrs
- axelrod
- bidict
- binaryornot
- brotlicffi
- cmph-cffi
- cryptography
- dbus-signature-pyparsing
- dry-python/returns
- fastnumbers
- flocker
- flownetpy
- funsize
- fusion-index
- hyper-h2
- into-dbus-python
- justbases
- justbytes
- loris
- mariadb-dyncol
• mercurial
• natsort
• poliastro
• pretext
• priority
• PyCEbox
• PyPy
• pyrsistent
• python-humble-utils
• pyudev
• qutebrowser
• RubyMarshal
• Segpy
• simoa
• srt
• tchannel
• vdirsyncer
• wcag-contrast-ratio
• yacluster
• yturl
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 by keyword or filter by classifier, or search 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.

### 18.1 External strategies

Some packages provide strategies directly:

- **hypothesis-fspaths** - strategy to generate filesystem paths.
- **hypothesis-geojson** - strategy to generate GeoJson.
- **hypothesis-geometry** - strategies to generate geometric objects.
- **hs-dbus-signature** - strategy to generate arbitrary D-Bus signatures.
- **hypothesis-sqlalchemy** - strategies to generate SQLAlchemy objects.
- **hypothesis-ros** - strategies to generate messages and parameters for the Robot Operating System.
- **hypothesis-csv** - strategy to generate CSV files.
- **hypothesis-networkx** - strategy to generate networkx graphs.
- **hypothesis-bio** - strategies for bioinformatics data, such as DNA, codons, FASTA, and FASTQ formats.
- **hypothesmith** - strategy to generate syntactically-valid Python code.

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

- **hypothesis-jsonschema** - infer strategies from JSON schemas.
- **lollipop-hypothesis** - infer strategies from lollipop schemas.
- **hypothesis-drf** - infer strategies from a djangorestframework serialiser.
- **hypothesis-graphql** - infer strategies from GraphQL schemas.
- **hypothesis-mongoengine** - infer strategies from a mongoengine model.
- **hypothesis-pb** - infer strategies from Protocol Buffer schemas.

Or some other custom integration, such as a “hypothesis” entry point:

- **deal** is a design-by-contract library with built-in Hypothesis support.
• `icontract-hypothesis` infers strategies from `icontract` code contracts.

• Pandera schemas all have a `.strategy()` method, which returns a strategy for matching DataFrames.

• Pydantic automatically registers constrained types - so `builds()` and `from_type()` “just work” regardless of the underlying implementation.

### 18.2 Other cool things

`schemathesis` is a tool for testing web applications built with Open API / Swagger specifications. It reads the schema and generates test cases which will ensure that the application is compliant with its schema. The application under test could be written in any language, the only thing you need is a valid API schema in a supported format. Includes CLI and convenient `pytest` integration. Powered by Hypothesis and `hypothesis-jsonschema`, inspired by the earlier `swagger-conformance` library.

`Trio` is an async framework with “an obsessive focus on usability and correctness”, so naturally it works with Hypothesis! `pytest-trio` includes a custom hook that allows `@given(...)` to work with Trio-style async test functions, and `hypothesis-trio` includes stateful testing extensions to support concurrent programs.

`pymtl3` is “an open-source Python-based hardware generation, simulation, and verification framework with multi-level hardware modeling support”, which ships with Hypothesis integrations to check that all of those levels are equivalent, from function-level to register-transfer level and even to hardware.

`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-subtest` hack functions as a workaround for issue #377.

`returns` uses Hypothesis to verify that Higher Kinded Types correctly implement functor, applicative, monad, and other laws; allowing a declarative approach to be combined with traditional pythonic code.

`icontract-hypothesis` includes a ghostwriter for test files and IDE integrations such as `icontract-hypothesis-vim`, `icontract-hypothesis-pycharm`, and `icontract-hypothesis-vscode` - you can run a quick ‘smoke test’ with only a few keystrokes for any type-annotated function, even if it doesn’t have any contracts!

### 18.3 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. And make sure you use the `Framework :: Hypothesis` trove classifier!

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`.

Tools such as assertion helpers may also need to check whether the current test is using Hypothesis:
hypothesis.currently_in_test_context()

Return True if the calling code is currently running inside an @given or stateful test, False otherwise.

This is useful for third-party integrations and assertion helpers which may be called from traditional or property-based tests, but can only use assume() or target() in the latter case.

18.4 Registering strategies via setuptools entry points

If you would like to ship Hypothesis strategies for a custom type - either as part of the upstream library, or as a third-party extension, there's a catch: from_type() only works after the corresponding call to register_type_strategy(). This means that either

- you have to try importing Hypothesis to register the strategy when your library is imported, though that's only useful at test time, or
- the user has to call a ‘register the strategies’ helper that you provide before running their tests

**Entry points** are Python's standard way of automating the latter: when you register a "hypothesis" entry point in your setup.py, we'll import and run it automatically when hypothesis is imported. Nothing happens unless Hypothesis is already in use, and it’s totally seamless for downstream users!

Let's look at an example. You start by adding a function somewhere in your package that does all the Hypothesis-related setup work:

```python
# mymodule.py

class MyCustomType:
    def __init__(self, x: int):
        assert x >= 0, f"got {x}, but only positive numbers are allowed"
        self.x = x

def _hypothesis_setup_hook():
    import hypothesis.strategies as st
    st.register_type_strategy(MyCustomType, st.integers(min_value=0))
```

and then tell setuptools that this is your "hypothesis" entry point:

```python
# setup.py

# You can list a module to import by dotted name
entry_points = {"hypothesis": ["_ = mymodule.a_submodule"]}

# Or name a specific function too, and Hypothesis will call it for you
entry_points = {"hypothesis": ["_ = mymodule:_hypothesis_setup_hook"]}
```

And that’s all it takes!

**Note:** On Python 3.6 and 3.7, where the importlib.metadata module is not in the standard library, loading entry points requires either the importlib_metadata (preferred) or setuptools (fallback) package to be installed.
18.4.1 Interaction with pytest-cov

Because pytest does not load plugins from entrypoints in any particular order, using the Hypothesis entrypoint may import your module before pytest-cov starts. This is a known issue, but there are workarounds.

You can use `coverage run pytest ...` instead of `pytest --cov ...`, opting out of the pytest plugin entirely. Alternatively, you can ensure that Hypothesis is loaded after coverage measurement is started by disabling the entrypoint, and loading our pytest plugin from your `conftest.py` instead:

```
echo "pytest_plugins = ['hypothesis.extra.pytestplugin']\n" > tests/conftest.py
pytest -p "no:hypothesispyp tst" ...
```
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 PyPI.

19.1 Hypothesis 6.x

19.1.1 6.25.0 - 2021-11-19

This release adds special filtering logic to make a few special cases like \texttt{s.map(lambda x: x)} and \texttt{lists().filter(len)} more efficient (issue \#2701).

19.1.2 6.24.6 - 2021-11-18

This patch makes \texttt{hypothesis.strategies.floats()} generate “subnormal” floating point numbers more often, as these rare values can have strange interactions with unsafe compiler optimisations like -ffast-math (issue \#2976).

19.1.3 6.24.5 - 2021-11-16

This patch fixes a rare internal error in the \texttt{hypothesis.strategies.datetimes()} strategy, where the implementation of \texttt{allow_imaginary=False} crashed when checking a time during the skipped hour of a DST transition if the DST offset is negative - only true of Europe/Dublin, who we presume have their reasons - and the \texttt{tzinfo} object is a \texttt{pytz} timezone (which predates PEP 495).

19.1.4 6.24.4 - 2021-11-15

This patch gives Hypothesis its own internal \texttt{Random} instance, ensuring that test suites which reset the global random state don’t induce weird correlations between property-based tests (issue \#2135).
19.1.5 6.24.3 - 2021-11-13
This patch updates documentation of `note()` (issue #3147).

19.1.6 6.24.2 - 2021-11-05
This patch updates internal testing for the Array API extra to be consistent with new specification changes: `sum()` not accepting boolean arrays (#234), `unique()` split into separate functions (#275), and treating NaNs as distinct (#310). It has no user visible impact.

19.1.7 6.24.1 - 2021-11-01
This patch updates our vendored list of top-level domains, which is used by the provisional `domains()` strategy.

19.1.8 6.24.0 - 2021-10-23
This patch updates our vendored list of top-level domains, which is used by the provisional `domains()` strategy. (did you know that gTLDs can be both added and removed?)

19.1.9 6.23.4 - 2021-10-20
This patch adds an error for when shapes in `xps.arrays()` is not passed as either a valid shape or strategy.

19.1.10 6.23.3 - 2021-10-18
This patch updates our formatting with shed.

19.1.11 6.23.2 - 2021-10-08
This patch replaces external links to NumPy API docs with sphinx.ext.intersphinx cross-references. It is purely a documentation improvement.

19.1.12 6.23.1 - 2021-09-29
This patch cleans up internal logic for `xps.arrays()`. There is no user-visible change.

19.1.13 6.23.0 - 2021-09-26
This release follows pytest in considering SystemExit and GeneratorExit exceptions to be test failures, meaning that we will shink to minimal examples and check for flakiness even though they subclass BaseException directly (issue #2223). KeyboardInterrupt continues to interrupt everything, and will be re-raised immediately.
19.1.14 6.22.0 - 2021-09-24

This release adds `LiveServerTestCase` and `StaticLiveServerTestCase` for django test. Thanks to Ivan Tham for this feature!

19.1.15 6.21.6 - 2021-09-19

This patch fixes some new linter warnings such as flake8-bugbear’s B994 for explicit exception chaining, so tracebacks might be a bit nicer.

19.1.16 6.21.5 - 2021-09-16

This release fixes None being inferred as the float64 dtype in `from_dtype()` and `arrays()` from the `Array API extra`.

19.1.17 6.21.4 - 2021-09-16

This release fixes the type hint for the `@given()` decorator when decorating an async function (issue #3099).

19.1.18 6.21.3 - 2021-09-15

This release improves Ghostwritten tests for builtins (issue #2977).

19.1.19 6.21.2 - 2021-09-15

This release deprecates use of both `min_dims > len(shape)` and `max_dims > len(shape)` when `allow_newaxis == False` in `basic_indices()` (issue #3091).

19.1.20 6.21.1 - 2021-09-13

This release improves the behaviour of `builds()` and `from_type()` in certain situations involving decorators (issue #2495 and issue #3029).

19.1.21 6.21.0 - 2021-09-11

This release introduces strategies for array/tensor libraries adopting the `Array API standard` (issue #3037). They are available in `the hypothesis.extra.array_api extra`, and work much like the existing strategies for NumPy.

19.1.22 6.20.1 - 2021-09-10

This patch fixes issue #961, where calling `given()` inline on a bound method would fail to handle the `self` argument correctly.
19.1.23 6.20.0 - 2021-09-09

This release allows \texttt{slices()} to generate \texttt{step=None}, and fixes an off-by-one error where the \texttt{start} index could be equal to \texttt{size}. This works fine for all Python sequences and Numpy arrays, but is undefined behaviour in the Array API standard (see pull request \#3065).

19.1.24 6.19.0 - 2021-09-08

This release makes \textit{stateful testing} more likely to tell you if you do something unexpected and unsupported:

- The \texttt{return_value} health check now applies to \texttt{rule()} and \texttt{initialize()} rules, if they don’t have \texttt{target} bundles, as well as \texttt{invariant()}.
- Using a \texttt{consumes()} bundle as a \texttt{target} is deprecated, and will be an error in a future version.

If existing code triggers these new checks, check for related bugs and misunderstandings - these patterns \textit{never} had any effect.

19.1.25 6.18.0 - 2021-09-06

This release teaches \texttt{from_type()} a neat trick: when resolving an \texttt{typing.Annotated} type, if one of the annotations is a strategy object we use that as the inferred strategy. For example:

\begin{verbatim}
PositiveInt = Annotated[int, st.integers(min_value=1)]
\end{verbatim}

If there are multiple strategies, we use the last outer-most annotation. See issue \#2978 and pull request \#3082 for discussion.

\textit{Requires Python 3.9 or later for get_type_hints(..., include_extras=False)}.

19.1.26 6.17.4 - 2021-08-31

This patch makes unique \texttt{arrays()} much more efficient, especially when there are only a few valid elements - such as for eight-bit integers (issue \#3066).

19.1.27 6.17.3 - 2021-08-30

This patch fixes the repr of \texttt{array_shapes()}.

19.1.28 6.17.2 - 2021-08-30

This patch wraps some internal helper code in our proxies decorator to prevent mutations of method docstrings carrying over to other instances of the respective methods.
19.1.29 6.17.1 - 2021-08-29

This patch moves some internal helper code in preparation for issue #3065. There is no user-visible change, unless you depended on undocumented internals.

19.1.30 6.17.0 - 2021-08-27

This release adds type annotations to the stateful testing API.
Thanks to Ruben Opdebeeck for this contribution!

19.1.31 6.16.0 - 2021-08-27

This release adds the DrawFn type as a reusable type hint for the draw argument of @composite functions.
Thanks to Ruben Opdebeeck for this contribution!

19.1.32 6.15.0 - 2021-08-22

This release emits a more useful error message when @given() is applied to a coroutine function, i.e. one defined using async def (issue #3054).
This was previously only handled by the generic return_value health check, which doesn’t direct you to use either a custom executor or a library such as pytest-trio or pytest-asyncio to handle it for you.

19.1.33 6.14.9 - 2021-08-20

This patch fixes a regression in Hypothesis 6.14.8, where from_type() failed to resolve types which inherit from multiple parametrised generic types, affecting the returns package (issue #3060).

19.1.34 6.14.8 - 2021-08-16

This patch ensures that registering a strategy for a subclass of a a parametrised generic type such as class Lines(Sequence[str]): will not “leak” into unrelated strategies such as st.from_type(Sequence[int]) (issue #2951). Unfortunately this fix requires PEP 560, meaning Python 3.7 or later.

19.1.35 6.14.7 - 2021-08-14

This patch fixes issue #3050, where attrs classes could cause an internal error in the ghostwriter.
19.1.36 6.14.6 - 2021-08-07

This patch improves the error message for issue #3016, where PEP 585 built-in generics with self-referential forward-reference strings cannot be resolved to a strategy by `from_type()`.

19.1.37 6.14.5 - 2021-07-27

This patch fixes `hypothesis.strategies._internal.types.is_a_new_type`. It was failing on Python 3.10.0b4, where `NewType` is a function.

19.1.38 6.14.4 - 2021-07-26

This patch fixes `from_type()` and `register_type_strategy()` for `typing.NewType` on Python 3.10, which changed the underlying implementation (see bpo-44353 for details).

19.1.39 6.14.3 - 2021-07-18

This patch updates our autoformatting tools, improving our code style without any API changes.

19.1.40 6.14.2 - 2021-07-12

This patch ensures that we shorten traceback for tests which fail due to inconsistent data generation between runs (i.e. raise `Flaky`).

19.1.41 6.14.1 - 2021-07-02

This patch updates some internal type annotations. There is no user-visible change.

19.1.42 6.14.0 - 2021-06-09

The `explain phase` now requires shrinking to be enabled, and will be automatically skipped for deadline-exceeded errors.

19.1.43 6.13.14 - 2021-06-04

This patch improves the `tuples()` strategy type annotations, to preserve the element types for up to length-five tuples (issue #3005).

As for `one_of()`, this is the best we can do before a planned extension to PEP 646 is released, hopefully in Python 3.11.
19.1.44 6.13.13 - 2021-06-04

This patch teaches the Ghostwriter how to find custom ufuncs from any module that defines them, and that yaml.unsafe_load() does not undo yaml.safe_load().

19.1.45 6.13.12 - 2021-06-03

This patch reduces the amount of internal code excluded from our test suite’s code coverage checks.
There is no user-visible change.

19.1.46 6.13.11 - 2021-06-02

This patch removes some old internal helper code that previously existed to make Python 2 compatibility easier.
There is no user-visible change.

19.1.47 6.13.10 - 2021-05-30

This release adjusts some internal code to help make our test suite more reliable.
There is no user-visible change.

19.1.48 6.13.9 - 2021-05-30

This patch cleans up some internal code related to filtering strategies.
There is no user-visible change.

19.1.49 6.13.8 - 2021-05-28

This patch slightly improves the performance of some internal code for generating integers.

19.1.50 6.13.7 - 2021-05-27

This patch fixes a bug in from_regex() that caused from_regex(“”, fullmatch=True) to unintentionally generate non-empty strings (issue #4982).
The only strings that completely match an empty regex pattern are empty strings.

19.1.51 6.13.6 - 2021-05-26

This patch fixes a bug that caused integers() to shrink towards negative values instead of positive values in some cases.
19.1.52 6.13.5 - 2021-05-24

This patch fixes rare cases where `hypothesis write --binary-op` could print `reproducing instructions` from the internal search for an identity element.

19.1.53 6.13.4 - 2021-05-24

This patch removes some unnecessary intermediate list-comprehensions, using the latest versions of `pyupgrade` and `shed`.

19.1.54 6.13.3 - 2021-05-23

This patch adds a `.hypothesis` property to invalid test functions, bringing them inline with valid tests and fixing a bug where `pytest-asyncio` would swallow the real error message and mistakenly raise a version incompatibility error.

19.1.55 6.13.2 - 2021-05-23

Some of Hypothesis’s numpy/pandas strategies use a `fill` argument to speed up generating large arrays, by generating a single fill value and sharing that value among many array slots instead of filling every single slot individually.

When no `fill` argument is provided, Hypothesis tries to detect whether it is OK to automatically use the `elements` argument as a fill strategy, so that it can still use the faster approach.

This patch fixes a bug that would cause that optimization to trigger in some cases where it isn’t 100% guaranteed to be OK.

If this makes some of your numpy/pandas tests run more slowly, try adding an explicit `fill` argument to the relevant strategies to ensure that Hypothesis always uses the faster approach.

19.1.56 6.13.1 - 2021-05-20

This patch strengthens some internal import-time consistency checks for the built-in strategies.

There is no user-visible change.

19.1.57 6.13.0 - 2021-05-18

This release adds URL fragment generation to the `urls()` strategy (issue #2908). Thanks to Pax (R. Margret) for contributing this patch at the PyCon US Mentored Sprints!

19.1.58 6.12.1 - 2021-05-17

This patch fixes issue #2964, where `.map()` and `.filter()` methods were omitted from the `repr()` of `just()` and `sampled_from()` strategies, since version 5.43.7.
This release automatically rewrites some simple filters, such as `integers().filter(lambda x: x > 9)` to the more efficient `integers(min_value=10)`, based on the AST of the predicate.

We continue to recommend using the efficient form directly wherever possible, but this should be useful for e.g. pandera “Checks” where you already have a simple predicate and translating manually is really annoying. See issue #2701 for ideas about floats and simple text strategies.

*hypothesis.target()* now returns the `observation` value, allowing it to be conveniently used inline in expressions such as `assert target(abs(a - b)) < 0.1`.

This patch fixes a deprecation warning if you’re using recent versions of `importlib-metadata` (issue #2934), which we use to load third-party plugins such as Pydantic’s integration. On older versions of `importlib-metadata`, there is no change and you don’t need to upgrade.

This release teaches the `Ghostwriter` to read parameter types from Sphinx, Google, or Numpy-style structured doc-strings, and improves some related heuristics about how to test scientific and numerical programs.

This release improves the `Ghostwriter`’s handling of exceptions, by reading `:raises ...:` entries in function doc-strings and ensuring that we don’t suppress the error raised by test assertions.

This patch updates our autoformatting tools, improving our code style without any API changes.

This release teaches `from_type()` how to see through `typing.Annotated`. Thanks to Vytautas Strimaitis for reporting and fixing issue #2919!
If `rich` is installed, the `hypothesis write` command will use it to syntax-highlight the *Ghostwritten* code.

This patch improves an error message from `from_type()` when `builds()` would be more suitable (issue #2930).

This patch updates the type annotations for `arrays()` to reflect that `shape: SearchStrategy[int]` is supported.

This patch fixes `from_type()` with abstract types which have either required but non-type-annotated arguments to `__init__`, or where `from_type()` can handle some concrete subclasses but not others.

This patch teaches `hypothesis write` to check for possible roundtrips in several more cases, such as by looking for an inverse in the module which defines the function to test.

This patch adds a more helpful error message if you try to call `sampled_from()` on an `Enum` which has no members, but *does* have `dataclass()`-style annotations (issue #2923).

The `fixed_dictionaries()` strategy now preserves dict iteration order instead of sorting the keys. This also affects the pretty-printing of keyword arguments to `@given()` (issue #2913).

This patch teaches `hypothesis write` to default to ghostwriting tests with `--style=pytest` only if pytest is installed, or `--style=unittest` otherwise.

This patch adds type annotations for the `settings` decorator, to avoid an error when running mypy in strict mode.
19.1.75 6.8.3 - 2021-03-28

This patch improves the *Ghostwriter’s* handling of strategies to generate various fiddly types including frozensets, keysviews, valuesviews, regex matches and patterns, and so on.

19.1.76 6.8.2 - 2021-03-27

This patch fixes some internal typos. There is no user-visible change.

19.1.77 6.8.1 - 2021-03-14

This patch lays more groundwork for filter rewriting ([issue #2701](https://github.com/PythonQuest/Hypothesis/issues/2701)). There is no user-visible change... yet.

19.1.78 6.8.0 - 2021-03-11

This release *registers* the remaining builtin types, and teaches `from_type()` to try resolving `ForwardRef` and `Type` references to built-in types.

19.1.79 6.7.0 - 2021-03-10

This release teaches `RuleBasedStateMachine` to avoid checking `invariant()`s until all `initialize()` rules have been run. You can enable checking of specific invariants for incompletely initialized machines by using `@invariant(check_during_init=True)` ([issue #2868](https://github.com/PythonQuest/Hypothesis/issues/2868)).

In previous versions, it was possible if awkward to implement this behaviour using `precondition()` and an auxiliary variable.

19.1.80 6.6.1 - 2021-03-09

This patch improves the error message when `from_type()` fails to resolve a forward-reference inside a `typing.Type` such as `Type["int"]` ([issue #2565](https://github.com/PythonQuest/Hypothesis/issues/2565)).

19.1.81 6.6.0 - 2021-03-07

This release makes it an explicit error to apply `invariant()` to a `rule()` or `initialize()` rule in *stateful testing*. Such a combination had unclear semantics, especially in combination with `precondition()`, and was never meant to be allowed ([issue #2681](https://github.com/PythonQuest/Hypothesis/issues/2681)).

19.1.82 6.5.0 - 2021-03-07

This release adds the *explain phase*, in which Hypothesis attempts to explain why your test failed by pointing to suspicious lines of code (i.e. those which were always, and only, run on failing inputs). We plan to include “generalising” failing examples in this phase in a future release ([issue #2192](https://github.com/PythonQuest/Hypothesis/issues/2192)).
19.1.83 6.4.3 - 2021-03-04

This patch fixes issue #2794, where nesting deferred() strategies within recursive() strategies could trigger an internal assertion. While it was always possible to get the same results from a more sensible strategy, the convoluted form now works too.

19.1.84 6.4.2 - 2021-03-04

This patch fixes several problems with mypy when --no-implicit-reexport was activated in user projects. Thanks to Nikita Sobolev for fixing issue #2884!

19.1.85 6.4.1 - 2021-03-04

This patch fixes an exception that occurs when using type unions of the typing_extensions Literal backport on Python 3.6. Thanks to Ben Anhalt for identifying and fixing this bug.

19.1.86 6.4.0 - 2021-03-02

This release fixes stateful testing methods with multiple precondition() decorators. Previously, only the outer-most precondition was checked (issue #2681).

19.1.87 6.3.4 - 2021-02-28

This patch refactors some internals of RuleBasedStateMachine. There is no change to the public API or behaviour.

19.1.88 6.3.3 - 2021-02-26

This patch moves some internal code, so that future work can avoid creating import cycles. There is no user-visible change.

19.1.89 6.3.2 - 2021-02-25

This patch enables register_type_strategy() for subclasses of typing.TypedDict. Previously, from_type() would ignore the registered strategy (issue #2872). Thanks to Ilya Lebedev for identifying and fixing this bug!
19.1.90 6.3.1 - 2021-02-24

This release lays the groundwork for automatic rewriting of simple filters, for example converting integers().
filter(lambda x: x > 9) to integers(min_value=10).

Note that this is not supported yet, and we will continue to recommend writing the efficient form directly wherever possible - predicate rewriting is provided mainly for the benefit of downstream libraries which would otherwise have to implement it for themselves (e.g. pandera and icontract-hypothesis). See issue #2701 for details.

19.1.91 6.3.0 - 2021-02-20

The Hypothesis pytest plugin now requires pytest version 4.6 or later. If the plugin detects an earlier version of pytest, it will automatically deactivate itself.

(4.6.x is the earliest pytest branch that still accepts community bugfixes.)

Hypothesis-based tests should continue to work in earlier versions of pytest, but enhanced integrations provided by the plugin (such as --hypothesis-show-statistics and other command-line flags) will no longer be available in obsolete pytest versions.

19.1.92 6.2.0 - 2021-02-12

If you use pytest-html, Hypothesis now includes the summary statistics for each test in the HTML report, whether or not the --hypothesis-show-statistics argument was passed to show them in the command-line output.

19.1.93 6.1.1 - 2021-01-31

This patch updates our automatic code formatting to use shed, which includes autoflake, black, isort, and pyupgrade (issue #2780).

19.1.94 6.1.0 - 2021-01-29

This release teaches Hypothesis to distinguish between errors based on the __cause__ or __context__ of otherwise identical exceptions, which is particularly useful when internal errors can be wrapped by a library-specific or semantically appropriate exception such as:

```python
try:
    do_the_thing(foo, timeout=10)
except Exception as err:
    raise FooError("Failed to do the thing") from err
```

Earlier versions of Hypothesis only see the FooError, while we can now distinguish a FooError raised because of e.g. an internal assertion from one raised because of a TimeoutExceeded exception.
19.1.95 6.0.4 - 2021-01-27

This release prevents a race condition inside `recursive()` strategies. The race condition occurs when the same `recursive()` strategy is shared among tests that are running in multiple threads (issue #2717).

19.1.96 6.0.3 - 2021-01-23

This patch improves the type annotations for `one_of()`, by adding overloads to handle up to five distinct arguments as `Union` before falling back to `Any`, as well as annotating the `| (__or__)` operator for strategies (issue #2765).

19.1.97 6.0.2 - 2021-01-14

This release makes some small improvements to how filtered strategies work. It should improve the performance of shrinking filtered strategies, and may under some (probably rare) circumstances improve the diversity of generated examples.

19.1.98 6.0.1 - 2021-01-13

This patch fixes an interaction where our test statistics handling made Pytest's `--junit-xml` output fail to validate against the strict `xunit2` schema (issue #1975).

19.1.99 6.0.0 - 2021-01-08

Welcome to the next major version of Hypothesis!

There are no new features here, as we release those in minor versions. Instead, 6.0 is a chance for us to remove deprecated features (many already converted into no-ops), and turn a variety of warnings into errors.

If you were running on the last version of Hypothesis 5.x without any Hypothesis deprecation warnings, this will be a very boring upgrade. In fact, nothing will change for you at all.

Changes

- Many functions now use PEP 3102 keyword-only arguments where passing positional arguments was deprecated since 5.5.
- `hypothesis.extra.django.from_model()` no longer accepts `model` as a keyword argument, where it could conflict with fields named “model”.
- `randoms()` now defaults to `use_true_random=False`.
- `complex_numbers()` no longer accepts `min_magnitude=None`; either use `min_magnitude=0` or just omit the argument.
- `hypothesis.provisional.ip4_addr_strings` and `ip6_addr_strings` are removed in favor of `ip_addresses(v=...)`.map(str).
- `register_type_strategy()` no longer accepts generic types with type arguments, which were always pretty badly broken.
- Using function-scoped pytest fixtures is now a health-check error, instead of a warning.
Tip: The `hypothesis codemod` command can automatically refactor your code, particularly to convert positional to keyword arguments where those are now required.

# 19.2 Hypothesis 5.x

## 19.2.1 5.49.0 - 2021-01-07

This release adds the `function_scoped_fixture` health check value, which can be used to suppress the existing warning that appears when `@given` is applied to a test that uses pytest function-scoped fixtures.

(This warning exists because function-scoped fixtures only run once per function, not once per example, which is usually unexpected and can cause subtle problems.)

When this warning becomes a health check error in a future release, suppressing it via Python warning settings will no longer be possible. In the rare case that once-per-function behaviour is intended, it will still be possible to use `function_scoped_fixture` to opt out of the health check error for specific tests.

## 19.2.2 5.48.0 - 2021-01-06

This release adds `hypothesis.currently_in_test_context()`, which can be used to check whether the calling code is currently running inside an `@given` or `stateful` test.

This is most useful for third-party integrations and assertion helpers which may wish to use `assume()` or `target()`, without also requiring that the helper only be used from property-based tests (issue #2581).

## 19.2.3 5.47.0 - 2021-01-05

This release upgrades the import logic for `ghostwritten tests`, handling many cases where imports would previously be missing or from unexpected locations.

## 19.2.4 5.46.0 - 2021-01-04

This release upgrades `from_type()`, to infer strategies for type-annotated arguments even if they have defaults when it otherwise falls back to `builds()` (issue #2708).

## 19.2.5 5.45.0 - 2021-01-04

This release adds the `hypothesis[codemods]` extra, which you can use to check for and automatically fix issues such as use of deprecated Hypothesis APIs (issue #2705).
19.2.6 5.44.0 - 2021-01-03
This patch fixes `from_type()` with the `typing_extensions.Literal` backport on Python 3.6.

19.2.7 5.43.9 - 2021-01-02
This patch fixes issue #2722, where certain orderings of `register_type_strategy()`, `ForwardRef`, and `from_type()` could trigger an internal error.

19.2.8 5.43.8 - 2021-01-02
This patch makes some strategies for collections with a uniqueness constraint much more efficient, including `dictionaries(keys=sampled_from(...), values=...)` and `lists(tuples(sampled_from(...), ...), unique_by=lambda x: x[0])`. (related to issue #2036)

19.2.9 5.43.7 - 2021-01-02
This patch extends our faster special case for `sampled_from()` elements in unique `lists()` to account for chains of `.map(...)` and `.filter(...)` calls (issue #2036).

19.2.10 5.43.6 - 2021-01-02
This patch improves the type annotations on `assume()` and `@reproduce_failure()`.

19.2.11 5.43.5 - 2021-01-01
This patch updates our copyright headers to include 2021. Happy new year!

19.2.12 5.43.4 - 2020-12-24
This change fixes a documentation error in the `database` setting.

The previous documentation suggested that callers could specify a database path string, or the special string `":memory:"`, but this setting has never actually allowed string arguments.

Permitted values are `None`, and instances of `ExampleDatabase`.

19.2.13 5.43.3 - 2020-12-11
This patch fixes issue #2696, an internal error triggered when the `@example` decorator was used and the `verbosity` setting was `quiet`. 
This patch improves the error message from the `data_frames()` strategy when both the `rows` and `columns` arguments are given, but there is a missing entry in `rows` and the corresponding column has no `fill` value (issue #2678).

This patch improves the error message if `builds()` is passed an `Enum` which cannot be called without arguments, to suggest using `sampled_from()` (issue #2693).

This release adds new `timezones()` and `timezone_keys()` strategies (issue #2630) based on the new `zoneinfo` module in Python 3.9.

`pip install hypothesis[zoneinfo]` will ensure that you have the appropriate backports installed if you need them.

This patch fixes an internal error in `datetimes()` with `allow_imaginary=False` where the `timezones` argument can generate `tzinfo=None` (issue #2662).

This patch teaches `hypothesis.extra.django.from_field()` to infer more efficient strategies by inspecting (not just filtering by) field validators for numeric and string fields (issue #1116).

This patch refactors `hypothesis.settings` to use type-annotated keyword arguments instead of `**kwargs`, which makes tab-completion much more useful - as well as type-checkers like `mypy`.

This patch teaches the `magic()` ghostwriter to recognise “en/de” function roundtrips other than the common encode/decode pattern, such as encrypt/decrypt or, encipher/decipher.

This adds a performance optimisation to avoid saving redundant seeds when using the `.fuzz_one_input` hook.
19.2.22  5.41.4 - 2020-11-28

This patch fixes issue #2657, where passing unicode patterns compiled with re.IGNORECASE to from_regex() could trigger an internal error when casefolding a character creates a longer string (e.g. "\u0130".lower() --> "i\u0370").

19.2.23  5.41.3 - 2020-11-18

This patch adds a final fallback clause to our plugin logic to fail with a warning rather than error on Python < 3.8 when neither the importlib_metadata (preferred) or setuptools (fallback) packages are available.

19.2.24  5.41.2 - 2020-11-08

This patch fixes urls() strategy ensuring that ~ (tilde) is treated as one of the url-safe characters (issue #2658).

19.2.25  5.41.1 - 2020-11-03

This patch improves our CLI help and documentation.

19.2.26  5.41.0 - 2020-10-30

Hypothesis now shrinks examples where the error is raised while drawing from a strategy. This makes complicated custom strategies much easier to debug, at the cost of a slowdown for use-cases where you catch and ignore such errors.

19.2.27  5.40.0 - 2020-10-30

This release teaches from_type() how to handle ChainMap, Counter, Deque, Generator, Match, OrderedDict, Pattern, and Set (issue #2654).

19.2.28  5.39.0 - 2020-10-30

from_type() now knows how to resolve PEP 585 parameterized standard collection types, which are new in Python 3.9 (issue #2629).

19.2.29  5.38.1 - 2020-10-26

This patch fixes builds(), so that when passed infer for an argument with a non-Optional type annotation and a default value of None to build a class which defines an explicit __signature__ attribute, either None or that type may be generated.

This is unlikely to happen unless you are using pydantic (issue #2648).
19.2.30 5.38.0 - 2020-10-24
This release improves our support for `@st.composite` on a `classmethod` or `staticmethod` (issue #2578).

19.2.31 5.37.5 - 2020-10-24
This patch fixes `from_type()` with `Iterable[T]` (issue #2645).

19.2.32 5.37.4 - 2020-10-20
This patch teaches the `magic()` ghostwriter to recognise that pairs of functions like `rgb_to_hsv()` and `hsv_to_rgb()` should `roundtrip()`.

19.2.33 5.37.3 - 2020-10-15
This patch improves `builds()` and `from_type()` support for explicitly defined `__signature__` attributes, from version 5.8.3, to support generic types from the `typing` module.
Thanks to Rónán Carrigan for identifying and fixing this problem!

19.2.34 5.37.2 - 2020-10-14
This patch fixes `from_lark()` with version 0.10.1+ of the `lark-parser` package.

19.2.35 5.37.1 - 2020-10-07
This patch fixes some broken links in the `lark` extra documentation.

19.2.36 5.37.0 - 2020-10-03
This release adds a new `RedisExampleDatabase`, along with the `ReadOnlyDatabase` and `MultiplexedDatabase` helpers, to support team workflows where failing examples can be seamlessly shared between everyone on the team - and your CI servers or buildbots.

19.2.37 5.36.2 - 2020-10-02
This patch ensures that if the "hypothesis" entry point is callable, we call it after importing it. You can still use non-callable entry points (like modules), which are only imported.
We also prefer `importlib.metadata` or the backport over `pkg_resources`, which makes `import hypothesis` around 200 milliseconds faster (issue #2571).
This patch adds some helpful suggestions to error messages you might see while learning to use the `@example()` decorator (issue #2611) or the `one_of()` strategy.

This release upgrades the `from_dtype()` strategy to pass optional `**kwargs` to the inferred strategy, and upgrades the `arrays()` strategy to accept an `elements=kwargs` dict to pass through to `from_dtype()`. `arrays(floating_dtypes(), shape, elements={"min_value": -10, "max_value": 10})` is a particularly useful pattern, as it allows for any floating dtype without triggering the roundoff warning for smaller types or sacrificing variety for larger types (issue #2552).

This patch reformats our code with the latest `black` to take advantage of the support for magic trailing commas.

This release significantly improves the performance of Hypothesis’s internal implementation of automaton learning. However this code does not run as part of the user-accessible API so this has no user-visible impact.

This patch ensures that, when the `generate` phases is disabled, we can replay up to `max_examples` examples from the database - which is very useful when `using Hypothesis with a fuzzer`. Thanks to Afrida Tabassum for fixing issue #2585!

This patch changes some internal `struct.Struct.format` strings from `bytes` to `str`, to avoid `BytesWarning` when running `python -bb`. Thanks to everyone involved in `pytest-xdist issue 596, bpo-16349, bpo-21071, and bpo-41777` for their work on this - it was a remarkably subtle issue!

The `target()` function now accepts integers as well as floats.
19.2.45 5.34.1 - 2020-09-11

This patch adds explicit Optional annotations to our public API, to better support users who run mypy with --strict or no_implicit_optional=True.

Thanks to Krzysztof Przybyła for bringing this to our attention and writing the patch!

19.2.46 5.34.0 - 2020-09-11

This release drops support for Python 3.5, which reached end of life upstream on 2020-09-13.

19.2.47 5.33.2 - 2020-09-09

This patch fixes a problem with builds() that was not able to generate valid data for annotated classes with constructors.

Thanks to Nikita Sobolev for fixing issue #2603!

19.2.48 5.33.1 - 2020-09-07

This patch improves the error message from the hypothesis write command if black (required for the ghostwriter) is not installed.

Thanks to Nikita Sobolev for fixing issue #2604!

19.2.49 5.33.0 - 2020-09-06

When reporting failing examples, or tried examples in verbose mode, Hypothesis now identifies which were from @example(...) explicit examples.

19.2.50 5.32.1 - 2020-09-06

This patch contains some internal refactoring. Thanks to Felix Sheldon for fixing issue #2516!

19.2.51 5.32.0 - 2020-09-04

An array drawn from arrays() will own its own memory; previously most arrays returned by this strategy were views.

19.2.52 5.31.0 - 2020-09-04

builds() will use the __signature__ attribute of the target, if it exists, to retrieve type hints. Previously typing.get_type_hints(), was used by default. If argument names varied between the __annotations__ and __signature__, they would not be supplied to the target.

This was particularly an issue for pydantic models which use an alias generator.
**19.2.53 5.30.1 - 2020-09-04**

This patch makes the ghostwriter much more robust when passed unusual modules.

- improved support for non-resolvable type annotations
- `magic()` can now write `equivalent()` tests
- running `magic()` on modules where some names in `__all__` are undefined skips such names, instead of raising an error
- `magic()` now knows to skip mocks
- improved handling of import-time errors found by the ghostwriter CLI

**19.2.54 5.30.0 - 2020-08-30**

`register_type_strategy()` now supports `typing.TypeVar`, which was previously hard-coded, and allows a variety of types to be generated for an unconstrained `TypeVar` instead of just `text()`.

Thanks again to Nikita Sobolev for all your work on advanced types!

**19.2.55 5.29.4 - 2020-08-28**

This release fixes some hard to trigger bugs in Hypothesis’s automata learning code. This code is only run as part of the Hypothesis build process, and not for user code, so this release has no user visible impact.

**19.2.56 5.29.3 - 2020-08-27**

This patch adds type annotations to the `hypothesis.database` module. There is no runtime change, but your typechecker might notice.

**19.2.57 5.29.2 - 2020-08-27**

This patch tracks some additional information in Hypothesis internals, and has no user-visible impact.

**19.2.58 5.29.1 - 2020-08-27**

This release fixes a bug in some Hypothesis internal support code for learning automata. This mostly doesn’t have any user visible impact, although it slightly affects the learned shrink passes so shrinking may be subtly different.

**19.2.59 5.29.0 - 2020-08-24**

This release adds support for *Registering strategies via setuptools entry points*, which allows for smoother integration of third-party Hypothesis extensions and external libraries. Unless you’re publishing a library with Hypothesis integration, you’ll probably only ever use this indirectly!
19.2.60 5.28.0 - 2020-08-24

*from_type()* can now resolve TypeVar instances when the bound is a ForwardRef, so long as that name is in fact defined in the same module as the typevar (no TYPE_CHECKING tricks, sorry). This feature requires Python 3.7 or later.

Thanks to Zac Hatfield-Dodds and Nikita Sobolev for this feature!

19.2.61 5.27.0 - 2020-08-20

This patch adds two new ghostwriters to test binary operations, like `operator.add()`, and Numpy ufuncs and gufuncs like `np.matmul()`.

19.2.62 5.26.1 - 2020-08-19

This release improves the performance of some methods in Hypothesis’s internal automaton library. These are currently only lightly used by user code, but this may result in slightly faster shrinking.

19.2.63 5.26.0 - 2020-08-17

*register_type_strategy()* no longer accepts parametrised user-defined generic types, because the resolution logic was quite badly broken (issue #2537).

Instead of registering a strategy for e.g. `MyCollection[int]`, you should register a function for `MyCollection` and inspect the type parameters within that function.

Thanks to Nikita Sobolev for the bug report, design assistance, and pull request to implement this feature!

19.2.64 5.25.0 - 2020-08-16

Tired of writing tests? Or new to Hypothesis and not sure where to start?

This release is for you! With our new Ghostwriter functions and `hypothesis write ... command-line interface`, you can stop writing tests entirely... or take the source code Hypothesis writes for you as a starting point.

This has been in the works for months, from issue #2118 to versions 5.18.3, 5.23.5, and 5.23.5 - particular thanks to the many people who reviewed pull requests or commented on demos, and to Timothy Crosley’s hypothesis-auto project for inspiration.

19.2.65 5.24.4 - 2020-08-14

This patch adds yet more internal functions to support a new feature we’re working on, like version 5.18.3 and version 5.23.6. We promise it’s worth the wait!
19.2.66 5.24.3 - 2020-08-13

This release fixes a small internal bug in Hypothesis’s internal automaton library. Fortunately this bug was currently impossible to hit in user facing code, so this has no user visible impact.

19.2.67 5.24.2 - 2020-08-12

This release improves shrink quality by allowing Hypothesis to automatically learn new shrink passes for difficult to shrink tests.

The automatic learning is not currently accessible in user code (it still needs significant work on robustness and performance before it is ready for that), but this release includes learned passes that should improve shrinking quality for tests which use any of the `text()`, `floats()`, `datetimes()`, `emails()`, and `complex_numbers()` strategies.

19.2.68 5.24.1 - 2020-08-12

This patch updates some docstrings, without changing runtime behaviour.

19.2.69 5.24.0 - 2020-08-10

The `functions()` strategy has a new argument `pure=True`, which ensures that the same return value is generated for identical calls to the generated function (issue #2538).

Thanks to Zac Hatfield-Dodds and Nikita Sobolev for this feature!

19.2.70 5.23.12 - 2020-08-10

This release removes a number of Hypothesis’s internal “shrink passes” - transformations it makes to a generated test case during shrinking - which appeared to be redundant with other transformations.

It is unlikely that you will see much impact from this. If you do, it will likely show up as a change in shrinking performance (probably slower, maybe faster), or possibly in worse shrunk results. If you encounter the latter, please let us know.

19.2.71 5.23.11 - 2020-08-04

This release fixes a bug in some internal Hypothesis support code. It has no user visible impact.

19.2.72 5.23.10 - 2020-08-04

This release improves the quality of shrunk test cases in some special cases. Specifically, it should get shrinking unstuck in some scenarios which require simultaneously changing two parts of the generated test case.
19.2.73  5.23.9 - 2020-08-03

This release improves the performance of some internal support code. It has no user visible impact, as that code is not currently run during normal Hypothesis operation.

19.2.74  5.23.8 - 2020-07-31

This release adds a heuristic to detect when shrinking has finished despite the fact that there are many more possible transformations to try. This will be particularly useful for tests where the minimum failing test case is very large despite there being many smaller test cases possible, where it is likely to speed up shrinking dramatically.

In some cases it is likely that this will result in worse shrunk test cases. In those cases rerunning the test will result in further shrinking.

19.2.75  5.23.7 - 2020-07-29

This release makes some performance improvements to shrinking. They should only be noticeable for tests that are currently particularly slow to shrink.

19.2.76  5.23.6 - 2020-07-29

This patch adds some more internal functions to support a new feature we’re working on, like version 5.18.3. There is still no user-visible change… yet.

19.2.77  5.23.5 - 2020-07-29

This release makes some changes to internal support code that is not currently used in production Hypothesis. It has no user visible effect at present.

19.2.78  5.23.4 - 2020-07-29

This release improves shrinking quality in some special cases.

19.2.79  5.23.3 - 2020-07-27

This release fixes issue #2507, where lazy evaluation meant that the values drawn from a sampled_from() strategy could depend on mutations of the sampled sequence that happened after the strategy was constructed.

19.2.80  5.23.2 - 2020-07-27

This patch fixes issue #2462, a bug in our handling of unittest.TestCase.subTest(). Thanks to Israel Fruchter for fixing this at the EuroPython sprints!
This release improves the behaviour of the characters() strategy when shrinking, by changing which characters are considered smallest to prefer more “normal” ascii characters where available.

The default print_blob setting is now smarter. It defaults to True in CI and False for local development. Thanks to Hugo van Kemenade for implementing this feature at the EuroPython sprints!

The slices() strategy can now generate slices for empty sequences, slices with negative start and stop indices (from the end of the sequence), and step=None in place of step=1. Thanks to Sangarshanan for implementing this feature at the EuroPython sprints!

This release ensures that tests which raise RecursionError are not reported as flaky simply because we run them from different initial stack depths (issue #2494).

This release improves the performance of the sample method on objects obtained from randoms() when use_true_random=False. This should mostly only be noticeable when the sample size is a large fraction of the population size, but may also help avoid health check failures in some other cases.

This release makes some internal changes for testing purposes and should have no user visible effect.

This release fixes a small caching bug in Hypothesis internals that may under some circumstances have resulted in a less diverse set of test cases being generated than was intended.

Fixing this problem revealed some performance problems that could occur during targeted property based testing, so this release also fixes those. Targeted property-based testing should now be significantly faster in some cases, but this may be at the cost of reduced effectiveness.
19.2.88 5.20.1 - 2020-07-17
This patch updates our formatting to use isort 5. There is no user-visible change.

19.2.89 5.20.0 - 2020-07-17
The basic_indices() strategy can now generate bare indexers in place of length-one tuples. Thanks to Andrea for this patch!

19.2.90 5.19.3 - 2020-07-15
This patch removes an internal use of distutils in order to avoid this setuptools warning for some users.

19.2.91 5.19.2 - 2020-07-13
This patch contains a small internal refactoring with no user-visible impact.
Thanks to Andrea for writing this at the SciPy 2020 Sprints!

19.2.92 5.19.1 - 2020-07-12
This release slightly improves shrinking behaviour. This should mainly only impact stateful tests, but may have some minor positive impact on shrinking collections (lists, sets, etc).

19.2.93 5.19.0 - 2020-06-30
This release improves the randoms() strategy by adding support for Random instances where Hypothesis generates the random values rather than having them be “truly” random.

19.2.94 5.18.3 - 2020-06-27
This patch adds some internal functions to support a new feature we’re working on. There is no user-visible change… yet.

19.2.95 5.18.2 - 2020-06-26
This patch improves our docs for the derandomize setting.

19.2.96 5.18.1 - 2020-06-25
This release consists of some internal refactoring to the shrinker in preparation for future work. It has no user visible impact.
19.2.97 5.18.0 - 2020-06-22

This release teaches Hypothesis to shorten tracebacks for explicit examples, as we already do for generated examples, so that you can focus on your code rather than ours.

If you have multiple failing explicit examples, they will now all be reported. To report only the first failure, you can use the `report_multiple_bugs=False` setting as for generated examples.

19.2.98 5.17.0 - 2020-06-22

This patch adds strategy inference for the Literal, NewType, Type, DefaultDict, and TypedDict types from the `typing_extensions` backport on PyPI.

19.2.99 5.16.3 - 2020-06-21

This patch precomputes some of the setup logic for our experimental external fuzzer integration and sets `deadline=None` in fuzzing mode, saving around 150us on each iteration.

This is around two-thirds the runtime to fuzz an empty test with `@given(st.none())`, and nice to have even as a much smaller fraction of the runtime for non-trivial tests.

19.2.100 5.16.2 - 2020-06-19

This patch fixes an internal error when warning about the use of function-scoped fixtures for parametrised tests where the parametrised value contained a % character. Thanks to Bryant for reporting and fixing this bug!

19.2.101 5.16.1 - 2020-06-10

If you pass a list or tuple where a strategy was expected, the error message now mentions `sampled_from()` as an example strategy.

Thanks to the enthusiastic participants in the PyCon Mentored Sprints who suggested adding this hint.

19.2.102 5.16.0 - 2020-05-27

`functions()` can now infer the appropriate returns strategy if you pass a like function with a return-type annotation. Before, omitting the returns argument would generate functions that always returned None.

19.2.103 5.15.1 - 2020-05-21

Fix `from_type()` with generic types under Python 3.9.
19.2.104 5.15.0 - 2020-05-19

This patch fixes an error that happens when multiple threads create new strategies.


Passing min_magnitude=None to complex_numbers() is now deprecated - you can explicitly pass min_magnitude=0, or omit the argument entirely.

19.2.106 5.13.1 - 2020-05-13

This patch fixes an internal error in from_type() for typing.NamedTuple in Python 3.9. Thanks to Michel Salim for reporting and fixing issue #2427!

19.2.107 5.13.0 - 2020-05-12

This release upgrades the test statistics available via the --hypothesis-show-statistics option to include separate information on each of the phases (issue #1555).

19.2.108 5.12.2 - 2020-05-12

This patch teaches the from_type() internals to return slightly more efficient strategies for some generic sets and mappings.

19.2.109 5.12.1 - 2020-05-12

This patch adds a # noqa comment for flake8 3.8.0, which disagrees with mypy about how to write the type of ...

19.2.110 5.12.0 - 2020-05-10

This release limits the maximum duration of the shrinking phase to five minutes, so that Hypothesis does not appear to hang when making very slow progress shrinking a failing example (issue #2340).

If one of your tests triggers this logic, we would really appreciate a bug report to help us improve the shrinker for difficult but realistic workloads.

19.2.111 5.11.0 - 2020-05-07

This release improves the interaction between assume() and the @example() decorator, so that the following test no longer fails with UnsatisfiedAssumption (issue #2125):

```python
@given(value=floats(0, 1))
@example(value=0.56789)  # used to make the test fail!
@pytest.mark.parametrize("threshold", [0.5, 1])
def test_foo(threshold, value):
    assume(value < threshold)
    ...
```
If you have django installed but don’t use it, this patch will make import hypothesis a few hundred milliseconds faster (e.g. 0.704s -> 0.271s).

Thanks to importtime-waterfall for highlighting this problem and Jake Vanderplas for the solution - it’s impossible to misuse code from a module you haven’t imported!

This patch improves the internals of builds() type inference, to handle recursive forward references in certain data-classes. This is useful for e.g. hypothesis’s forthcoming LibCST mode.

This release reverses the order in which some operations are tried during shrinking. This should generally be a slight performance improvement, but most tests are unlikely to notice much difference.

This patch fixes issue #2406, where use of pandas.Timestamp objects as bounds for the datetimes() strategy caused an internal error. This bug was introduced in version 5.8.1.

This release is a small internal refactoring to how shrinking interacts with targeted property-based testing that should have no user visible impact.

This release improves our support for datetimes and times around DST transitions.

times() and datetimes() are now sometimes generated with fold=1, indicating that they represent the second occurrence of a given wall-time when clocks are set backwards. This may be set even when there is no transition, in which case the fold value should be ignored.

For consistency, timezones provided by the pytz package can now generate imaginary times (such as the hour skipped over when clocks ‘spring forward’ to daylight saving time, or during some historical timezone transitions). All other timezones have always supported generation of imaginary times.

If you prefer the previous behaviour, datetimes() now takes an argument allow_imaginary which defaults to True but can be set to False for any timezones strategy.
19.2.118 5.9.1 - 2020-04-16
This patch fixes the rendering of `binary()` docstring by using the proper backticks syntax.

19.2.119 5.9.0 - 2020-04-15
Failing tests which use `target()` now report the highest score observed for each target alongside the failing example(s), even without explicitly showing test statistics.
This improves the debugging workflow for tests of accuracy, which assert that the total imprecision is within some error budget - for example, $\text{abs}(a - b) < 0.5$. Previously, shrinking to a minimal failing example could often make errors seem smaller or more subtle than they really are (see the threshold problem, and issue #2180).

19.2.120 5.8.6 - 2020-04-15
This patch improves the docstring of `binary()`, the `repr()` of `sampled_from()` on an `enum.Enum` subclass, and a warning in our pytest plugin. There is no change in runtime behaviour.

19.2.121 5.8.5 - 2020-04-15
This release (potentially very significantly) improves the performance of failing tests in some rare cases, mostly only relevant when using targeted property-based testing, by stopping further optimisation of unrelated test cases once a failing example is found.

19.2.122 5.8.4 - 2020-04-14
This release fixes issue #2395, where under some circumstances targeted property-based testing could cause Hypothesis to get caught in an infinite loop.

19.2.123 5.8.3 - 2020-04-12
This patch teaches `builds()` and `from_type()` to use the `__signature__` attribute of classes where it has been set, improving our support for Pydantic models (in pydantic >= 1.5).

19.2.124 5.8.2 - 2020-04-12
This release improves the performance of the part of the core engine that deliberately generates duplicate values.

19.2.125 5.8.1 - 2020-04-12
This patch improves `dates()` shrinking, to simplify year, month, and day like `datetimes()` rather than minimizing the number of days since 2000-01-01.
19.2.126 5.8.0 - 2020-03-24

This release adds a .hypothesis.fuzz_one_input attribute to @given tests, for easy integration with external fuzzers such as python-afl (supporting issue #171).

19.2.127 5.7.2 - 2020-03-24

This patch fixes issue #2341, ensuring that the printed output from a stateful test cannot use variable names before they are defined.

19.2.128 5.7.1 - 2020-03-23

This patch fixes issue #2375, preventing incorrect failure when a function scoped fixture is overridden with a higher scoped fixture.

19.2.129 5.7.0 - 2020-03-19

This release allows the array_dtypes() strategy to generate Numpy dtypes which have field titles in addition to field names. We expect this to expose latent bugs where code expects that set(dtype.names) == set(dtype.fields), though the latter may include titles.

19.2.130 5.6.1 - 2020-03-18

This makes model a positional-only argument to from_model(), to support models with a field literally named “model” (issue #2369).

19.2.131 5.6.0 - 2020-02-29

This release adds an explicit warning for tests that are both decorated with @given(...) and request a function-scoped pytest fixture, because such fixtures are only executed once for all Hypothesis test cases and that often causes trouble (issue #377).

It’s very difficult to fix this on the pytest side, so since 2015 our advice has been “just don’t use function-scoped fixtures with Hypothesis”. Now we detect and warn about the issue at runtime!

19.2.132 5.5.5 - 2020-02-29

This release cleans up the internal machinery for Stateful testing, after we dropped the legacy APIs in Hypothesis 5.0 (issue #2218). There is no user-visible change.
**19.2.133 5.5.4 - 2020-02-16**

This patch fixes issue #2351, `arrays()` would raise a confusing error if we inferred a strategy for `datetime64` or `timedelta64` values with varying time units.

We now infer an internally-consistent strategy for such arrays, and have a more helpful error message if an inconsistent strategy is explicitly specified.

**19.2.134 5.5.3 - 2020-02-14**

This patch improves the signature of `builds()` by specifying `target` as a positional-only argument on Python 3.8 (see PEP 570). The semantics of `builds()` have not changed at all - this just clarifies the documentation.

**19.2.135 5.5.2 - 2020-02-13**

This release makes Hypothesis faster at generating test cases that contain duplicated values in their inputs.

**19.2.136 5.5.1 - 2020-02-07**

This patch has some tiny internal code clean-ups, with no user-visible change.

**19.2.137 5.5.0 - 2020-02-07**

Our style guide suggests that optional parameters should usually be keyword-only arguments (see PEP 3102) to prevent confusion based on positional arguments - for example, `hypothesis.strategies.floats()` takes up to four boolean flags and many of the Numpy strategies have both `dims` and `side` bounds.

This release converts most optional parameters in our API to use keyword-only arguments - and adds a compatibility shim so you get warnings rather than errors everywhere (issue #2130).

**19.2.138 5.4.2 - 2020-02-06**

This patch fixes compatibility with Python 3.5.2 (issue #2334). Note that we only test the latest patch of each minor version, though as in this case we usually accept pull requests for older patch versions.

**19.2.139 5.4.1 - 2020-02-01**

This patch improves the repr of `from_type()`, so that in most cases it will display the strategy it resolves to rather than `from_type(...)`. The latter form will continue to be used where resolution is not immediately successful, e.g. invalid arguments or recursive type definitions involving forward references.
This release removes support for Python 3.5.0 and 3.5.1, where the `typing` module was quite immature (e.g. missing `overload()` and `Type`).

Note that Python 3.5 will reach its end-of-life in September 2020, and new releases of Hypothesis may drop support somewhat earlier.

Note: `pip install hypothesis` should continue to give you the latest compatible version. If you have somehow ended up with an incompatible version, you need to update your packaging stack to `pip >= 9.0` and `setuptools >= 24.2` - see here for details. Then `pip uninstall hypothesis && pip install hypothesis` will get you back to a compatible version.

This patch does some minor internal cleanup; there is no user-visible change.

The standard library `ipaddress` module is new in Python 3, and this release adds the new `ip_addresses()` strategy to generate `IPv4Address`es and/or `IPv6Address`es (depending on the `v` and `network` arguments).

If you use them in type annotations, `from_type()` now has strategies registered for `ipaddress address`, network, and interface types.

The provisional strategies for IP address strings are therefore deprecated.

This patch reverts version 5.2, due to a strange issue where indexing an array of strings can raise an error instead of returning an item which contains certain surrogate characters.

This release allows `from_dtype()` to generate Unicode strings which cannot be encoded in UTF-8, but are valid in Numpy arrays (which use UTF-32).

This patch fixes issue #2320, where `from_type(Set[Hashable])` could raise an internal error because `Decimal("snan")` is of a hashable type, but raises an error when hashed. We now ensure that set elements and dict keys in generic types can actually be hashed.
19.2.146 5.1.5 - 2020-01-12

This patch fixes an internal error when running in an IPython repl or Jupyter notebook on Windows (issue #2319), and an internal error on Python 3.5.1 (issue #2318).

19.2.147 5.1.4 - 2020-01-11

This patch fixes a bug where errors in third-party extensions such as hypothesis-trio or hypothesis-jsonschema were incorrectly considered to be Hypothesis internal errors, which could result in confusing error messages.
Thanks to Vincent Michel for reporting and fixing the bug!

19.2.148 5.1.3 - 2020-01-11

This release converts the type hint comments on our public API to PEP 484 type annotations.
Thanks to Ivan Levkivskyi for com2ann - with the refactoring tools from 5.0.1 it made this process remarkably easy!

19.2.149 5.1.2 - 2020-01-09

This patch makes multiple() iterable, so that output like a, b = state.some_rule() is actually executable and can be used to reproduce failing examples.
Thanks to Vincent Michel for reporting and fixing issue #2311!

19.2.150 5.1.1 - 2020-01-06

This patch contains many small refactorings to replace our Python 2 compatibility functions with their native Python 3 equivalents. Since Hypothesis is now Python 3 only, there is no user-visible change.

19.2.151 5.1.0 - 2020-01-03

This release teaches from_type() how to generate datetime.timezone. As a result, you can now generate datetime.tzinfo objects without having pytz installed.
If your tests specifically require pytz timezones, you should be using hypothesis.extra.pytz.timezones() instead of st.from_type(tzinfo).

19.2.152 5.0.1 - 2020-01-01

This patch contains mostly-automated refactorings to remove code that we only needed to support Python 2. Since Hypothesis is now Python 3 only (hurray!), there is no user-visible change.
Our sincere thanks to the authors of autoflake, black, isort, and pyupgrade, who have each and collectively made this kind of update enormously easier.
Welcome to the next major version of Hypothesis!

There are no new features here, as we release those in minor versions. Instead, 5.0 is a chance for us to remove deprecated features (many already converted into no-ops), and turn a variety of warnings into errors.

If you were running on the last version of Hypothesis 4.x without any Hypothesis deprecation warnings, this will be a very boring upgrade. In fact, nothing will change for you at all.

Note: This release drops support for Python 2, which has passed its end of life date. The Python 3 Statement outlines our reasons, and lists many other packages that have made the same decision.

pip install hypothesis should continue to give you the latest compatible version. If you have somehow ended up with Hypothesis 5.0 on Python 2, you need to update your packaging stack to pip >= 9.0 and setuptools >= 24.2 - see here for details. Then pip uninstall hypothesis && pip install hypothesis will get you back to a compatible version.

Strategies

- integers() bounds must be equal to an integer, though they can still be other types.
- If fractions() is passed a max_denominator, the bounds must have at most that denominator.
- floats() bounds must be exactly representable as a floating-point number with the given width. If not, the error message includes the nearest such number.
- sampled_from([]) is now an error.
- The values from the elements and fill strategies for hypothesis.extra.numpy.arrays() must be losslessly representable in an array of the given dtype.
- The min_size and max_size arguments to all collection strategies must be of type int (or max_size may be None).

Miscellaneous

- The .example() method of strategies (intended for interactive exploration) no longer takes a random argument.
- It is now an error to apply @example, @seed, or @reproduce_failure without also applying @given.
- You may pass either the target or targets argument to stateful rules, but not both.
- deadline must be None (to disable), a timedelta, or an integer or float number of milliseconds.
- Both of derandomize and print_blob must be either True or False, where they previously accepted other values.
- stateful_step_count must be at least one.
- max_examples must be at least one. To disable example generation, use the phases setting.
Removals

• `hypothesis.stateful.GenericStateMachine` in favor of `hypothesis.stateful.RuleBasedStateMachine`
• `hypothesis.extra.django.models.models` in favor of `hypothesis.extra.django.from_model()` and `hypothesis.extra.django.models.add_default_field_mapping` in favor of `hypothesis.extra.django.register_field_strategy()`
• `hypothesis.HealthCheck.hung_test`, without replacement
• `hypothesis.settings.buffer`, without replacement
• `hypothesis.PrintSettings`, because `hypothesis.settings.print_blob` takes True or False
• `hypothesis.settings.timeout`, in favor of `hypothesis.settings.deadline`
• `hypothesis.unlimited` without replacement (only only useful as argument to `timeout`)

19.3 Hypothesis 4.x

19.3.1 4.57.1 - 2019-12-29

This patch improves the type hints and documentation for the `django extra`. There is no runtime change.

19.3.2 4.57.0 - 2019-12-28

This release improves support for the SupportsOp protocols from the `typing` module when using on `from_type()` as outlined in issue #2292. The following types now generate much more varied strategies when called with `from_type()`:

• `typing.SupportsAbs`
• `typing.SupportsBytes`
• `typing.SupportsComplex`
• `typing.SupportsInt`
• `typing.SupportsFloat`
• `typing.SupportsRound`

Note that using `from_type()` with one of the above strategies will not ensure that the the specified function will execute successfully (ie : the strategy returned for `from_type(typing.SupportsAbs)` may include NaNs or things which cause the `abs()` function to error.)

Thanks to Lea Provenzano for this patch.
19.3.3 4.56.3 - 2019-12-22

This release fixes a small internal bug in shrinking which could have caused it to perform slightly more tests than were necessary. Fixing this shouldn't have much effect but it will make shrinking slightly faster.

19.3.4 4.56.2 - 2019-12-21

This release removes an internal heuristic that was no longer providing much benefit. It is unlikely that there will be any user visible effect.

19.3.5 4.56.1 - 2019-12-19

This release further improves the optimisation algorithm for targeted property-based testing.

19.3.6 4.56.0 - 2019-12-18

This release enables deprecation warnings even when the verbosity setting is quiet, in preparation for Hypothesis 5.0 (issue #2218).

Warnings can still be filtered by the standard mechanisms provided in the standard-library warnings module.

19.3.7 4.55.4 - 2019-12-18

This release improves Hypothesis’s management of the set of test cases it tracks between runs. It will only do anything if you have Phase.target enabled and an example database set. In those circumstances it should result in a more thorough and faster set of examples that are tried on each run.

19.3.8 4.55.3 - 2019-12-18

This release makes Hypothesis better at generating test cases where generated values are duplicated in different parts of the test case. This will be especially noticeable with reasonably complex values, as it was already able to do this for simpler ones such as integers or floats.

19.3.9 4.55.2 - 2019-12-17

This release expands the set of test cases that Hypothesis saves in its database for future runs to include a representative set of “structurally different” test cases - e.g. it might try to save test cases where a given list is empty or not.

Currently this is unlikely to have much user visible impact except to produce slightly more consistent behaviour between consecutive runs of a test suite. It is mostly groundwork for future improvements which will exploit this functionality more effectively.
19.3.10 4.55.1 - 2019-12-16

This patch fixes issue #2257, where `from_type()` could incorrectly generate bytestrings when passed a generic `typing.Sequence` such as `Sequence[set]`.

19.3.11 4.55.0 - 2019-12-16

This release adds database support for targeted property-based testing, so the best examples based on the targeting will be saved and reused between runs. This is mostly laying groundwork for future features in this area, but will also make targeted property-based tests more useful during development, where the same tests tend to get run over and over again. If `max_examples` is large, this may increase memory usage significantly under some circumstances, but these should be relatively rare.

This release also adds a dependency on the `sortedcontainers` package.

19.3.12 4.54.2 - 2019-12-16

This release improves the optimisation algorithm for targeted property-based testing, so that it will find higher quality results more reliably. Specifically, in cases where it would previously have got near a local optimum, it will now tend to achieve the locally optimal value.

19.3.13 4.54.1 - 2019-12-16

This release is mostly internal changes in support of better testing of the core engine. You are unlikely to see much effect, although some internal heuristics have changed slightly.

19.3.14 4.54.0 - 2019-12-15

This release adds a dedicated phase for targeted property-based testing, and (somewhat) improves the targeting algorithm so that it will find higher quality results more reliably. This comes at a cost of making it more likely to get stuck in a local optimum.

19.3.15 4.53.3 - 2019-12-15

This patch fixes `from_type()` with `typing.Hashable` and `typing.Sized`, which previously failed with an internal error on Python 3.7 or later.

Thanks to Lea Provenzano for both reporting issue #2272 and writing the patch!

19.3.16 4.53.2 - 2019-12-11

This release reorganises a number of the Hypothesis internal modules into a package structure. If you are only depending on the public API it should have no effect. If you are depending on the internal API (which you shouldn’t be, and which we don’t guarantee compatibility on) you may have to rename some imports.
19.3.17 4.53.1 - 2019-12-09

This release changes the size distribution of the number of steps run in stateful testing: It will now almost always run the maximum number of steps permitted.

19.3.18 4.53.0 - 2019-12-09

*Test statistics* now include the best score seen for each label, which can help avoid the threshold problem when the minimal example shrinks right down to the threshold of failure (issue #2180).

19.3.19 4.52.0 - 2019-12-09

This release changes the `stateful_step_count` setting to raise an error if set to 0. This is a backwards compatible change because a value of 0 would never have worked and attempting to run it would have resulted in an internal assertion error.

19.3.20 4.51.1 - 2019-12-09

This release makes a small internal change to the distribution of test cases. It is unlikely to have much user visible impact.

19.3.21 4.51.0 - 2019-12-07

This release deprecates use of `@example`, `@seed`, or `@reproduce_failure` without `@given`. Thanks to Nick Anyos for the patch!

19.3.22 4.50.8 - 2019-12-05

This patch makes certain uses of Bundles more efficient in stateful testing (issue #2078).

19.3.23 4.50.7 - 2019-12-05

This release refactors some of Hypothesis's internal interfaces for representing data generation. It should have no user visible effect.

19.3.24 4.50.6 - 2019-12-02

This patch removes some old debugging helpers in our Numpy extra which have not been needed since issue #1963 and issue #2245.
19.3.25 4.50.5 - 2019-12-01

This patch fixes issue #2229, where Numpy arrays of unsized strings would only ever have strings of size one due to an interaction between our generation logic and Numpy’s allocation strategy.

19.3.26 4.50.4 - 2019-12-01

This patch fixes a rare internal error in strategies for a list of unique items sampled from a short non-unique sequence (issue #2247). The bug was discovered via hypothesis-jsonschema.

19.3.27 4.50.3 - 2019-12-01

This release improves the error message when @settings tries to inherit settings from a parent argument that isn’t a settings instance.

19.3.28 4.50.2 - 2019-11-29

This release improves Hypothesis’s “Falsifying example” output, by breaking output across multiple lines where necessary, and by removing irrelevant information from the stateful testing output.

19.3.29 4.50.1 - 2019-11-29

This patch adds flake8-comprehensions to our linter suite. There is no user-visible change - expect perhaps via some strange microbenchmarks - but certain parts of the code now have a clear and more consistent style.

19.3.30 4.50.0 - 2019-11-28

This release fixes some cases where we might previously have failed to run the validation logic for some strategies. As a result tests which would previously have been silently testing significantly less than they should may now start to raise InvalidArgument now that these errors are caught.

19.3.31 4.49.0 - 2019-11-28

This release significantly improves the data distribution in rule based stateful testing, by using a technique called Swarm Testing (Groce, Alex, et al. “Swarm testing.” Proceedings of the 2012 International Symposium on Software Testing and Analysis. ACM, 2012.) to select which rules are run in any given test case. This should allow it to find many issues that it would previously have missed.

This change is likely to be especially beneficial for stateful tests with large numbers of rules.
19.3.32 4.48.1 - 2019-11-28

This release adds some heuristics to test case generation that try to ensure that test cases generated early on will be relatively small.

This fixes a bug introduced in Hypothesis 4.42.0 which would cause occasional too_slow failures on some tests.

19.3.33 4.48.0 - 2019-11-28

This release revokes the deprecation of find, as we’ve now rebuilt it on top of @given, which means it has minimal maintenance burden and we’re happy to support it.

19.3.34 4.47.5 - 2019-11-28

This release rebuilds find() on top of @given in order to have more code in common. It should have minimal user visible effect.

19.3.35 4.47.4 - 2019-11-27

This patch removes an internal compatibility shim that we no longer need.

19.3.36 4.47.3 - 2019-11-26

This patch fixes several typos in our docstrings and comments, with no change in behaviour. Thanks to Dmitry Dygalo for identifying and fixing them!

19.3.37 4.47.2 - 2019-11-25

This release fixes an internal issue where Hypothesis would sometimes generate test cases that were above its intended maximum size. This would only have happened rarely and probably would not have caused major problems when it did.

Users of the new targeted property-based testing might see minor impact (possibly slightly faster tests and slightly worse target scores), but only in the unlikely event that they were hitting this problem. Other users should not see any effect at all.

19.3.38 4.47.1 - 2019-11-24

This release removes some unused code from the core engine. There is no user-visible change.
19.3.39 4.47.0 - 2019-11-24

This release commonizes some code between running explicit examples and normal test execution. The main user visible impact of this is that deadlines are now enforced when running explicit examples.

19.3.40 4.46.1 - 2019-11-23

This patch ensures that a KeyboardInterrupt received during example generation is not treated as a mystery test failure but instead propagates to the top level, not recording the interrupted generation in the conjecture data tree. Thanks to Anne Archibald for identifying and fixing the problem.

19.3.41 4.46.0 - 2019-11-22

This release changes the behaviour of floats() when excluding signed zeros - floats(max_value=0.0, exclude_max=True) can no longer generate -0.0 nor the much rarer floats(min_value=-0.0, exclude_min=True) generate +0.0.

The correct interaction between signed zeros and exclusive endpoints was unclear; we now enforce the invariant that floats() will never generate a value equal to an excluded endpoint (issue #2201).

If you prefer the old behaviour, you can pass floats(max_value=-0.0) or floats(min_value=0.0) which is exactly equivalent and has not changed. If you had two endpoints equal to zero, we recommend clarifying your tests by using just() or sampled_from() instead of floats().

19.3.42 4.45.1 - 2019-11-20

This patch improves the error message when invalid arguments are passed to rule() or invariant() (issue #2149). Thanks to Benjamin Palmer for this bugfix!

19.3.43 4.45.0 - 2019-11-20

This release supports typing.Final and typing.TypedDict in from_type().

19.3.44 4.44.5 - 2019-11-20

This patch disables our pytest plugin when running on versions of pytest before 4.3, the oldest our plugin supports. Note that at time of writing the Pytest developers only support 4.6 and later!

Hypothesis tests using @given() work on any test runner, but our integrations to e.g. avoid example database collisions when combined with @pytest.mark.parametrize eventually drop support for obsolete versions.
19.3.45 4.44.4 - 2019-11-20

This patch adds some internal comments and clarifications to the Hypothesis implementation. There is no user-visible change.

19.3.46 4.44.3 - 2019-11-20

This patch avoids importing test runners such as pytest, unittest2, or nose solely to access their special “skip test” exception types - if the module is not in sys.modules, the exception can’t be raised anyway.

This fixes a problem where importing an otherwise unused module could cause spurious errors due to import-time side effects (and possibly -Werror).

19.3.47 4.44.2 - 2019-11-12

This release fixes @given to only complain about missing keyword-only arguments if the associated test function is actually called.

This matches the behaviour of other InvalidArgumentException errors produced by @given.

19.3.48 4.44.1 - 2019-11-11

This patch allows Hypothesis to run in environments that do not specify a __file__, such as a zipapp (issue #2196).

19.3.49 4.44.0 - 2019-11-11

This release adds a signature argument to mutually_broadcastable_shapes() (issue #2174), which allows us to generate shapes which are valid for functions like np.matmul() that require shapes which are not simply broadcastable.

Thanks to everyone who has contributed to this feature over the last year, and a particular shout-out to Zac Hatfield-Dodds and Ryan Soklaski for mutually_broadcastable_shapes() and to Ryan Turner for the downstream hypothesis-gufunc project.

19.3.50 4.43.9 - 2019-11-11

This patch fixes issue #2108, where the first test using data() to draw from characters() or text() would be flaky due to unreliable test timings.

Time taken by lazy instantiation of strategies is now counted towards drawing from the strategy, rather than towards the deadline for the test function.

19.3.51 4.43.8 - 2019-11-08

This release ensures that the strategies passed to @given are properly validated when applied to a test method inside a test class.

This should result in clearer error messages when some of those strategies are invalid.
19.3.52 4.43.7 - 2019-11-08

This release changes how Hypothesis manages its search space in cases where it generates redundant data. This should cause it to generate significantly fewer duplicated examples (especially with short integer ranges), and may cause it to produce more useful examples in some cases (especially ones where there is a significant amount of filtering).

19.3.53 4.43.6 - 2019-11-07

This patch refactors width handling in floats(); you may notice small performance improvements but the main purpose is to enable work on issue #1704 (improving shrinking of bounded floats).

19.3.54 4.43.5 - 2019-11-06

This patch removes an unused internal flag. There is no user-visible change.

19.3.55 4.43.4 - 2019-11-05

This patch corrects the exception type and error message you get if you attempt to use data() to draw from something which is not a strategy. This never worked, but the error is more helpful now.

19.3.56 4.43.3 - 2019-11-05

We’ve adopted flake8-bugbear to check for a few more style issues, and this patch implements the minor internal cleanups it suggested. There is no user-visible change.

19.3.57 4.43.2 - 2019-11-05

This patch fixes the formatting of some documentation, but there is no change to any executed code.

19.3.58 4.43.1 - 2019-11-04

Python 3.8’s new typing.Literal type - see PEP 586 for details - is now supported in from_type().

19.3.59 4.43.0 - 2019-11-04

This release adds the strategy mutually_broadcastable_shapes(), which generates multiple array shapes that are mutually broadcast-compatible with an optional user-specified base-shape.

This is a generalisation of broadcastable_shapes(). It relies heavily on non-public internals for performance when generating and shrinking examples. We intend to support generating shapes matching a ufunc signature in a future version (issue #2174).

Thanks to Ryan Soklaski, Zac Hatfield-Dodds, and @rdturnermtl who contributed to this new feature.
This release fixes `from_type()` when used with bounded or constrained `typing.TypeVar` objects (issue #2094). Previously, distinct typevars with the same constraints would be treated as all single typevar, and in cases where a typevar bound was resolved to a union of subclasses this could result in mixed types being generated for that typevar.

This patch ensures that the default value `broadcastable_shapes()` chooses for `max_dims` is always valid (at most 32), even if you pass `min_dims=32`.

This patch ensures that we only add profile information to the pytest header if running either pytest or Hypothesis in verbose mode, matching the `builtin cache plugin` (issue #2155).

This patch makes stateful step printing expand the result of a step into multiple variables when you return `multiple()` (issue #2139). Thanks to Joseph Weston for reporting and fixing this bug!

This release fixes a bug (issue #2166) where a Unicode character info cache file was generated but never used on subsequent test runs, causing tests to run more slowly than they should have. Thanks to Robert Knight for this bugfix!

This patch corrects some internal documentation. There is no user-visible change.

This release fixes a bug (issue #2160) where decorators applied after `@settings` and before `@given` were ignored. Thanks to Tom Milligan for this bugfix!

This release updates Hypothesis’s formatting to the new version of `black`, and has absolutely no user visible effect.
19.3.68 4.42.2 - 2019-10-30

This release fixes a bug in `recursive()` which would have meant that in practice `max_leaves` was treated as if it was lower than it actually is - specifically it would be capped at the largest power of two smaller than it. It is now handled correctly.

19.3.69 4.42.1 - 2019-10-30

Python 3.8’s new `typing.SupportsIndex` type - see PEP 357 for details - is now supported in `from_type()`.

Thanks to Grigorios Giannakopoulos for the patch!

19.3.70 4.42.0 - 2019-10-27

This release significantly simplifies Hypothesis’s internal logic for data generation, by removing a number of heuristics of questionable or unproven value.

The results of this change will vary significantly from test to test. Most test suites will see significantly faster data generation and lower memory usage. The “quality” of the generated data may go up or down depending on your particular test suites.

If you see any significant regressions in Hypothesis’s ability to find bugs in your code as a result of this release, please file an issue to let us know.

Users of the new targeted property-based testing functionality are reasonably likely to see improvements in data generation, as this release changes the search algorithm for targeted property based testing to one that is more likely to be productive than the existing approach.

19.3.71 4.41.3 - 2019-10-21

This patch is to ensure that our internals remain comprehensible to mypy 0.740 - there is no user-visible change.

19.3.72 4.41.2 - 2019-10-17

This patch changes some internal hashes to SHA384, to better support users subject to FIPS-140. There is no user-visible API change.

Thanks to Paul Kehrer for this contribution!

19.3.73 4.41.1 - 2019-10-16

This release makes `--hypothesis-show-statistics` much more useful for tests using a `RuleBasedStateMachine`, by simplifying the reprs so that events are aggregated correctly.
19.3.74 4.41.0 - 2019-10-16
This release upgrades the fixed_dictionaries() strategy to support optional keys (issue #1913).

19.3.75 4.40.2 - 2019-10-16
This release makes some minor internal changes in support of improving the Hypothesis test suite. It should not have any user visible impact.

19.3.76 4.40.1 - 2019-10-14
This release changes how Hypothesis checks if a parameter to a test function is a mock object. It is unlikely to have any noticeable effect, but may result in a small performance improvement, especially for test functions where a mock object is being passed as the first argument.

19.3.77 4.40.0 - 2019-10-09
This release fixes a bug where our example database logic did not distinguish between failing examples based on arguments from a @pytest.mark.parametrize(...). This could in theory cause data loss if a common failure overwrote a rare one, and in practice caused occasional file-access collisions in highly concurrent workloads (e.g. during a 300-way parametrize on 16 cores).
For internal reasons this also involves bumping the minimum supported version of pytest to 4.3
Thanks to Peter C Kroon for the Hacktoberfest patch!

19.3.78 4.39.3 - 2019-10-09
This patch improves our type hints on the emails(), functions(), integers(), iterables(), and slices() strategies, as well as the .filter() method.
There is no runtime change, but if you use mypy or a similar type-checker on your tests the results will be a bit more precise.

19.3.79 4.39.2 - 2019-10-09
This patch improves the performance of unique collections such as sets() of just() or booleans() strategies. They were already pretty good though, so you’re unlikely to notice much!

19.3.80 4.39.1 - 2019-10-09
If a value in a dict passed to fixed_dictionaries() is not a strategy, Hypothesis now tells you which one.
19.3.81 4.39.0 - 2019-10-07

This release adds the `basic_indices()` strategy, to generate basic indexes for arrays of the specified shape (issue #1930).

It generates tuples containing some mix of integers, `slice` objects, ... (Ellipsis), and `numpy.newaxis`; which when used to index an array of the specified shape produce either a scalar or a shared-memory view of the array. Note that the index tuple may be longer or shorter than the array shape, and may produce a view with another dimensionality again!

Thanks to Lampros Mountrakis, Ryan Soklaski, and Zac Hatfield-Dodds for their collaboration on this surprisingly subtle strategy!

19.3.82 4.38.3 - 2019-10-04

This patch defers creation of the `.hypothesis` directory until we have something to store in it, meaning that it will appear when Hypothesis is used rather than simply installed.

Thanks to Peter C Kroon for the Hacktoberfest patch!

19.3.83 4.38.2 - 2019-10-02

This patch bumps our dependency on `attrs` to >=19.2.0; but there are no user-visible changes to Hypothesis.

19.3.84 4.38.1 - 2019-10-01

This is a comment-only patch which tells `mypy` 0.730 to ignore some internal compatibility shims we use to support older Pythons.

19.3.85 4.38.0 - 2019-10-01

This release adds the `hypothesis.target()` function, which implements experimental support for targeted property-based testing (issue #1779).

By calling `target()` in your test function, Hypothesis can do a hill-climbing search for bugs. If you can calculate a suitable metric such as the load factor or length of a queue, this can help you find bugs with inputs that are highly improbably from unguided generation - however good our heuristics, example diversity, and deduplication logic might be. After all, those features are at work in targeted PBT too!

19.3.86 4.37.0 - 2019-09-28

This release emits a warning if you use the `.example()` method of a strategy in a non-interactive context. `.given()` is a much better choice for writing tests, whether you care about performance, minimal examples, reproducing failures, or even just the variety of inputs that will be tested!
19.3.87 4.36.2 - 2019-09-20

This patch disables part of the typing-based inference for the attrs package under Python 3.5.0, which has some incompatible internal details (issue #2095).

19.3.88 4.36.1 - 2019-09-17

This patch fixes a bug in strategy inference for attrs classes where Hypothesis would fail to infer a strategy for attributes of a generic type such as Union[int, str] or List[bool] (issue #2091).

Thanks to Jonathan Gayvallet for the bug report and this patch!

19.3.89 4.36.0 - 2019-09-09

This patch deprecates min_len or max_len of 0 in byte_string_dtypes() and unicode_string_dtypes(). The lower limit is now 1.

Numpy uses a length of 0 in these dtypes to indicate an undetermined size, chosen from the data at array creation. However, as the arrays() strategy creates arrays before filling them, strings were truncated to 1 byte.

19.3.90 4.35.1 - 2019-09-09

This patch improves the messaging that comes from invalid size arguments to collection strategies such as lists().

19.3.91 4.35.0 - 2019-09-04

This release improves the from_lark() strategy, tightening argument validation and adding the explicit argument to allow use with terminals that use @declare instead of a string or regular expression.

This feature is required to handle features such as indent and dedent tokens in Python code, which can be generated with the hypothesmith package.

19.3.92 4.34.0 - 2019-08-23

The from_type() strategy now knows to look up the subclasses of abstract types, which cannot be instantiated directly.

This is very useful for hypothesmith to support libCST.

19.3.93 4.33.1 - 2019-08-21

This patch works around a crash when an incompatible version of Numpy is installed under PyPy 5.10 (Python 2.7).

If you are still using Python 2, please upgrade to Python 3 as soon as possible - it will be unsupported at the end of this year.
19.3.94 4.33.0 - 2019-08-20

This release improves the `domains()` strategy, as well as the `urls()` and the `emails()` strategies which use it. These strategies now use the full IANA list of Top Level Domains and are correct as per RFC 1035.

Passing tests using these strategies may now fail.

Thanks to TechDragon for this improvement.

19.3.95 4.32.3 - 2019-08-05

This patch tidies up the repr of several settings-related objects, at runtime and in the documentation, and deprecates the undocumented edge case that `phases=None` was treated like `phases=tuple(Phase)`.

It also fixes `from_lark()` with lark 0.7.2 and later.

19.3.96 4.32.2 - 2019-07-30

This patch updates some internal comments for `mypy` 0.720. There is no user-visible impact.

19.3.97 4.32.1 - 2019-07-29

This release changes how the shrinker represents its progress internally. For large generated test cases this should result in significantly less memory usage and possibly faster shrinking. Small generated test cases may be slightly slower to shrink but this shouldn’t be very noticeable.

19.3.98 4.32.0 - 2019-07-28

This release makes `arrays()` more pedantic about `elements` strategies that cannot be exactly represented as array elements.

In practice, you will see new warnings if you were using a `float16` or `float32` dtype without passing `floats()` the `width=16` or `width=32` arguments respectively.

The previous behaviour could lead to silent truncation, and thus some elements being equal to an explicitly excluded bound (issue #1899).


This patch changes an internal use of MD5 to SHA hashes, to better support users subject to FIPS-140. There is no user-visible or API change.

Thanks to Alex Gaynor for this patch.
19.3.100 4.31.0 - 2019-07-24

This release simplifies the logic of the print_blob setting by removing the option to set it to PrintSettings.INFER. As a result the print_blob setting now takes a single boolean value, and the use of PrintSettings is deprecated.

19.3.101 4.28.2 - 2019-07-14

This patch improves the docstrings of several Hypothesis strategies, by clarifying markup and adding cross-references. There is no runtime change.

Thanks to Elizabeth Williams and Serah Njambi Rono for their contributions at the SciPy 2019 sprints!

19.3.102 4.28.1 - 2019-07-12

This patch improves the behaviour of the text() strategy when passed an alphabet which is not a strategy. The value is now interpreted as whitelist_characters to characters() instead of a sequence for sampled_from(), which standardises the distribution of examples and the shrinking behaviour.

You can get the previous behaviour by using lists(sampled_from(alphabet)).map(''.map) instead.

19.3.103 4.28.0 - 2019-07-11

This release deprecates find(). The .example() method is a better replacement if you want an example, and for the rare occasions where you want the minimal example you can get it from @given.

@given has steadily outstripped find() in both features and performance over recent years, and as we do not have the resources to maintain and test both we think it is better to focus on just one.

19.3.104 4.27.0 - 2019-07-08

This release refactors the implementation of the .example() method, to more accurately represent the data which will be generated by @given.

As a result, calling s.example() on an empty strategy s (such as nothing()) now raises Unsatisfiable instead of the old NoExamples exception.

19.3.105 4.26.4 - 2019-07-07

This patch ensures that the Pandas extra will keep working when Python 3.8 removes abstract base classes from the top-level collections namespace. This also fixes the relevant warning in Python 3.7, but there is no other difference in behaviour and you do not need to do anything.
19.3.106 4.26.3 - 2019-07-05

This release fixes issue #2027, by changing the way Hypothesis tries to generate distinct examples to be more efficient. This may result in slightly different data distribution, and should improve generation performance in general, but should otherwise have minimal user impact.


This release fixes issue #1864, where some simple tests would perform very slowly, because they would run many times with each subsequent run being progressively slower. They will now stop after a more reasonable number of runs without hitting this problem.

Unless you are hitting exactly this issue, it is unlikely that this release will have any effect, but certain classes of custom generators that are currently very slow may become a bit faster, or start to trigger health check failures.


This release adds the strategy `integer_array_indices()`, which generates tuples of Numpy arrays that can be used for advanced indexing to select an array of a specified shape.


This release significantly improves the performance of drawing unique collections whose elements are drawn from `sampled_from()` strategies.

As a side effect, this detects an error condition that would previously have passed silently: When the `min_size` argument on a collection with distinct elements is greater than the number of elements being sampled, this will now raise an error.

19.3.110 4.25.1 - 2019-07-03

This release removes some defunct internal functionality that was only being used for testing. It should have no user visible impact.

19.3.111 4.25.0 - 2019-07-03

This release deprecates and disables the `buffer_size` setting, which should have been treated as a private implementation detail all along. We recommend simply deleting this settings argument.

19.3.112 4.24.6 - 2019-06-26

This patch makes `datetimes()` more efficient, as it now handles short months correctly by construction instead of filtering.
This patch improves the development experience by simplifying the tracebacks you will see when e.g. you have used the `.map(...)` method of a strategy and the mapped function raises an exception.

No new exceptions can be raised, nor existing exceptions change anything but their traceback. We’re simply using if-statements rather than exceptions for control flow in a certain part of the internals!

This patch fixes issue #2014, where our compatibility layer broke with version 3.7.4 of the typing module backport on PyPI.

This issue only affects Python 2. We remind users that Hypothesis, like many other packages, will drop Python 2 support on 2020-01-01 and already has several features that are only available on Python 3.

This patch improves the implementation of an internal wrapper on Python 3.8 beta1 (and will break on the alphas; but they’re not meant to be stable). On other versions, there is no change at all.

Thanks to Daniel Hahler for the patch, and Victor Stinner for his work on bpo-37032 that made it possible.

Deprecation messages for functions in `hypothesis.extra.django.models` now explicitly name the deprecated function to make it easier to track down usages. Thanks to Kristian Glass for this contribution!

This patch fixes issue #1999, a spurious bug raised when a `@st.composite` function was passed a keyword-only argument.

Thanks to Jim Nicholls for his fantastic bug report.

This release deprecates `GenericStateMachine`, in favor of `RuleBasedStateMachine`. Rule-based stateful testing is significantly faster, especially during shrinking.

If your use-case truly does not fit rule-based stateful testing, we recommend writing a custom test function which drives your specific control-flow using `data()`.
19.3.119 4.23.9 - 2019-05-28

This patch fixes a very rare example database issue with file permissions. When running a test that uses both `@given` and `pytest.mark.parametrize`, using `pytest-xdist` on Windows, with failing examples in the database, two attempts to read a file could overlap and we caught `FileNotFoundError` but not other `OSError`s.

19.3.120 4.23.8 - 2019-05-26

This patch has a minor cleanup of the internal engine. There is no user-visible impact.

19.3.121 4.23.7 - 2019-05-26

This patch clarifies some error messages when the test function signature is incompatible with the arguments to `@given`, especially when the `@settings()` decorator is also used (issue #1978).

19.3.122 4.23.6 - 2019-05-19

This release adds the `pyupgrade` fixer to our code style, for consistent use of dict and set literals and comprehensions.

19.3.123 4.23.5 - 2019-05-16

This release slightly simplifies a small part of the core engine. There is no user-visible change.

19.3.124 4.23.4 - 2019-05-09

Fixes a minor formatting issue the docstring of `from_type()`

19.3.125 4.23.3 - 2019-05-09

Adds a recipe to the docstring of `from_type()` that describes a means for drawing values for “everything except” a specified type. This recipe is especially useful for writing tests that perform input-type validation.

19.3.126 4.23.2 - 2019-05-08

This patch uses `autoflake` to remove some pointless `pass` statements, which improves our workflow but has no user-visible impact.
19.3.127 4.23.1 - 2019-05-08

This patch fixes an OverflowError in `from_type(xrange)` on Python 2.
It turns out that not only do the start and stop values have to fit in a C long, but so does stop - start. We now handle this even on 32bit platforms, but remind users that Python2 will not be supported after 2019 without specific funding.

19.3.128 4.23.0 - 2019-05-08

This release implements the `slices()` strategy, to generate slices of a length-size sequence.
Thanks to Daniel J. West for writing this patch at the PyCon 2019 sprints!

19.3.129 4.22.3 - 2019-05-07

This patch exposes `DataObject`, solely to support more precise type hints. Objects of this type are provided by `data()`, and can be used to draw examples from strategies intermixed with your test code.

19.3.130 4.22.2 - 2019-05-07

This patch fixes the very rare issue #1798 in `array_dtypes()`, which caused an internal error in our tests.

19.3.131 4.22.1 - 2019-05-07

This patch fixes a rare bug in `from_type(range)`.
Thanks to Zebulun Arendsee for fixing the bug at the PyCon 2019 Sprints.

19.3.132 4.22.0 - 2019-05-07

The `unique_by` argument to `lists` now accepts a tuple of callables such that every element of the generated list will be unique with respect to each callable in the tuple (issue #1916).
Thanks to Marco Sirabella for this feature at the PyCon 2019 sprints!

19.3.133 4.21.1 - 2019-05-06

This patch cleans up the internals of `one_of()`. You may see a slight change to the distribution of examples from this strategy but there is no change to the public API.
Thanks to Marco Sirabella for writing this patch at the PyCon 2019 sprints!

The `from_type()` strategy now supports `slice` objects.
Thanks to Charlie El. Awbery for writing this feature at the PyCon 2019 Mentored Sprints.

19.3.135 4.20.0 - 2019-05-05

This release improves the `array_shapes()` strategy, to choose an appropriate default for `max_side` based on the `min_side`, and `max_dims` based on the `min_dims`. An explicit error is raised for dimensions greater than 32, which are not supported by Numpy, as for other invalid combinations of arguments.
Thanks to Jenny Rouleau for writing this feature at the PyCon 2019 Mentored Sprints.

19.3.136 4.19.0 - 2019-05-05

The `from_type()` strategy now supports `range` objects (or `xrange` on Python 2).
Thanks to Katrina Durance for writing this feature at the PyCon 2019 Mentored Sprints.

19.3.137 4.18.3 - 2019-04-30

This release fixes a very rare edge case in the test-case mutator, which could cause an internal error with certain unusual tests.

19.3.138 4.18.2 - 2019-04-30

This patch makes Hypothesis compatible with the Python 3.8 alpha, which changed the representation of code objects to support positional-only arguments. Note however that Hypothesis does not (yet) support such functions as e.g. arguments to `builds()` or inputs to `@given`.
Thanks to Paul Ganssle for identifying and fixing this bug.

19.3.139 4.18.1 - 2019-04-29

This patch improves the performance of unique collections such as `sets()` when the elements are drawn from a `sampled_from()` strategy (issue #1115).

19.3.140 4.18.0 - 2019-04-24

This release adds the `functions()` strategy, which can be used to imitate your ‘real’ function for callbacks.
19.3.141 4.17.2 - 2019-04-19

This release refactors stateful rule selection to share the new machinery with `sampled_from()` instead of using the original independent implementation.

19.3.142 4.17.1 - 2019-04-16

This patch allows Hypothesis to try a few more examples after finding the first bug, in hopes of reporting multiple distinct bugs. The heuristics described in issue #847 ensure that we avoid wasting time on fruitless searches, while still surfacing each bug as soon as possible.

19.3.143 4.17.0 - 2019-04-16

This release adds the strategy `broadcastable_shapes()`, which generates array shapes that are broadcast-compatible with a provided shape.

19.3.144 4.16.0 - 2019-04-12

This release allows `register_type_strategy()` to be used with `typing.NewType` instances. This may be useful to e.g. provide only positive integers for `from_type(UserId)` with a `UserId = NewType('UserId', int)` type.

Thanks to PJCampi for suggesting and writing the patch!

19.3.145 4.15.0 - 2019-04-09

This release supports passing a `timedelta` as the `deadline` setting, so you no longer have to remember that the number is in milliseconds (issue #1900).

Thanks to Damon Francisco for this change!

19.3.146 4.14.7 - 2019-04-09

This patch makes the type annotations on `hypothesis.extra.dateutil` compatible with `mypy 0.700`.

19.3.147 4.14.6 - 2019-04-07

This release fixes a bug introduced in Hypothesis 4.14.3 that would sometimes cause `sampled_from(...).filter(...)` to encounter an internal assertion failure when there are three or fewer elements, and every element is rejected by the filter.

This patch takes the previous efficiency improvements to `sampled_from(...).filter(...)` strategies that reject most elements, and generalises them to also apply to `sampled_from(...).filter(...).filter(...)` and longer chains of filters.

19.3.149 4.14.4 - 2019-04-05

This release fixes a bug that prevented `random_module()` from correctly restoring the previous state of the random module.

The random state was instead being restored to a temporary deterministic state, which accidentally caused subsequent tests to see the same random values across multiple test runs.

19.3.150 4.14.3 - 2019-04-03

This patch adds an internal special case to make `sampled_from(...).filter(...)` much more efficient when the filter rejects most elements (issue #1885).

19.3.151 4.14.2 - 2019-03-31

This patch improves the error message if the function $f$ in `s.flatmap(f)` does not return a strategy.

Thanks to Kai Chen for this change!


This release modifies how Hypothesis selects operations to run during shrinking, by causing it to deprioritise previously useless classes of shrink until others have reached a fixed point.

This avoids certain pathological cases where the shrinker gets very close to finishing and then takes a very long time to finish the last small changes because it tries many useless shrinks for each useful one towards the end. It also should cause a more modest improvement (probably no more than about 30%) in shrinking performance for most tests.


This release blocks installation of Hypothesis on Python 3.4, which reached its end of life date on 2019-03-18.

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.

19.3.154 4.13.0 - 2019-03-19

This release makes it an explicit error to call `floats(min_value=inf, exclude_min=True)` or `floats(max_value=-inf, exclude_max=True)`, as there are no possible values that can be generated (issue #1859).

`floats(min_value=0.0, max_value=-0.0)` is now deprecated. While $0. \equiv -0.$ and we could thus generate either if comparing by value, violating the sequence ordering of floats is a special case we don’t want or need.
This release should significantly reduce the amount of memory that Hypothesis uses for representing large test cases, by storing information in a more compact representation and only unpacking it lazily when it is first needed.

This update adds the `report_multiple_bugs` setting, which you can use to disable multi-bug reporting and only raise whichever bug had the smallest minimal example. This is occasionally useful when using a debugger or tools that annotate tracebacks via introspection.

This change makes a tiny improvement to the core engine’s bookkeeping. There is no user-visible change.

This release changes some of Hypothesis’s internal shrinking behaviour in order to reduce memory usage and hopefully improve performance.

This release adds a micro-optimisation to how Hypothesis handles debug reporting internally. Hard to shrink test may see a slight performance improvement, but in most common scenarios it is unlikely to be noticeable.

This release removes some redundant code that was no longer needed but was still running a significant amount of computation and allocation on the hot path. This should result in a modest speed improvement for most tests, especially those with large test cases.

This release adds a micro-optimisation to how Hypothesis caches test cases. This will cause a small improvement in speed and memory usage for large test cases, but in most common scenarios it is unlikely to be noticeable.

This release removes some internal code that populates a field that is no longer used anywhere. This should result in some modest performance and speed improvements and no other user visible effects.
19.3.163 4.11.1 - 2019-03-13

This is a formatting-only patch, enabled by a new version of isort.

19.3.164 4.11.0 - 2019-03-12

This release deprecates `sampled_from()` with empty sequences. This returns `nothing()`, which gives a clear error if used directly... but simply vanishes if combined with another strategy.

Tests that silently generate less than expected are a serious problem for anyone relying on them to find bugs, and we think reliability more important than convenience in this case.

19.3.165 4.10.0 - 2019-03-11

This release improves Hypothesis’s to detect flaky tests, by noticing when the behaviour of the test changes between runs. In particular this will notice many new cases where data generation depends on external state (e.g. external sources of randomness) and flag those as flaky sooner and more reliably.

The basis of this feature is a considerable reengineering of how Hypothesis stores its history of test cases, so on top of this its memory usage should be considerably reduced.

19.3.166 4.9.0 - 2019-03-09

This release adds the strategy `valid_tuple_axes()`, which generates tuples of axis-indices that can be passed to the `axis` argument in NumPy’s sequential functions (e.g. `numpy.sum()`).

Thanks to Ryan Soklaski for this strategy.

19.3.167 4.8.0 - 2019-03-06

This release significantly tightens validation in `hypothesis.settings`. `max_examples`, `buffer_size`, and `stateful_step_count` must be positive integers; `deadline` must be a positive number or `None`; and `derandomize` must be either `True` or `False`.

As usual, this replaces existing errors with a more helpful error and starts new validation checks as deprecation warnings.

19.3.168 4.7.19 - 2019-03-04

This release makes some micro-optimisations to certain calculations performed in the shrinker. These should particularly speed up large test cases where the shrinker makes many small changes. It will also reduce the amount allocated, but most of this is garbage that would have been immediately thrown away, so you probably won’t see much effect specifically from that.
19.3.169 4.7.18 - 2019-03-03

This patch removes some overhead from `arrays()` with a constant shape and dtype. The resulting performance improvement is modest, but worthwhile for small arrays.

19.3.170 4.7.17 - 2019-03-01

This release makes some micro-optimisations within Hypothesis’s internal representation of test cases. This should cause heavily nested test cases to allocate less during generation and shrinking, which should speed things up slightly.

19.3.171 4.7.16 - 2019-02-28

This changes the order in which Hypothesis runs certain operations during shrinking. This should significantly decrease memory usage and speed up shrinking of large examples.

19.3.172 4.7.15 - 2019-02-28

This release allows Hypothesis to calculate a number of attributes of generated test cases lazily. This should significantly reduce memory usage and modestly improve performance, especially for large test cases.

19.3.173 4.7.14 - 2019-02-28

This release reduces the number of operations the shrinker will try when reordering parts of a test case. This should in some circumstances significantly speed up shrinking. It may result in different final test cases, and if so usually slightly worse ones, but it should not generally have much impact on the end result as the operations removed were typically useless.

19.3.174 4.7.13 - 2019-02-27

This release changes how Hypothesis reorders examples within a test case during shrinking. This should make shrinking considerably faster.

19.3.175 4.7.12 - 2019-02-27

This release slightly improves the shrinker’s ability to replace parts of a test case with their minimal version, by allowing it to do so in bulk rather than one at a time. Where this is effective, shrinker performance should be modestly improved.

19.3.176 4.7.11 - 2019-02-25

This release makes some micro-optimisations to common operations performed during shrinking. Shrinking should now be slightly faster, especially for large examples with relatively fast test functions.
19.3.177 4.7.10 - 2019-02-25
This release is a purely internal refactoring of Hypothesis's API for representing test cases. There should be no user visible effect.

19.3.178 4.7.9 - 2019-02-24
This release changes certain shrink passes to make them more efficient when they aren’t making progress.

19.3.179 4.7.8 - 2019-02-23
This patch removes some unused code, which makes the internals a bit easier to understand. There is no user-visible impact.

19.3.180 4.7.7 - 2019-02-23
This release reduces the number of operations the shrinker will try when reordering parts of a test case. This should in some circumstances significantly speed up shrinking. It may result in different final test cases, and if so usually slightly worse ones, but it should not generally have much impact on the end result as the operations removed were typically useless.

19.3.181 4.7.6 - 2019-02-23
This patch removes some unused code from the shrinker. There is no user-visible change.

19.3.182 4.7.5 - 2019-02-23
This release changes certain shrink passes to make them adaptive - that is, in cases where they are successfully making progress they may now do so significantly faster.

19.3.183 4.7.4 - 2019-02-22
This is a docs-only patch, noting that because the lark-parser is under active development at version 0.x, hypothesis[lark] APIs may break in minor releases if necessary to keep up with the upstream package.

19.3.184 4.7.3 - 2019-02-22
This changes Hypothesis to no longer import various test frameworks by default (if they are installed), which will speed up the initial import hypothesis call.
19.3.185 4.7.2 - 2019-02-22

This release changes Hypothesis’s internal representation of a test case to calculate some expensive structural information on demand rather than eagerly. This should reduce memory usage a fair bit, and may make generation somewhat faster.

19.3.186 4.7.1 - 2019-02-21

This release refactors the internal representation of previously run test cases. The main thing you should see as a result is that Hypothesis becomes somewhat less memory hungry.

19.3.187 4.7.0 - 2019-02-21

This patch allows `array_shapes()` to generate shapes with side-length or even dimension zero, though the minimum still defaults to one. These shapes are rare and have some odd behavior, but are particularly important to test for just that reason!

In a related bigfix, `arrays()` now supports generating zero-dimensional arrays with `dtype=object` and a strategy for iterable elements. Previously, the array element would incorrectly be set to the first item in the generated iterable.

Thanks to Ryan Turner for continuing to improve our Numpy support.

19.3.188 4.6.1 - 2019-02-19

This release is a trivial micro-optimisation inside Hypothesis which should result in it using significantly less memory.

19.3.189 4.6.0 - 2019-02-18

This release changes some inconsistent behavior of `arrays()` from the Numpy extra when asked for an array of `shape=()`. `arrays()` will now always return a Numpy `ndarray`, and the array will always be of the requested `dtype`.

Thanks to Ryan Turner for this change.

19.3.190 4.5.12 - 2019-02-18

This release fixes a minor typo in an internal comment. There is no user-visible change.

19.3.191 4.5.11 - 2019-02-15

This release fixes issue #1813, a bug introduced in 3.59.1, which caused `random_module()` to no longer affect the body of the test: Although Hypothesis would claim to be seeding the random module in fact tests would always run with a seed of zero.
19.3.192 4.5.10 - 2019-02-14
This patch fixes an off-by-one error in the maximum length of `emails()`. Thanks to Krzysztof Jurewicz for pull request #1812.

19.3.193 4.5.9 - 2019-02-14
This patch removes some unused code from the shrinker. There is no user-visible change.

19.3.194 4.5.8 - 2019-02-12
This release fixes an internal `IndexError` in Hypothesis that could sometimes be triggered during shrinking.

19.3.195 4.5.7 - 2019-02-11
This release modifies the shrinker to interleave different types of reduction operations, e.g. switching between deleting data and lowering scalar values rather than trying entirely deletions then entirely lowering.
This may slow things down somewhat in the typical case, but has the major advantage that many previously difficult to shrink examples should become much faster, because the shrinker will no longer tend to stall when trying some ineffective changes to the shrink target but will instead interleave it with other more effective operations.

19.3.196 4.5.6 - 2019-02-11
This release makes a number of internal changes to the implementation of `hypothesis.extra.lark.from_lark()`. These are primarily intended as a refactoring, but you may see some minor improvements to performance when generating large strings, and possibly to shrink quality.

19.3.197 4.5.5 - 2019-02-10
This patch prints an explanatory note when issue #1798 is triggered, because the error message from Numpy is too terse to locate the problem.

19.3.198 4.5.4 - 2019-02-08
In Python 2, `long` integers are not allowed in the shape argument to `arrays()`. Thanks to Ryan Turner for fixing this.

19.3.199 4.5.3 - 2019-02-08
This release makes a small internal refactoring to clarify how Hypothesis instructs tests to stop running when appropriate. There is no user-visible change.
19.3.200 4.5.2 - 2019-02-06

This release standardises all of the shrinker’s internal operations on running in a random order.
The main effect you will see from this that it should now be much less common for the shrinker to stall for a long time
before making further progress. In some cases this will correspond to shrinking more slowly, but on average it should
result in faster shrinking.

19.3.201 4.5.1 - 2019-02-05

This patch updates some docstrings, but has no runtime changes.

19.3.202 4.5.0 - 2019-02-03

This release adds `exclude_min` and `exclude_max` arguments to `floats()`, so that you can easily generate values
from open or half-open intervals (issue #1622).

19.3.203 4.4.6 - 2019-02-03

This patch fixes a bug where `from_regex()` could throw an internal error if the `re.IGNORECASE` flag was used (issue
#1786).

19.3.204 4.4.5 - 2019-02-02

This release removes two shrink passes that Hypothesis runs late in the process. These were very expensive when the
test function was slow and often didn’t do anything useful.
Shrinking should get faster for most failing tests. If you see any regression in example quality as a result of this release,
please let us know.

19.3.205 4.4.4 - 2019-02-02

This release modifies the way that Hypothesis deletes data during shrinking. It will primarily be noticeable for very
large examples, which should now shrink faster.
The shrinker is now also able to perform some deletions that it could not previously, but this is unlikely to be very
noticeable.

19.3.206 4.4.3 - 2019-01-25

This release fixes an open file leak that used to cause `ResourceWarnings`.
19.3.207 4.4.2 - 2019-01-24

This release changes Hypothesis’s internal approach to caching the results of executing test cases. The result should be that it is now significantly less memory hungry, especially when shrinking large test cases.

Some tests may get slower or faster depending on whether the new or old caching strategy was well suited to them, but any change in speed in either direction should be minor.

19.3.208 4.4.1 - 2019-01-24

This patch tightens up some of our internal heuristics to deal with shrinking floating point numbers, which will now run in fewer circumstances.

You are fairly unlikely to see much difference from this, but if you do you are likely to see shrinking become slightly faster and/or producing slightly worse results.

19.3.209 4.4.0 - 2019-01-24

This release adds the `from_form()` function, which allows automatic testing against Django forms. (issue #35)

Thanks to Paul Stiverson for this feature, which resolves our oldest open issue!

19.3.210 4.3.0 - 2019-01-24

This release deprecates `HealthCheck.hung_test` and disables the associated runtime check for tests that ran for more than five minutes. Such a check is redundant now that we enforce the `deadline` and `max_examples` setting, which can be adjusted independently.

19.3.211 4.2.0 - 2019-01-23

This release adds a new module, `hypothesis.extra.lark`, which you can use to generate strings matching a context-free grammar.

In this initial version, only lark-parser EBNF grammars are supported, by the new `hypothesis.extra.lark.from_lark()` function.

19.3.212 4.1.2 - 2019-01-23

This patch fixes a very rare overflow bug (issue #1748) which could raise an `InvalidArgument` error in `complex_numbers()` even though the arguments were valid.

19.3.213 4.1.1 - 2019-01-23

This release makes some improvements to internal code organisation and documentation and has no impact on behaviour.
This release adds `register_random()`, which registers `random.Random` instances or compatible objects to be seeded and reset by Hypothesis to ensure that test cases are deterministic.

We still recommend explicitly passing a `random.Random` instance from `randoms()` if possible, but registering a framework-global state for Hypothesis to manage is better than flaky tests!

This patch fixes issue #1387, where bounded `integers()` with a very large range would almost always generate very large numbers. Now, we usually use the same tuned distribution as unbounded `integers()`.

This release randomizes the order in which the shrinker tries some of its initial normalization operations. You are unlikely to see much difference as a result unless your generated examples are very large. In this case you may see some performance improvements in shrinking.

Welcome to the next major version of Hypothesis!

There are no new features here, as we release those in minor versions. Instead, 4.0 is a chance for us to remove deprecated features (many already converted into no-ops), and turn a variety of warnings into errors.

If you were running on the last version of Hypothesis 3.x without any Hypothesis deprecation warnings (or using private APIs), this will be a very boring upgrade. In fact, nothing will change for you at all. Per our deprecation policy, warnings added in the last six months (after 2018-07-05) have not been converted to errors.

**Removals**

- `hypothesis.extra.datetime` has been removed, replaced by the core date and time strategies.
- `hypothesis.extra.fakelfactory` has been removed, replaced by general expansion of Hypothesis’ strategies and the third-party ecosystem.
- The SQLite example database backend has been removed.

**Settings**

- The `deadline` is now enforced by default, rather than just emitting a warning when the default (200 milliseconds per test case) deadline is exceeded.
- The `database_file` setting has been removed; use `database`.
- The `perform_health_check` setting has been removed; use `suppress_health_check`.
- The `max_shrinks` setting has been removed; use `phases` to disable shrinking.
- The `min_satisfying_examples`, `max_iterations`, `strict`, `timeout`, and `use_coverage` settings have been removed without user-configurable replacements.
Strategies

- The elements argument is now required for collection strategies.
- The average_size argument was a no-op and has been removed.
- Date and time strategies now only accept min_value and max_value for bounds.
- builds() now requires that the thing to build is passed as the first positional argument.
- Alphabet validation for text() raises errors, not warnings, as does category validation for characters().
- The choices() strategy has been removed. Instead, you can use data() with sampled_from(), so choice(elements) becomes data.draw(sampled_from(elements)).
- The streaming() strategy has been removed. Instead, you can use data() and replace iterating over the stream with data.draw() calls.
- sampled_from() and permutations() raise errors instead of warnings if passed a collection that is not a sequence.

Miscellaneous

- Applying @given to a test function multiple times was really inefficient, and now it’s also an error.
- Using the .example() method of a strategy (intended for interactive exploration) within another strategy or a test function always weakened data generation and broke shrinking, and now it’s an error too.
- The HYPOTHESIS_DATABASE_FILE environment variable is no longer supported, as the database_file setting has been removed.
- The HYPOTHESIS_VERBOSITY_LEVEL environment variable is no longer supported. You can use the --hypothesis-verbosity pytest argument instead, or write your own setup code using the settings profile system to replace it.
- Using @seed or derandomize=True now forces database=None to ensure results are in fact reproducible. If database is not None, doing so also emits a HypothesisWarning.
- Unused exception types have been removed from hypothesis.errors; namely AbnormalExit, BadData, BadTemplateDraw, DefinitelyNoSuchExample, Timeout, and WrongFormat.

19.4 Hypothesis 3.x

19.4.1 3.88.3 - 2019-01-11

This changes the order that the shrinker tries certain operations in its “emergency” phase which runs late in the process. The new order should be better at avoiding long stalls where the shrinker is failing to make progress, which may be helpful if you have difficult to shrink test cases. However this will not be noticeable in the vast majority of use cases.
19.4.2 3.88.2 - 2019-01-11

This is a pure refactoring release that extracts some logic from the core Hypothesis engine into its own class and file. It should have no user visible impact.

19.4.3 3.88.1 - 2019-01-11

This patch fixes some markup in our documentation.

19.4.4 3.88.0 - 2019-01-10

Introduces `hypothesis.stateful.multiple()`, which allows rules in rule based state machines to send multiple results at once to their target Bundle, or none at all.

19.4.5 3.87.0 - 2019-01-10

This release contains a massive cleanup of the Hypothesis for Django extra:

- `hypothesis.extra.django.models.models()` is deprecated in favor of `hypothesis.extra.django.from_model()`.
- `hypothesis.extra.django.models.add_default_field_mapping()` is deprecated in favor of `hypothesis.extra.django.register_field_strategy()`.
- `from_model()` does not infer a strategy for nullable fields or fields with a default unless passed `infer`, like `builds().models.models()` would usually but not always infer, and a special `default_value` marker object was required to disable inference.

19.4.6 3.86.9 - 2019-01-09

This release improves some internal logic about when a test case in Hypothesis’s internal representation could lead to a valid test case. In some circumstances this can lead to a significant speed up during shrinking. It may have some minor negative impact on the quality of the final result due to certain shrink passes now having access to less information about test cases in some circumstances, but this should rarely matter.

19.4.7 3.86.8 - 2019-01-09

This release has no user visible changes but updates our URLs to use HTTPS.

19.4.8 3.86.7 - 2019-01-08

Hypothesis can now automatically generate values for Django models with a `URLfield`, thanks to a new provisional strategy for URLs (issue #1388).
19.4.9  3.86.6 - 2019-01-07

This release is a pure refactoring that extracts some internal code into its own file. It should have no user visible effect.

19.4.10  3.86.5 - 2019-01-06

This is a docs-only patch, which fixes some typos and removes a few hyperlinks for deprecated features.

19.4.11  3.86.4 - 2019-01-04

This release changes the order in which the shrinker tries to delete data. For large and slow tests this may significantly improve the performance of shrinking.

19.4.12  3.86.3 - 2019-01-04

This release fixes a bug where certain places Hypothesis internal errors could be raised during shrinking when a user exception occurred that suppressed an exception Hypothesis uses internally in its generation.

The two known ways to trigger this problem were:

• Errors raised in stateful tests’ teardown function.
• Errors raised in finally blocks that wrapped a call to `data.draw`.

These cases will now be handled correctly.

19.4.13  3.86.2 - 2019-01-04

This patch is a docs-only change to fix a broken hyperlink.

19.4.14  3.86.1 - 2019-01-04

This patch fixes issue #1732, where `integers()` would always return `long` values on Python 2.

19.4.15  3.86.0 - 2019-01-03

This release ensures that infinite numbers are never generated by `floats()` with `allow_infinity=False`, which could previously happen in some cases where one bound was also provided.

The trivially inconsistent `min_value=inf, allow_infinity=False` now raises an `InvalidArgumentError`, as does the inverse with `max_value`. You can still use `just(inf)` to generate `inf` without violating other constraints.
Happy new year everyone! This release has no user visible changes but updates our copyright headers to include 2019.

This release makes a small change to the way the shrinker works. You may see some improvements to speed of shrinking on especially large and hard to shrink examples, but most users are unlikely to see much difference.

This patch fixes issue #1700, where a line that contained a Unicode character before a lambda definition would cause an internal exception.

Introduces the `hypothesis.stateful.consumes()` function. When defining a rule in stateful testing, it can be used to mark bundles from which values should be consumed, i.e. removed after use in the rule. This has been proposed in issue #136.

Thanks to Jochen Müller for this long-awaited feature.

This patch makes a small internal change to fix an issue in Hypothesis’s own coverage tests (issue #1718). There is no user-visible change.

This patch refactors the `hypothesis.strategies` module, so that private names should no longer appear in tab-completion lists. We previously relied on `__all__` for this, but not all editors respect it.

This is a follow-up patch to ensure that the deprecation date is automatically recorded for any new deprecations. There is no user-visible effect.

This patch updates the Hypothesis pytest plugin to avoid a recently deprecated hook interface. There is no user-visible change.
19.4.24 3.84.2 - 2018-12-19

This patch fixes the internals for `integers()` with one bound. Values from this strategy now always shrink towards zero instead of towards the bound, and should shrink much more efficiently too. On Python 2, providing a bound incorrectly excluded `long` integers, which can now be generated.

19.4.25 3.84.1 - 2018-12-18

This patch adds information about when features were deprecated, but this is only recorded internally and has no user-visible effect.

19.4.26 3.84.0 - 2018-12-18

This release changes the stateful testing backend from `find()` to use `@given` (issue #1300). This doesn’t change how you create stateful tests, but does make them run more like other Hypothesis tests.

`@reproduce_failure` and `@seed` now work for stateful tests.

Stateful tests now respect the `deadline` and `suppress_health_check` settings, though they are disabled by default. You can enable them by using `@settings(...)` as a class decorator with whatever arguments you prefer.

19.4.27 3.83.2 - 2018-12-17

Hypothesis has adopted Black as our code formatter (issue #1686). There are no functional changes to the source, but it’s prettier!

19.4.28 3.83.1 - 2018-12-13

This patch increases the variety of examples generated by `from_type()`.

19.4.29 3.83.0 - 2018-12-12

Our pytest plugin now warns you when strategy functions have been collected as tests, which may happen when e.g. using the `@composite` decorator when you should be using `@given(st.data())` for inline draws. Such functions always pass when treated as tests, because the lazy creation of strategies mean that the function body is never actually executed!

19.4.30 3.82.6 - 2018-12-11

Hypothesis can now `show statistics` when running under `pytest-xdist`. Previously, statistics were only reported when all tests were run in a single process (issue #700).
19.4.31  3.82.5 - 2018-12-08

This patch fixes issue #1667, where passing bounds of Numpy dtype int64 to \texttt{integers()} could cause errors on Python 3 due to internal rounding.

19.4.32  3.82.4 - 2018-12-08

Hypothesis now seeds and resets the global state of \texttt{np.random} for each test case, to ensure that tests are reproducible. This matches and complements the existing handling of the \texttt{random} module - Numpy simply maintains an independent PRNG for performance reasons.

19.4.33  3.82.3 - 2018-12-08

This is a no-op release to add the new \texttt{Framework :: Hypothesis} trove classifier to \texttt{hypothesis} on PyPI. You can use it as a filter to find Hypothesis-related packages such as extensions as they add the tag over the coming weeks, or simply visit our curated list.

19.4.34  3.82.2 - 2018-12-08

The \textit{Hypothesis for Pandas extension} is now listed in \texttt{setup.py}, so you can \texttt{pip install hypothesis[pandas]}. Thanks to jmshi for this contribution.

19.4.35  3.82.1 - 2018-10-29

This patch fixes \texttt{from_type()} on Python 2 for classes where \texttt{cls.__init__} is \texttt{object.__init__}. Thanks to ccxcz for reporting issue #1656.

19.4.36  3.82.0 - 2018-10-29

The \texttt{alphabet} argument for \texttt{text()} now uses its default value of \texttt{characters(blacklist_categories=('Cs',))} directly, instead of hiding that behind \texttt{alphabet=None} and replacing it within the function. Passing \texttt{None} is therefore deprecated.

19.4.37  3.81.0 - 2018-10-27

\texttt{GenericStateMachine} and \texttt{RuleBasedStateMachine} now raise an explicit error when instances of \texttt{settings} are assigned to the classes’ \texttt{settings} attribute, which is a no-op (issue #1643). Instead assign to \texttt{SomeStateMachine}. \texttt{TestCase.settings}, or use \texttt{@settings(...) as a class decorator to handle this automatically.}
19.4.38 3.80.0 - 2018-10-25

Since version 3.68.0, arrays() checks that values drawn from the elements and fill strategies can be safely cast to the dtype of the array, and emits a warning otherwise.

This release expands the checks to cover overflow for finite complex64 elements and string truncation caused by too-long elements or trailing null characters (issue #1591).

19.4.39 3.79.4 - 2018-10-25

Tests using @given now shrink errors raised from pytest helper functions, instead of reporting the first example found. This was previously fixed in version 3.56.0, but only for stateful testing.

19.4.40 3.79.3 - 2018-10-23

Traceback elision is now disabled on Python 2, to avoid an import-time SyntaxError under Python < 2.7.9 (Python: bpo-21591, Hypothesis 3.79.2: issue #1648).

We encourage all users to upgrade to Python 3 before the end of 2019.

19.4.41 3.79.2 - 2018-10-23

This patch shortens tracebacks from Hypothesis, so you can see exactly happened in your code without having to skip over irrelevant details about our internals (issue #848).

In the example test (see pull request #1582), this reduces tracebacks from nine frames to just three - and for a test with multiple errors, from seven frames per error to just one!

If you do want to see the internal details, you can disable frame elision by setting verbosity to debug.

19.4.42 3.79.1 - 2018-10-22

The abstract number classes Number, Complex, Real, Rational, and Integral are now supported by the from_type() strategy. Previously, you would have to use register_type_strategy() before they could be resolved (issue #1636)

19.4.43 3.79.0 - 2018-10-18

This release adds a CLI flag for verbosity --hypothesis-verbosity to the Hypothesis pytest plugin, applied after loading the profile specified by --hypothesis-profile. Valid options are the names of verbosity settings, quiet, normal, verbose or debug.

Thanks to Bex Dunn for writing this patch at the PyCon Australia sprints!

The pytest header now correctly reports the current profile if --hypothesis-profile has been used.

Thanks to Mathieu Paturel for the contribution at the Canberra Python Hacktoberfest.
19.4.44 3.78.0 - 2018-10-16

This release has deprecated the generation of integers, floats and fractions when the conversion of the upper and/or lower bound is not 100% exact, e.g. when an integer gets passed a bound that is not a whole number. (issue #1625)
Thanks to Felix Grünewald for this patch during Hacktoberfest 2018.

19.4.45 3.77.0 - 2018-10-16

This minor release adds functionality to settings allowing it to be used as a decorator on RuleBasedStateMachine and GenericStateMachine.
Thanks to Tyler Nickerson for this feature in #hacktoberfest!

19.4.46 3.76.1 - 2018-10-16

This patch fixes some warnings added by recent releases of pydocstyle and mypy.

19.4.47 3.76.0 - 2018-10-11

This release deprecates using floats for min_size and max_size.
The type hint for average_size arguments has been changed from Optional[int] to None, because non-None values are always ignored and deprecated.

19.4.48 3.75.4 - 2018-10-10

This patch adds more internal comments to the core engine’s sequence-length shrinker. There should be no user-visible change.

19.4.49 3.75.3 - 2018-10-09

This patch adds additional comments to some of the core engine’s internal data structures. There is no user-visible change.

19.4.50 3.75.2 - 2018-10-09

This patch avoids caching a trivial case, fixing issue #493.

19.4.51 3.75.1 - 2018-10-09

This patch fixes a broken link in a docstring. Thanks to Benjamin Lee for this contribution!
19.4.52 3.75.0 - 2018-10-08
This release deprecates the use of min_size=None, setting the default min_size to 0 (issue #1618).

19.4.53 3.74.3 - 2018-10-08
This patch makes some small internal changes to comply with a new lint setting in the build. There should be no user-visible change.

19.4.54 3.74.2 - 2018-10-03
This patch fixes issue #1153, where time spent reifying a strategy was also counted in the time spent generating the first example. Strategies are now fully constructed and validated before the timer is started.

19.4.55 3.74.1 - 2018-10-03
This patch fixes some broken formatting and links in the documentation.

19.4.56 3.74.0 - 2018-10-01
This release checks that the value of the print_blob setting is a PrintSettings instance.
Being able to specify a boolean value was not intended, and is now deprecated. In addition, specifying True will now cause the blob to always be printed, instead of causing it to be suppressed.
Specifying any value that is not a PrintSettings or a boolean is now an error.

19.4.57 3.73.5 - 2018-10-01
Changes the documentation for hypothesis.strategies.datetimes, hypothesis.strategies.dates, hypothesis.strategies.times to use the new parameter names min_value and max_value instead of the deprecated names

19.4.58 3.73.4 - 2018-09-30
This patch ensures that Hypothesis deprecation warnings display the code that emitted them when you’re not running in -Werror mode (issue #652).

19.4.59 3.73.3 - 2018-09-27
Tracebacks involving @composite are now slightly shorter due to some internal refactoring.
19.4.60 3.73.2 - 2018-09-26

This patch fixes errors in the internal comments for one of the shrinker passes. There is no user-visible change.

19.4.61 3.73.1 - 2018-09-25

This patch substantially improves the distribution of data generated with `recursive()`, and fixes a rare internal error (issue #1502).

19.4.62 3.73.0 - 2018-09-24

This release adds the `fulfill()` function, which is designed for testing code that uses `dpcontracts` 0.4 or later for input validation. This provides some syntactic sugar around use of `assume()`, to automatically filter out and retry calls that cause a precondition check to fail (issue #1474).

19.4.63 3.72.0 - 2018-09-24

This release makes setting attributes of the `hypothesis.settings` class an explicit error. This has never had any effect, but could mislead users who confused it with the current settings `instance hypothesis.settings.default` (which is also immutable). You can change the global settings with `settings profiles`.

19.4.64 3.71.11 - 2018-09-24

This patch factors out some common code in the shrinker for iterating over pairs of data blocks. There should be no user-visible change.

19.4.65 3.71.10 - 2018-09-18

This patch allows `from_type()` to handle the empty tuple type, `typing.Tuple[()]`.

19.4.66 3.71.9 - 2018-09-17

This patch updates some internal comments for `mypy`. There is no user-visible effect, even for Mypy users.

19.4.67 3.71.8 - 2018-09-17

This patch fixes a rare bug that would cause a particular shrinker pass to raise an IndexError, if a shrink improvement changed the underlying data in an unexpected way.
19.4.68 3.71.7 - 2018-09-17

This release fixes the broken cross-references in our docs, and adds a CI check so we don’t add new ones.

19.4.69 3.71.6 - 2018-09-16

This patch fixes two bugs (issue #944 and issue #1521), where messages about @seed did not check the current verbosity setting, and the wrong settings were active while executing explicit examples.

19.4.70 3.71.5 - 2018-09-15

This patch fixes a DeprecationWarning added in Python 3.8 (issue #1576).
Thanks to tirkarthi for this contribution!

19.4.71 3.71.4 - 2018-09-14

This is a no-op release, which implements automatic DOI minting and code archival of Hypothesis via Zenodo. Thanks to CERN and the EU Horizon 2020 programme for providing this service!
Check our CITATION.cff file for details, or head right on over to doi.org/10.5281/zenodo.1412597

19.4.72 3.71.3 - 2018-09-10

This release adds the test name to some deprecation warnings, for easier debugging.
Thanks to Sanyam Khurana for the patch!

19.4.73 3.71.2 - 2018-09-10

This release makes Hypothesis’s memory usage substantially smaller for tests with many examples, by bounding the number of past examples it keeps around.

You will not see much difference unless you are running tests with max_examples set to well over 1000, but if you do have such tests then you should see memory usage mostly plateau where previously it would have grown linearly with time.

19.4.74 3.71.1 - 2018-09-09

This patch adds internal comments to some tree traversals in the core engine. There is no user-visible change.
This release deprecates the coverage-guided testing functionality, as it has proven brittle and does not really pull its weight.

We intend to replace it with something more useful in the future, but the feature in its current form does not seem to be worth the cost of using, and whatever replaces it will likely look very different.

This patch changes the behaviour of `reproduce_failure()` so that blobs are only printed in quiet mode when the `print_blob` setting is set to `ALWAYS`.

Thanks to Cameron McGill for writing this patch at the PyCon Australia sprints!

This patch removes some unnecessary code from the internals. There is no user-visible change.

This patch fixes an internal bug where a corrupted argument to `@reproduce_failure` could raise the wrong type of error. Thanks again to Pawel T. Jochym, who maintains Hypothesis on conda-forge and consistently provides excellent bug reports including issue #1558.

This patch updates hypothesis to report its version and settings when run with pytest. (issue #1223).

Thanks to Jack Massey for this feature.

This release adds a `fullmatch` argument to `from_regex()`. When `fullmatch=True`, the whole example will match the regex pattern as for `re.fullmatch()`.

Thanks to Jakub Nabaglo for writing this patch at the PyCon Australia sprints!

This release reverts the changes to logging handling in 3.69.11, which broke test that use the `pytest caplog` fixture internally because all logging was disabled (issue #1546).
19.4.82 3.69.11 - 2018-08-29

This patch will hide all logging messages produced by test cases before the final, minimal, failing test case (issue #356). Thanks to Gary Donovan for writing this patch at the PyCon Australia sprints!

19.4.83 3.69.10 - 2018-08-29

This patch fixes a bug that prevents coverage from reporting unexecuted Python files (issue #1085). Thanks to Gary Donovan for writing this patch at the PyCon Australia sprints!

19.4.84 3.69.9 - 2018-08-28

This patch improves the packaging of the Python package by adding LICENSE.txt to the sdist (issue #1311), clarifying the minimum supported versions of pytz and dateutil (issue #1383), and adds keywords to the metadata (issue #1520). Thanks to Graham Williamson for writing this patch at the PyCon Australia sprints!

19.4.85 3.69.8 - 2018-08-28

This is an internal change which replaces pickle with json to prevent possible security issues. Thanks to Vidya Rani D G for writing this patch at the PyCon Australia sprints!

19.4.86 3.69.7 - 2018-08-28

This patch ensures that note() prints the note for every test case when the verbosity setting is Verbosity.verbose. At normal verbosity it only prints from the final test case. Thanks to Tom McDermott for writing this patch at the PyCon Australia sprints!

19.4.87 3.69.6 - 2018-08-27

This patch improves the testing of some internal caching. It should have no user-visible effect.

19.4.88 3.69.5 - 2018-08-27

This change performs a small rename and refactoring in the core engine. There is no user-visible change.

19.4.89 3.69.4 - 2018-08-27

This change improves the core engine’s ability to avoid unnecessary work, by consulting its cache of previously-tryed inputs in more cases.
19.4.90 3.69.3 - 2018-08-27

This patch handles passing an empty `enum.Enum` to `from_type()` by returning `nothing()`, instead of raising an internal `AssertionError`.

Thanks to Paul Amazone for writing this patch at the PyCon Australia sprints!

19.4.91 3.69.2 - 2018-08-23

This patch fixes a small mistake in an internal comment. There is no user-visible change.

19.4.92 3.69.1 - 2018-08-21

This change fixes a small bug in how the core engine consults its cache of previously-tried inputs. There is unlikely to be any user-visible change.

19.4.93 3.69.0 - 2018-08-20

This release improves argument validation for stateful testing.

- If the target or targets of a `rule()` are invalid, we now raise a useful validation error rather than an internal exception.
- Passing both the `target` and `targets` arguments is deprecated - append the `target` bundle to the `targets` tuple of bundles instead.
- Passing the name of a Bundle rather than the Bundle itself is also deprecated.

19.4.94 3.68.3 - 2018-08-20

This is a docs-only patch, fixing some typos and formatting issues.

19.4.95 3.68.2 - 2018-08-19

This change fixes a small bug in how the core engine caches the results of previously-tried inputs. The effect is unlikely to be noticeable, but it might avoid unnecessary work in some cases.

19.4.96 3.68.1 - 2018-08-18

This patch documents the `from_dtype()` function, which infers a strategy for `numpy.dtypes`. This is used in `arrays()`, but can also be used directly when creating e.g. Pandas objects.
**19.4.97 3.68.0 - 2018-08-15**

`arrays()` now checks that integer and float values drawn from `elements` and `fill` strategies can be safely cast to the dtype of the array, and emits a warning otherwise (issue #1385).

Elements in the resulting array could previously violate constraints on the elements strategy due to floating-point overflow or truncation of integers to fit smaller types.

**19.4.98 3.67.1 - 2018-08-14**

This release contains a tiny refactoring of the internals. There is no user-visible change.

**19.4.99 3.67.0 - 2018-08-10**

This release adds a `width` argument to `floats()`, to generate lower-precision floating point numbers for e.g. Numpy arrays.

The generated examples are always instances of Python’s native `float` type, which is 64bit, but passing `width=32` will ensure that all values can be exactly represented as 32bit floats. This can be useful to avoid overflow (to +/- infinity), and for efficiency of generation and shrinking.

Half-precision floats (`width=16`) are also supported, but require Numpy if you are running Python 3.5 or earlier.

**19.4.100 3.66.33 - 2018-08-10**

This release fixes a bug in `floats()`, where setting `allow_infinity=False` and exactly one of `min_value` and `max_value` would allow infinite values to be generated.

**19.4.101 3.66.32 - 2018-08-09**

This release adds type hints to the `example()` and `seed()` decorators, and fixes the type hint on `register_type_strategy()`. The second argument to `register_type_strategy()` must either be a `SearchStrategy`, or a callable which takes a `type` and returns a `SearchStrategy`.

**19.4.102 3.66.31 - 2018-08-08**

Another set of changes designed to improve the performance of shrinking on large examples. In particular the shrinker should now spend considerably less time running useless shrinks.

**19.4.103 3.66.30 - 2018-08-06**

“Bug fixes and performance improvements”.

This release is a fairly major overhaul of the shrinker designed to improve its behaviour on large examples, especially around stateful testing. You should hopefully see shrinking become much faster, with little to no quality degradation (in some cases quality may even improve).
19.4.104 3.66.29 - 2018-08-05

This release fixes two very minor bugs in the core engine:

- it fixes a corner case that was missing in 3.66.28, which should cause shrinking to work slightly better.
- it fixes some logic for how shrinking interacts with the database that was causing Hypothesis to be insufficiently aggressive about clearing out old keys.

19.4.105 3.66.28 - 2018-08-05

This release improves how Hypothesis handles reducing the size of integers’ representation. This change should mostly be invisible as it’s purely about the underlying representation and not the generated value, but it may result in some improvements to shrink performance.

19.4.106 3.66.27 - 2018-08-05

This release changes the order in which Hypothesis chooses parts of the test case to shrink. For typical usage this should be a significant performance improvement on large examples. It is unlikely to have a major impact on example quality, but where it does change the result it should usually be an improvement.

19.4.107 3.66.26 - 2018-08-05

This release improves the debugging information that the shrinker emits about the operations it performs, giving better summary statistics about which passes resulted in test executions and whether they were successful.

19.4.108 3.66.25 - 2018-08-05

This release fixes several bugs that were introduced to the shrinker in 3.66.24 which would have caused it to behave significantly less well than advertised. With any luck you should actually see the promised benefits now.

19.4.109 3.66.24 - 2018-08-03

This release changes how Hypothesis deletes data when shrinking in order to better handle deletion of large numbers of contiguous sequences. Most tests should see little change, but this will hopefully provide a significant speed up for stateful testing.

19.4.110 3.66.23 - 2018-08-02

This release makes some internal changes to enable further improvements to the shrinker. You may see some changes in the final shrunk examples, but they are unlikely to be significant.
19.4.111 3.66.22 - 2018-08-01

This release adds some more internal caching to the shrinker. This should cause a significant speed up for shrinking, especially for stateful testing and large example sizes.

19.4.112 3.66.21 - 2018-08-01

This patch is for downstream packagers - our tests now pass under pytest 3.7.0 (released 2018-07-30). There are no changes to the source of Hypothesis itself.

19.4.113 3.66.20 - 2018-08-01

This release removes some functionality from the shrinker that was taking a considerable amount of time and does not appear to be useful any more due to a number of quality improvements in the shrinker.

You may see some degradation in shrink quality as a result of this, but mostly shrinking should just get much faster.

19.4.114 3.66.19 - 2018-08-01

This release slightly changes the format of some debugging information emitted during shrinking, and refactors some of the internal interfaces around that.

19.4.115 3.66.18 - 2018-07-31

This release is a very small internal refactoring which should have no user visible impact.

19.4.116 3.66.17 - 2018-07-31

This release fixes a bug that could cause an IndexError to be raised from inside Hypothesis during shrinking. It is likely that it was impossible to trigger this bug in practice - it was only made visible by some currently unreleased work.

19.4.117 3.66.16 - 2018-07-31

This release is a very small internal refactoring which should have no user visible impact.

19.4.118 3.66.15 - 2018-07-31

This release makes Hypothesis’s shrinking faster by removing some redundant work that it does when minimizing values in its internal representation.

This release expands the deprecation of timeout from 3.16.0 to also emit the deprecation warning in find or stateful testing.

19.4.120 3.66.13 - 2018-07-30

This release adds an additional shrink pass that is able to reduce the size of examples in some cases where the transformation is non-obvious. In particular this will improve the quality of some examples which would have regressed in 3.66.12.

19.4.121 3.66.12 - 2018-07-28

This release changes how we group data together for shrinking. It should result in improved shrinker performance, especially in stateful testing.

19.4.122 3.66.11 - 2018-07-28

This patch modifies how which rule to run is selected during rule based stateful testing. This should result in a slight performance increase during generation and a significant performance and quality improvement when shrinking. As a result of this change, some state machines which would previously have thrown an InvalidDefinition are no longer detected as invalid.

19.4.123 3.66.10 - 2018-07-28

This release weakens some minor functionality in the shrinker that had only modest benefit and made its behaviour much harder to reason about. This is unlikely to have much user visible effect, but it is possible that in some cases shrinking may get slightly slower. It is primarily to make it easier to work on the shrinker and pave the way for future work.

19.4.124 3.66.9 - 2018-07-26

This release improves the information that Hypothesis emits about its shrinking when verbosity is set to debug.

19.4.125 3.66.8 - 2018-07-24

This patch includes some minor fixes in the documentation, and updates the minimum version of pytest to 3.0 (released August 2016).
19.4.126 3.66.7 - 2018-07-24

This release fixes a bug where difficult to shrink tests could sometimes trigger an internal assertion error inside the shrinker.

19.4.127 3.66.6 - 2018-07-23

This patch ensures that Hypothesis fully supports Python 3.7, by upgrading \texttt{from\_type()} (issue \#1264) and fixing some minor issues in our test suite (issue \#1148).

19.4.128 3.66.5 - 2018-07-22

This patch fixes the online docs for various extras, by ensuring that their dependencies are installed on readthedocs.io (issue \#1326).

19.4.129 3.66.4 - 2018-07-20

This release improves the shrinker’s ability to reorder examples.

For example, consider the following test:

```python
import hypothesis.strategies as st
from hypothesis import given
@given(st.text(), st.text())
def test_non_equal(x, y):
    assert x != y
```

Previously this could have failed with either of x="", y="0" or x="0", y="". Now it should always fail with x="", y="0".

This will allow the shrinker to produce more consistent results, especially in cases where test cases contain some ordered collection whose actual order does not matter.

19.4.130 3.66.3 - 2018-07-20

This patch fixes inference in the \texttt{builds()} strategy with subtypes of \texttt{typing.NamedTuple}, where the \texttt{\_init\_} method is not useful for introspection. We now use the field types instead - thanks to James Uther for identifying this bug.

19.4.131 3.66.2 - 2018-07-19

This release improves the shrinker’s ability to handle situations where there is an additive constraint between two values.

For example, consider the following test:
@given(st.integers(), st.integers())
def test_does_not_exceed_100(m, n):
    assert m + n < 100

Previously this could have failed with almost any pair \((m, n)\) with \(0 \leq m \leq n\) and \(m + n = 100\). Now it should almost always fail with \(m=0, n=100\).

This is a relatively niche specialisation, but can be useful in situations where e.g. a bug is triggered by an integer overflow.

**19.4.132 3.66.1 - 2018-07-09**

This patch fixes a rare bug where an incorrect percentage drawtime could be displayed for a test, when the system clock was changed during a test running under Python 2 (we use `time.monotonic()` where it is available to avoid such problems). It also fixes a possible zero-division error that can occur when the underlying C library double-rounds an intermediate value in `math.fsum()` and gets the least significant bit wrong.

**19.4.133 3.66.0 - 2018-07-05**

This release improves validation of the `alphabet` argument to the `text()` strategy. The following misuses are now deprecated, and will be an error in a future version:

- passing an unordered collection (such as `set('abc')`), which violates invariants about shrinking and reproducibility
- passing an alphabet sequence with elements that are not strings
- passing an alphabet sequence with elements that are not of length one, which violates any size constraints that may apply

Thanks to Sushobhit for adding these warnings (issue #1329).

**19.4.134 3.65.3 - 2018-07-04**

This release fixes a mostly theoretical bug where certain usage of the internal API could trigger an assertion error inside Hypothesis. It is unlikely that this problem is even possible to trigger through the public API.

**19.4.135 3.65.2 - 2018-07-04**

This release fixes dependency information for coverage. Previously Hypothesis would allow installing `coverage` with any version, but it only works with `coverage 4.0` or later.

We now specify the correct metadata in our `setup.py`, so Hypothesis will only allow installation with compatible versions of `coverage`. 
This patch ensures that stateful tests which raise an error from a pytest helper still print the sequence of steps taken to reach that point (issue #1372). This reporting was previously broken because the helpers inherit directly from BaseException, and therefore require special handling to catch without breaking e.g. the use of ctrl-C to quit the test.

This release deprecates the max_shrinks setting in favor of an internal heuristic. If you need to avoid shrinking examples, use the phases setting instead. (issue #1235)

This release fixes a bug where an internal assertion error could sometimes be triggered while shrinking a failing test.

This patch fixes type-checking errors in our vendored pretty-printer, which were ignored by our mypy config but visible for anyone else (whoops). Thanks to Pi Delport for reporting issue #1359 so promptly.

This release adds an interface which can be used to insert a wrapper between the original test function and @given (issue #1257). This will be particularly useful for test runner extensions such as pytest-trio, but is not recommended for direct use by other users of Hypothesis.

This release adds a new mechanism to infer strategies for classes defined using attrs, based on the the type, converter, or validator of each attribute. This inference is now built in to builds() and from_type(). On Python 2, from_type() no longer generates instances of int when passed long, or vice-versa.

This release adds PEP 484 type hints to Hypothesis on a provisional basis, using the comment-based syntax for Python 2 compatibility. You can read more about our type hints here. It also adds the py.typed marker specified in PEP 561. After you pip install hypothesis, mypy 0.590 or later will therefore type-check your use of our public interface!
19.4.143 3.61.0 - 2018-06-24

This release deprecates the use of *settings* as a context manager, the use of which is somewhat ambiguous. Users should define settings with global state or with the *@settings(...)* decorator.

19.4.144 3.60.1 - 2018-06-20

Fixed a bug in generating an instance of a Django model from a strategy where the primary key is generated as part of the strategy. See details here.

Thanks to Tim Martin for this contribution.

19.4.145 3.60.0 - 2018-06-20

This release adds the *@initialize* decorator for stateful testing (originally discussed in issue #1216). All *@initialize* rules will be called once each in an arbitrary order before any normal rule is called.

19.4.146 3.59.3 - 2018-06-19

This is a no-op release to take into account some changes to the release process. It should have no user visible effect.

19.4.147 3.59.2 - 2018-06-18

This adds support for partially sorting examples which cannot be fully sorted. For example, [5, 4, 3, 2, 1, 0] with a constraint that the first element needs to be larger than the last becomes [1, 2, 3, 4, 5, 0].

Thanks to Luke for contributing.

19.4.148 3.59.1 - 2018-06-16

This patch uses *random.getstate()* and *random.setstate()* to restore the PRNG state after *@given* runs deterministic tests. Without restoring state, you might have noticed problems such as issue #1266. The fix also applies to stateful testing (issue #702).

19.4.149 3.59.0 - 2018-06-14

This release adds the *emails()* strategy, which generates unicode strings representing an email address.

Thanks to Sushobhit for moving this to the public API (issue #162).
19.4.150 3.58.1 - 2018-06-13

This improves the shrinker. It can now reorder examples: 3 1 2 becomes 1 2 3.
Thanks to Luke for contributing.

19.4.151 3.58.0 - 2018-06-13

This adds a new extra `timezones()` strategy that generates `dateutil` timezones.
Thanks to Conrad for contributing.

19.4.152 3.57.0 - 2018-05-20

Using an unordered collection with the `permutations()` strategy has been deprecated because the order in which e.g. a set shrinks is arbitrary. This may cause different results between runs.

19.4.153 3.56.10 - 2018-05-16

This release makes `hypothesis.settings.define_setting` a private method, which has the effect of hiding it from the documentation.

19.4.154 3.56.9 - 2018-05-11

This is another release with no functionality changes as part of changes to Hypothesis's new release tagging scheme.

19.4.155 3.56.8 - 2018-05-10

This is a release with no functionality changes that moves Hypothesis over to a new release tagging scheme.

19.4.156 3.56.7 - 2018-05-10

This release provides a performance improvement for most tests, but in particular users of `sampled_from()` who don't have numpy installed should see a significant performance improvement.

19.4.157 3.56.6 - 2018-05-09

This patch contains further internal work to support Mypy. There are no user-visible changes… yet.
19.4.158 3.56.5 - 2018-04-22
This patch contains some internal refactoring to run mypy in CI. There are no user-visible changes.

19.4.159 3.56.4 - 2018-04-21
This release involves some very minor internal clean up and should have no user visible effect at all.

19.4.160 3.56.3 - 2018-04-20
This release fixes a problem introduced in 3.56.0 where setting the hypothesis home directory (through currently undocumented means) would no longer result in the default database location living in the new home directory.

19.4.161 3.56.2 - 2018-04-20
This release fixes a problem introduced in 3.56.0 where setting max_examples to 1 would result in tests failing with Unsatisfiable. This problem could also occur in other harder to trigger circumstances (e.g. by setting it to a low value, having a hard to satisfy assumption, and disabling health checks).

19.4.162 3.56.1 - 2018-04-20
This release fixes a problem that was introduced in 3.56.0: Use of the HYPOTHESIS_VERBOSITY_LEVEL environment variable was, rather than deprecated, actually broken due to being read before various setup the deprecation path needed was done. It now works correctly (and emits a deprecation warning).

19.4.163 3.56.0 - 2018-04-17
This release deprecates several redundant or internally oriented settings, working towards an orthogonal set of configuration options that are widely useful without requiring any knowledge of our internals (issue #535).

- Deprecated settings that no longer have any effect are no longer shown in the __repr__ unless set to a non-default value.
- hypothesis.settings.perform_health_check is deprecated, as it duplicates suppress_health_check.
- hypothesis.settings.max_iterations is deprecated and disabled, because we can usually get better behaviour from an internal heuristic than a user-controlled setting.
- hypothesis.settings.min_satisfying_examples is deprecated and disabled, due to overlap with the filter_too_much healthcheck and poor interaction with max_examples.
- HYPOTHESIS_VERBOSITY_LEVEL is now deprecated. Set verbosity through the profile system instead.
- Examples tried by find() are now reported at debug verbosity level (as well as verbose level).
19.4.164 3.55.6 - 2018-04-14

This release fixes a somewhat obscure condition (issue #1230) under which you could occasionally see a failing test trigger an assertion error inside Hypothesis instead of failing normally.

19.4.165 3.55.5 - 2018-04-14

This patch fixes one possible cause of issue #966. When running Python 2 with hash randomisation, passing a bytes object to `random.seed()` would use `version=1`, which broke `derandomize` (because the seed depended on a randomised hash). If `derandomize` is still nondeterministic for you, please open an issue.

19.4.166 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.

19.4.167 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.

19.4.168 3.55.2 - 2018-04-11

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

19.4.169 3.55.1 - 2018-04-06

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

19.4.170 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.
19.4.171 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.

19.4.172 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.

19.4.173 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.

19.4.174 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.

19.4.175 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.

19.4.176 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.
19.4.177 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.

19.4.178 3.50.3 - 2018-03-24

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

19.4.179 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.

19.4.180 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).

19.4.181 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).
**19.4.182 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.

**19.4.183 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.

**19.4.184 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.

**19.4.185 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.

**19.4.186 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.
19.4.187 3.46.2 - 2018-03-01
This release removes an unnecessary branch from the code, and has no user-visible impact.

19.4.188 3.46.1 - 2018-03-01
This changes only the formatting of our docstrings and should have no user-visible effects.

19.4.189 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.

19.4.190 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.

19.4.191 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.

19.4.192 3.45.3 - 2018-02-23
This patch may improve example shrinking slightly for some strategies.

19.4.193 3.45.2 - 2018-02-18
This release makes our docstring style more consistent, thanks to flake8-docstrings. There are no user-visible changes.
19.4.194 3.45.1 - 2018-02-17

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

19.4.195 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.

19.4.196 3.44.26 - 2018-02-06

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

19.4.197 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.

19.4.198 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.

19.4.199 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.

19.4.200 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.
19.4.201 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).

19.4.202 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.

19.4.203 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.

19.4.204 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.

19.4.205 3.44.17 - 2018-01-15

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

19.4.206 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.
19.4.207 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.

19.4.208 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.

19.4.209 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.

19.4.210 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.

19.4.211 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.

19.4.212 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).

19.4.213 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).
19.4.214 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.

19.4.215 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.

19.4.216 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.

19.4.217 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.

19.4.218 3.44.4 - 2017-12-23

This release fixes issue #1041, which slowed tests by up to 6% due to broken caching.

19.4.219 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.

19.4.220 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.
19.4.221 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.

19.4.222 3.44.0 - 2017-12-17

This release adds a new feature: The `@reproduce_failure` decorator, 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.

19.4.223 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.

19.4.224 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.

19.4.225 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.

19.4.226 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.
19.4.227 3.42.0 - 2017-12-09
This release deprecates faker-extra, which was designed as a transition strategy but does not support example shrinking or coverage-guided discovery.

19.4.228 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.

19.4.229 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.

19.4.230 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.

19.4.231 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.

19.4.232 3.38.9 - 2017-11-29
This is a documentation release to improve the documentation of shrinking behaviour for Hypothesis’s strategies.
19.4.233 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.

19.4.234 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.

19.4.235 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.

19.4.236 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.

19.4.237 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.

19.4.238 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).

19.4.239 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.
19.4.240 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.

19.4.241 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.

19.4.242 3.37.0 - 2017-11-12

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

The following are now deprecated:

- Passing `HealthCheck.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 `HealthCheck.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.
19.4.243 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.

19.4.244 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.

19.4.245 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.

19.4.246 3.34.1 - 2017-11-02

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

19.4.247 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())`

19.4.248 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.
This release supports strategy inference for more Django field types - 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.

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.

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.

This changes the default value of the `use_coverage` setting 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.

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.
This release fixes a performance problem in tests where the use_coverage setting is 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 3.30.4.

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

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.

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.

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.

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.
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.

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.

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.

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

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.
19.4.265 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.

19.4.266 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.

19.4.267 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()`.

19.4.268 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.
19.4.269 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.

19.4.270 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.

19.4.271 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.

19.4.272 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.

19.4.273 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.

19.4.274 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.

19.4.275 3.24.2 - 2017-09-11

This release removes the (purely internal and no longer useful) `strategy_test_suite` function and the corresponding strategystests module.
**19.4.276 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.

**19.4.277 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).

**19.4.278 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.

**19.4.279 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.

**19.4.280 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.
19.4.281 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.

19.4.282 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 3.20.0. (See issue #822)

19.4.283 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.

19.4.284 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.

19.4.285 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.

19.4.286 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.

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.

19.4.288 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)`.

19.4.289 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.

19.4.290 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.

19.4.291 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.

19.4.292 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`. 
19.4.293 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)

19.4.294 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.

19.4.295 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.

19.4.296 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.

19.4.297 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()` contained overlapping elements, then an `InvalidArgument` exception would be raised.

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

19.4.298 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()`).
19.4.299 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.

19.4.300 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).

19.4.301 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.

19.4.302 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.
19.4.303 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.

19.4.304 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.

19.4.305 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.


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.


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.

19.4.308 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.
19.4.309 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.

19.4.310 3.11.6 - 2017-06-19

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

19.4.311 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.

19.4.312 3.11.4 - 2017-06-17

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

19.4.313 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)

19.4.314 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.
This is a minor ergonomics release. Tracebacks shown by pytest no longer include Hypothesis internals for test functions decorated with @given.

This is a feature release, adding datetime-related strategies to the core strategies. timezone() 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 timezone= 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.

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).

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.

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".
19.4.320 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).

19.4.321 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.Enum, as all enums have a reliable iteration order.

19.4.322 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.

19.4.323 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.


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

19.4.325 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.

19.4.326 3.7.4 - 2017-04-22

This patch fixes a bug in 3.7.3, where using @example and a pytest fixture in the same test could cause the test to fail to fill the arguments, and throw a TypeError.
19.4.327 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.

19.4.328 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.

19.4.329 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.

19.4.330 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).

**19.4.331 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.

**19.4.332 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.

**19.4.333 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).
19.4.334 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.

19.4.335 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).

19.4.336 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).
  
- `pytest` 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).

19.4.337 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.
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.

This release is entirely provided by Lucas Wiman:

Strategies constructed by the Django extra 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.

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

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.

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.

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.
19.4.344 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.

19.4.345 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.
- Add 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).

19.4.346 3.0.5 - 2016-02-25

- Fix a bug where Hypothesis would now error on pytest development versions.
19.4.347 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).

19.4.348 3.0.3 - 2016-02-23

- Fix version parsing of pytest to work with pytest 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.

19.4.349 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.

19.4.350 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.

19.4.351 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.

## 19.5 Older versions

### 19.5.1 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.

19.5.2 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.

19.5.3 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.
19.5.4 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.

19.5.5 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.

19.5.6 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.
19.5.7 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.

19.5.8 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 NotImplemented.
- 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.

19.5.9 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

19.5.10 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).
19.5.11 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__.

19.5.12 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
- pytest 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.
19.5.13 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.

19.5.14 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).

19.5.15 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).

19.5.16 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
19.5.17 1.11.0 - 2015-08-31

- text() with a non-string alphabet would have used the repr() of 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 contain no explicit positional arguments, only varargs (issue #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.

19.5.18 1.10.6 - 2015-08-26

Fix support for fixtures on Django 1.7.

19.5.19 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.

19.5.20 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.

19.5.21 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.
19.5.22 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 in-cluded recursive() strategies. This should take the case where you use one recursive() strategy as the base strategy of another from unussably 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.

19.5.23 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.

19.5.24 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.
19.5.25 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.

19.5.26 1.8.3 - 2015-07-20

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

19.5.27 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.

19.5.28 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.

19.5.29 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.

19.5. Older versions
19.5.30 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.

19.5.31 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.
19.5.32 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.

19.5.33 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.

19.5.34 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.

19.5.35 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.

19.5.36 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.

19.5.37 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.
19.5.38 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.

19.5.39 1.2.0 - 2015-04-15

Codename: Finders keepsers.

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.

19.5.40 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.
19.5.41 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.

19.5.42 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.

19.5.43 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.

19.5.44 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.

19.5.45 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.

### 19.5.46 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](https://github.com/hypothesis/hypothesis/issues/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.

### 19.5.47 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].

### 19.5.48 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)
19.5.49 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.

19.5.50 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.

**19.5.51 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.

**19.5.52 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.
19.5.53 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…

19.5.54 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.

19.5.55 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.

19.5.56 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.
19.5.57 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.

19.5.58 0.2.2 - 2015-01-08

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

19.5.59 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.

19.5.60 0.2.0 - 2015-01-07

- It’s aliive.
- 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

19.5. Older versions 255
19.5.61 0.1.4 - 2013-12-14

- Make verification runs time bounded with a configurable timeout

19.5.62 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

19.5.63 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.

19.5.64 0.1.1 - 2013-03-24

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

19.5.65 0.1.0 - 2013-03-23

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

19.5.66 0.0.5 - 2013-03-13

- No changes except trying to fix packaging

19.5.67 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
19.5.68 0.0.3 - 2013-03-13

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

19.5.69 0.0.2 - 2013-03-12

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

19.5.70 0.0.1 - 2013-03-10

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

19.5. Older versions 257
CHAPTER TWENTY

ONGOING HYPOTHESIS DEVELOPMENT

Hypothesis development is managed by David R. MacIver and Zac Hatfield-Dodds, respectively the first author and lead maintainer.

*However*, these roles don’t include unpaid feature development on Hypothesis. Our roles as leaders 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 these examples of how the process goes). This isn’t “patches welcome”, it’s “we will help you write a patch”.

20.1 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.
20.2 Project roadmap

Hypothesis does not have a long-term release plan. We respond to bug reports as they are made; new features are released as and when someone volunteers to write and maintain them.
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 the maintainers). 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.

22.1 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.

22.2 Dependencies

22.2.1 Python versions

Hypothesis is designed to work with a range of Python versions - we support all versions of CPython with upstream support. We also support the latest versions of PyPy for Python 3.

22.2.2 Other Python libraries

Hypothesis has mandatory dependencies on the following libraries:

- attrs
- sortedcontainers

Hypothesis has optional dependencies on the following libraries:

```python
extras_require = {
    "cli": ["click>=7.0", "black>=19.10b0", "rich>=9.0.0"],
    "codemods": ["libcst>=0.3.16"],
    "ghostwriter": ["black>=19.10b0"],
    "pytz": ["pytz>=2014.1"],
    "dateutil": ["python-dateutil>=1.4"],
}
```
(continues on next page)
The way this works when installing Hypothesis normally is that these features become available if the relevant library is installed.

Specifically for pytest, our plugin supports versions of pytest which have been out of upstream support for some time. Hypothesis tests can still be executed by even older versions of pytest - you just won't have the plugin to provide automatic marks, helpful usage warnings, and per-test statistics.

22.3 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 fairly recent tooling; check the tree/master/requirements/ directory for details.

The organisation of the tests is described in the hypothesis-python/tests/README.rst.

22.4 Examples

- arch linux
- fedora
- gentoo
One of the things that is often concerning for people using randomized testing is the question of how to reproduce failing test cases.

**Note:** It is better to think about the data Hypothesis generates as being *arbitrary*, rather than *random*. We deliberately generate any valid data that seems likely to cause errors, so you shouldn’t rely on any expected distribution of or relationships between generated data. You can read about “swarm testing” and “coverage guided fuzzing” if you’re interested, because you don’t need to know for Hypothesis!

Fortunately Hypothesis has a number of features to support reproducing test failures. 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.

### 23.1 Providing explicit examples

The simplest way to reproduce a failed test is to ask Hypothesis to run the failing example it printed. For example, if *Falsifying example: test(n=1)* was printed you can decorate `test` with `@example(n=1)`.

`@example` can also be used to ensure a specific example is *always* executed as a regression test or to cover some edge case - basically combining a Hypothesis test and a traditional parametrized test.

```python
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):
    pass
```

(continues on next page)
from unittest import TestCase

class TestThings(TestCase):
    @given(text())
    @example("Hello world")
    @example(x="Some very long string")
    def test_some_code(self, x):
        pass

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.

### 23.2 Reproducing a test run with @seed

```python
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`. For example, the following test function and `RuleBasedStateMachine` will each check the same examples each time they are executed, thanks to `@seed()`:

```python
@seed(1234)
@given(x=...)
def test(x):
    ...

@seed(6789)
class MyModel(RuleBasedStateMachine):
    ...
```

The seed will not be printed if you could simply use @example instead.
23.3 Reproducing an example 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_failure decorator.

```python
import 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:

```python
>>> from hypothesis import settings, given, PrintSettings
>>> import hypothesis.strategies as st
>>> @given(st.floats())
... @settings(print_blob=True)
... 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(...) 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).

By default these messages are not printed. If you want to see these you must set the `print_blob` setting to `True`. 
h
hypothesis.extra.cli, 45
hypothesis.extra.codemods, 45
hypothesis.extra.dateutil, 47
hypothesis.extra.dpcontracts, 46
hypothesis.extra.ghostwriter, 49
hypothesis.extra.lark, 46
hypothesis.extra.numpy, 57
hypothesis.extra.pandas, 62
hypothesis.extra.pytz, 47
hypothesis.provisional, 39
hypothesis.strategies, 29

X
xps, 66
INDEX

A
array_dtypes() (in module hypothesis.extra.numpy), 59
array_shapes() (in module hypothesis.extra.numpy), 58
array_shapes() (in module xps), 67
arrays() (in module hypothesis.extra.numpy), 57
arrays() (in module xps), 66
assume() (in module hypothesis), 11

B
basic_indices() (in module hypothesis.extra.numpy), 61
binary() (in module hypothesis.strategies), 29
binary_operation() (in module hypothesis.extra.ghostwriter), 51
boolean_dtypes() (in module hypothesis.extra.numpy), 59
boolean_dtypes() (in module xps), 67
booleans() (in module hypothesis.strategies), 29
broadcastable_shapes() (in module hypothesis.extra.numpy), 59
broadcastable_shapes() (in module xps), 68
builds() (in module hypothesis.strategies), 29
byte_string_dtypes() (in module hypothesis.extra.numpy), 59

c
characters() (in module hypothesis.strategies), 29
column (class in hypothesis.extra.pandas), 63
columns() (in module hypothesis.extra.pandas), 63
complex_number_dtypes() (in module hypothesis.extra.numpy), 59
complex_numbers() (in module hypothesis.strategies), 30
composite() (in module hypothesis.strategies), 30
consumes() (in module hypothesis.stateful), 80
currently_in_test_context() (in module hypothesis), 106

d
data() (in module hypothesis.strategies), 30
data_frames() (in module hypothesis.extra.pandas), 63
data_too_large (hypothesis.HealthCheck attribute), 71
database (hypothesis.settings attribute), 23
DataObject (class in hypothesis.strategies), 31
dates() (in module hypothesis.strategies), 31
datetime64_dtypes() (in module hypothesis.extra.numpy), 59
datetimes() (in module hypothesis.strategies), 31
deadline (hypothesis.settings attribute), 24
decimals() (in module hypothesis.strategies), 31
defered() (in module hypothesis.strategies), 31
delete() (hypothesis.database.ExampleDatabase method), 75
derandomize (hypothesis.settings attribute), 24
dictionaries() (in module hypothesis.strategies), 32
DirectoryBasedExampleDatabase (class in hypothesis.database), 73
domains() (in module hypothesis.provisional), 39
DrawFn (class in hypothesis.strategies), 32

e
emails() (in module hypothesis.strategies), 32
equivalent() (in module hypothesis.extra.ghostwriter), 51
event() (in module hypothesis), 10
example() (in module hypothesis), 265
ExampleDatabase (class in hypothesis.database), 75

F
fetch() (hypothesis.database.ExampleDatabase method), 75
filter_too_much (hypothesis.HealthCheck attribute), 71
fixed_dictionaries() (in module hypothesis.strategies), 32
floating_dtypes() (in module hypothesis.extra.numpy), 58
floating_dtypes() (in module xps), 67
floats() (in module hypothesis.strategies), 32
fractions() (in module hypothesis.strategies), 33
from_dtype() (in module hypothesis.extra.numpy), 57
from_dtype() (in module xps), 66
from_field() (in module hypothesis.extra.django), 55
from_form() (in module hypothesis.extra.django), 54
from_lark() (in module hypothesis.extra.lark), 47
from_model() (in module hypothesis.extra.django), 53
from_regex() (in module hypothesis.strategies), 33
from_type() (in module hypothesis.strategies), 33
frozensets() (in module hypothesis.strategies), 34
fulfill() (in module hypothesis.extra.dpcontracts), 46
function_scoped_fixture (hypothesis.HealthCheck attribute), 71
functions() (in module hypothesis.strategies), 34
fuzz() (in module hypothesis.extra.ghostwriter), 50

G
get_profile() (hypothesis.settings static method), 27
given() (in module hypothesis), 13

H
HealthCheck (class in hypothesis), 71
hypothesis.extra.cli module, 45
hypothesis.extra.codemods module, 45
hypothesis.extra.dateutil module, 47
hypothesis.extra.django.LiveServerTestCase (built-in class), 53
hypothesis.extra.django.StaticLiveServerTestCase (built-in class), 53
hypothesis.extra.django.TestCase (built-in class), 53
hypothesis.extra.django.TransactionTestCase (built-in class), 53
hypothesis.extra.dpcontracts module, 46
hypothesis.extra.ghostwriter module, 49
hypothesis.extra.lark module, 46
hypothesis.extra.numpy module, 57
hypothesis.extra.pandas module, 62
hypothesis.extra.pytz module, 47
hypothesis.infer (built-in variable), 18
hypothesis.Phase (built-in class), 25
hypothesis.provisional module, 39
hypothesis.strategies module, 29
hypothesis.strategies.SearchStrategy (built-in class), 19
HypothesisDeprecationWarning (class in hypothesis.errors), 72

I
idempotent() (in module hypothesis.extra.ghostwriter), 50
indexes() (in module hypothesis.extra.pandas), 62
indices() (in module xps), 69
initialize() (in module hypothesis.stateful), 81
InMemoryExampleDatabase (class in hypothesis.database), 73
integer_array_indices() (in module hypothesis.extra.numpy), 61
integer_dtypes() (in module hypothesis.extra.numpy), 58
integer_dtypes() (in module xps), 67
integers() (in module hypothesis.strategies), 34
invariant() (in module hypothesis.stateful), 82
ip_addresses() (in module hypothesis.stateful), 82
iterables() (in module hypothesis.strategies), 35

J
just() (in module hypothesis.strategies), 35

L
large_base_example (hypothesis.HealthCheck attribute), 71
lists() (in module hypothesis.strategies), 35
load_profile() (hypothesis.settings static method), 27

M
magic() (in module hypothesis.extra.ghostwriter), 50
make_strategies_namespace() (in module hypothesis.extra.array_api), 65
max_examples (hypothesis.settings attribute), 24
module
hypothesis.extra.cli, 45
hypothesis.extra.codemods, 45
hypothesis.extra.dateutil, 47
hypothesis.extra.dpcontracts, 46
hypothesis.extra.ghostwriter, 49
hypothesis.extra.lark, 46
hypothesis.extra.numpy, 57
hypothesis.extra.pandas, 62
hypothesis.extra.pytz, 47
hypothesis.provisional, 39
hypothesis.strategies, 29
xps, 66
move() (hypothesis.database.ExampleDatabase method), 75
multiple() (in module hypothesis.stateful), 80
MultiplexedDatabase (class in hypothesis.database), 74
mutually_broadcastable_shapes() (in module hypothesis.extra.numpy), 60
mutually_broadcastable_shapes() (in module xps), 68

N

nested_dtypes() (in module hypothesis.extra.numpy), 59

none() (in module hypothesis.strategies), 35

not_a_test_method (hypothesis.HealthCheck attribute), 71

note() (in module hypothesis), 9

nothing() (in module hypothesis.strategies), 35

numeric_dtypes() (in module xps), 67

O

one_of() (in module hypothesis.strategies), 35

P

permutations() (in module hypothesis.strategies), 36

phases (hypothesis.settings attribute), 24

precondition() (in module hypothesis.stateful), 81

print_blob (hypothesis.settings attribute), 24

Python Enhancement Proposals

PEP 20, 25

PEP 310, 122, 141

PEP 3107, 227, 231

PEP 357, 155

PEP 398, 216

PEP 429, 167

PEP 484, 18, 143, 197

PEP 484#forward-references, 208

PEP 495, 109

PEP 560, 113

PEP 561, 18, 197

PEP 565, 210

PEP 570, 141

PEP 585, 114, 126

PEP 586, 153

PEP 646, 114

R

random_module() (in module hypothesis.strategies), 36

randoms() (in module hypothesis.strategies), 36

range_indexes() (in module hypothesis.extra.pandas), 62

ReadOnlyDatabase (class in hypothesis.database), 74

recursive() (in module hypothesis.strategies), 36

RedisExampleDatabase (class in hypothesis.extra.redis), 74

refactor() (in module hypothesis.extra.codemods), 46

register_field_strategy() (in module hypothesis.extra.django), 55

register_profile() (hypothesis.settings static method), 27

register_random() (in module hypothesis), 17

gerister_type_strategy() (in module hypothesis.strategies), 36

report_multiple_bugs (hypothesis.settings attribute), 24

reproduce_failure() (in module hypothesis), 267

return_value (hypothesis.HealthCheck attribute), 71

RFC 1035, 39, 159

RFC 3986, 39

RFC 5234, 47

RFC 5322#section-3.4.1, 32

roundtrip() (in module hypothesis.extra.ghostwriter), 51

rule() (in module hypothesis.stateful), 80

RuleBasedStateMachine (class in hypothesis.stateful), 78

runner() (in module hypothesis.strategies), 37

S

sampled_from() (in module hypothesis.strategies), 37

save() (hypothesis.database.ExampleDatabase method), 75

scalar_dtypes() (in module hypothesis.extra.numpy), 58

scalar_dtypes() (in module xps), 67

seed() (in module hypothesis), 266

series() (in module hypothesis.extra.pandas), 63

sets() (in module hypothesis.strategies), 37

settings (class in hypothesis), 23

shared() (in module hypothesis.strategies), 37

slices() (in module hypothesis.strategies), 37

stateful_step_count (hypothesis.settings attribute), 24

suppress_health_check (hypothesis.settings attribute), 24

T

target() (in module hypothesis), 15

text() (in module hypothesis.strategies), 37

timedeltas() (in module hypothesis.strategies), 38

timedeltas() (in module hypothesis.extra.numpy), 59

timedelta64_dtypes() (in module hypothesis.extra.numpy), 59

timedelta64_dtypes() (in module hypothesis.extra.pandas), 63

times() (in module hypothesis.strategies), 38

timezone_keys() (in module hypothesis.strategies), 38

timezone_keys() (in module hypothesis.extra.django), 55

timezone_keys() (in module hypothesis.extra.pytz), 47

timezones() (in module hypothesis.strategies), 38

toSlow() (hypothesis.HealthCheck attribute), 71

tuples() (in module hypothesis.strategies), 39

U

ufunc() (in module hypothesis.extra.ghostwriter), 52

unicode_string_dtypes() (in module hypothesis.extra.numpy), 59
unsigned_integer_dtypes() (in module hypothesis.extra.numpy), 58
unsigned_integer_dtypes() (in module xps), 67
urls() (in module hypothesis.provisional), 39
uuids() (in module hypothesis.strategies), 39

valid_tuple_axes() (in module hypothesis.extra.numpy), 59
valid_tuple_axes() (in module xps), 67
verbosity (hypothesis.settings attribute), 25

x

xps
  module, 66