# Hypothesis: Throw away your assumptions to catch more bugs

Andy Clegg [andrew.clegg@gmx.com]

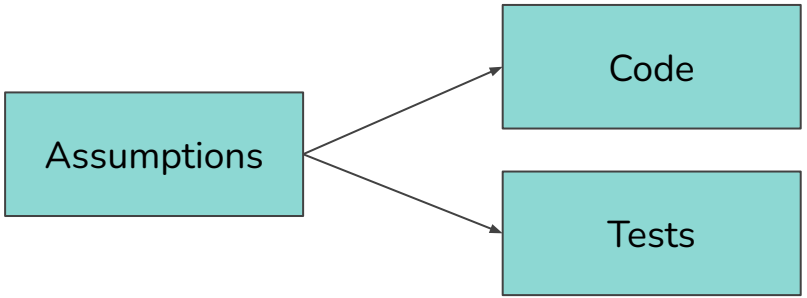
# What is Hypothesis?

- Python library for writing property-based tests
- Complementary to traditional unit tests
- Integrates nicely into Pytest and unittest

To understand Hypothesis, we're going to need to know:

- What problem are we trying to solve with it?
- What are these 'properties' I keep mentioning?
- How can we use them to test our code?

# The problem




- Often the same person writes the code and the test; the same assumptions are used in both
- If AI is writing the code, the assumptions might be based on the prompt (or worse!)
- Hand-picked scenarios take time to write
  - Small number of tests
  - Edge cases are missed

Alternatively:

- Describe the properties we expect our code to have
- Use an automated approach to test these properties

# Properties of code

"Some aspect of the output is always the same"

```
assert type(f(X)) == int
```

"This roundtrip operation preserves the original input"

```
assert load(save(X)) == X
```

"I trust another implementation", aka Oracle

```
assert my_func(X) == oracle(X)
```

https://fsharpforfunandprofit.com/posts/property-based-testing-2/

# base64

```
10110001110100110111001110001011001011 00 => sdNziyw=
```

```python
import string

base64_characters = string.ascii_uppercase + string.ascii_lowercase + string.digits + '+/'
assert len(base64_characters) == 64

def my_b64_encode(x):
    binary_8bit = ''.join(format(byte, '08b') for byte in x)
    binary_6bit_chunks = ['00' + binary_8bit[x:x+6] for x in range(0, len(binary_8bit), 6)]
    return  ''.join(base64_characters[int(x,2)] for x in binary_6bit_chunks).encode('ascii')


def my_b64_decode(x):
    data_points = [base64_characters.index(c) for c in x.decode('ascii')]
    binary_6bit = ''.join(format(a, '06b') for a in data_points)
    binary_8bit_chunks = [binary_6bit[x:x+8] for x in range(0, len(binary_6bit), 8)]
    return bytes(int(x,2) for x in binary_8bit_chunks)
```

https://base64.guru/learn/base64-algorithm/encode

# base64 – oracle & roundtrip

```python
from bad64 import my_b64_encode, my_b64_decode
from base64 import b64encode as stdlib_b64_encode

def test_known_example():
    assert my_b64_encode(b'ABC') == b'QUJD'

def test_roundtrip():
    assert my_b64_decode(my_b64_encode(b'XYZ')) == b'XYZ'

def test_oracle():
    assert my_b64_encode(b'123') == stdlib_b64_encode(b'123')
```

# base64 - oracle & roundtrip with hypothesis

```python
from bad64 import my_b64_encode, my_b64_decode
from base64 import b64encode as stdlib_b64_encode

from hypothesis import given, strategies

@given(strategies.binary())
def test_roundtrip(x):
    assert my_b64_decode(my_b64_encode(x)) == x

@given(strategies.binary())
def test_oracle(x):
    assert my_b64_encode(x) == stdlib_b64_encode(x)
```

```
>>> from hypothesis import strategies
>>> strategies.binary().example()
b'\x94dg'
>>> strategies.binary().example()
b"'\x9b"
>>> strategies.binary().example()
b'5'
```

# base64 - oracle & roundtrip with hypothesis

```
    @given(strategies.binary())
>   def test_oracle(x):

test_base64_with_hypothesis.py:11:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

x = b'\x00'

    @given(strategies.binary())
    def test_oracle(x):
>       assert my_b64_encode(x) == stdlib_b64_encode(x)
E       AssertionError: assert b'AA' == b'AA=='
E
E         Use -v to get more diff
E       Falsifying example: test_oracle(
E           x=b'\x00',
E       )

test_base64_with_hypothesis.py:12: AssertionError
=========================================================================================== short test
FAILED test_base64_with_hypothesis.py::test_roundtrip - AssertionError: assert b'\x00\x00' == b'\x00'
FAILED test_base64_with_hypothesis.py::test_oracle - AssertionError: assert b'AA' == b'AA=='
============================================================================================ 2 faile
```

# Hypothesis model

- Random data
  - 100 attempts per run
  - Non-deterministic, so each run is different
- Failed examples are stored in a local database, and are automatically re-run
- You can also provide specific examples to use

```
@example(b'ABC')
@given(st.binary())
def test_roundtrip(x):
    assert my_b64_decode(my_b64_encode(x)) == x
```

# Business logic example

```python
import dataclasses, json, os

@dataclasses.dataclass
class Account:
    name: str
    balance: int = 0

    def store(self, fn):
        with open(fn, 'at') as f:
            json.dump({"name": self.name, "balance": self.name}, f)

    @staticmethod
    def load(fn):
        with open(fn, 'rt') as f:
            data = json.load(f)
            return Account(data.get("name", ""), data.get("balance", 0))

    def transfer(self, other, update):
        # We don't allow a negative balance
        if self.balance >= update:
            self.balance -= update
            other.balance += update
```

# Conclusion

- Look for properties in your code
  - We've looked at a few classes of properties
  - There are many other ideas for testable properties
- Generate data for your tests
  - Start broad, only narrow the search when it is justified
- Add failing cases as `@example`s, and then fix the code
- Enjoy code with fewer bugs