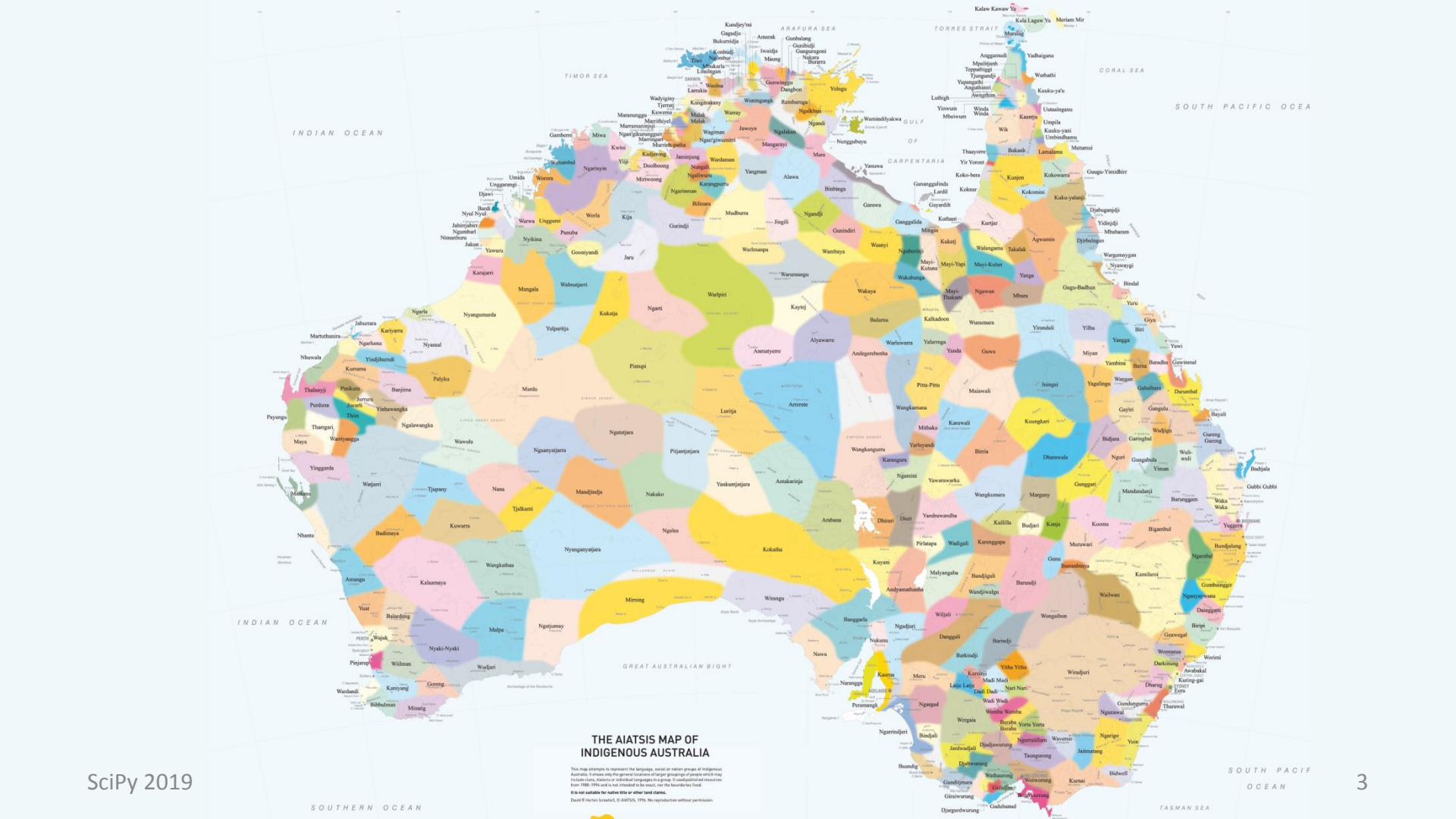
Exercise #1

- Clone repo: tiny.cc/zhd-workshop
  github.com/Zac-HD/escape-from-automanual-testing
- `pip install pytest hypothesis`
- `pytest *.py`
  - to check that it's all installed ☺

Escape from auto-manual testing with Hypothesis!

Zac Hatfield-Dodds


THE AIATSIS MAP OF INDIGENOUS AUSTRALIA
INDIAN OCEAN
TIMOR SEA
ARAFURA SEA
TORRES STRAIT
CORAL SEA
SOUTH PACIFIC OCEAN
GULF OF CARPENTARIA
GREAT AUSTRALIAN BIGHT
SOUTHERN OCEAN
TASMAN SEA

Outline

1. Motivation
2. Writing testable code
3. Property-based testing 101 *Ex 1*
4. Describing data with Hypothesis *Ex 2*
5. Testing tactics *Ex 3*

“This is your brain on software”

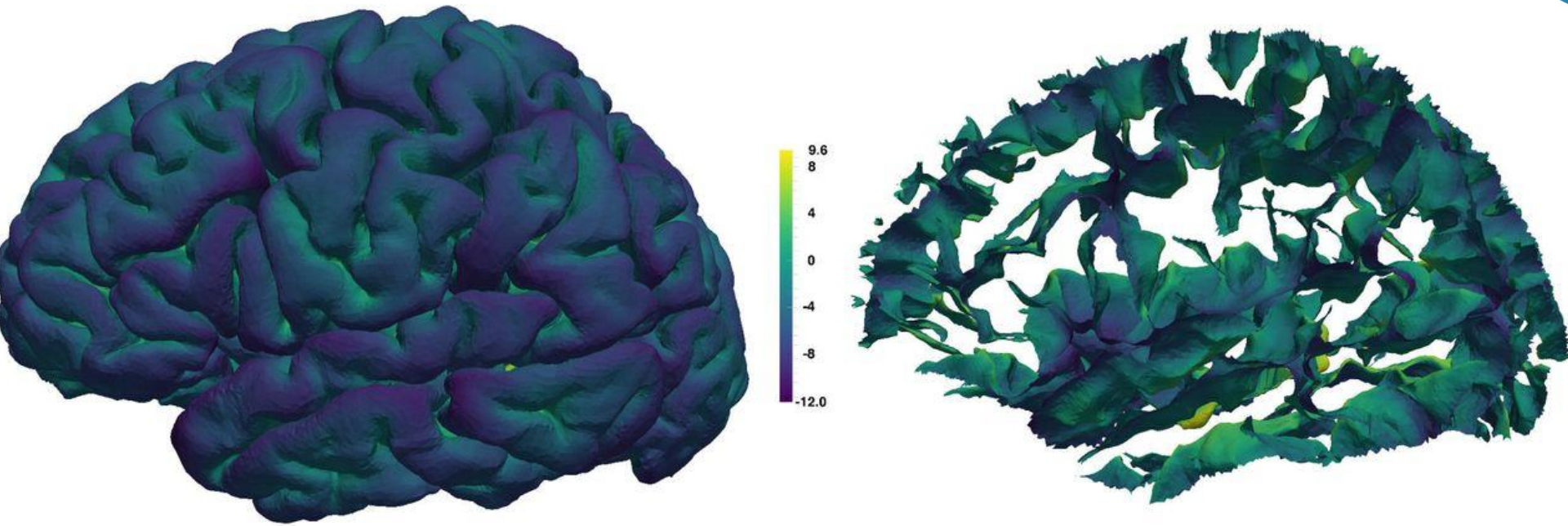
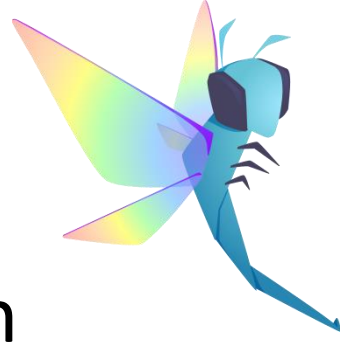


Image: Klein et al 2017, http://dx.doi.org/10.1371/journal.pcbi.1005350

... any reported scientific result could be wrong if data have passed through a computer, and these errors may remain largely undetected ...

-- Soergal (2015) https://dx.doi.org/10.12688%2Ff1000research.5930.2

Why are we here?



- Code has become critical to our research
  - Open source libraries
  - One-off analyses

- We need faster, better ways to test it
  - Still with me? Great!

Secretly the most important part:

WRITING TESTABLE CODE

Design for testability

- Deterministic behaviour
- Immutable data
- Canonical data
- Keep I/O out of logic
- Lots of assertions

What is an assertion?

"an expression in a program
which is **always true**
unless there is a bug."

http://wiki.c2.com/?WhatAreAssertions

What should a test do?

- Execute the “system under test”
  - “arrange, act, assert”
  - “given, when, then”

- Fail iff a bug is introduced

“Auto-manual tests”

- Humans
  - Decide the input
  - Write the test function
  - Determine and check expected results

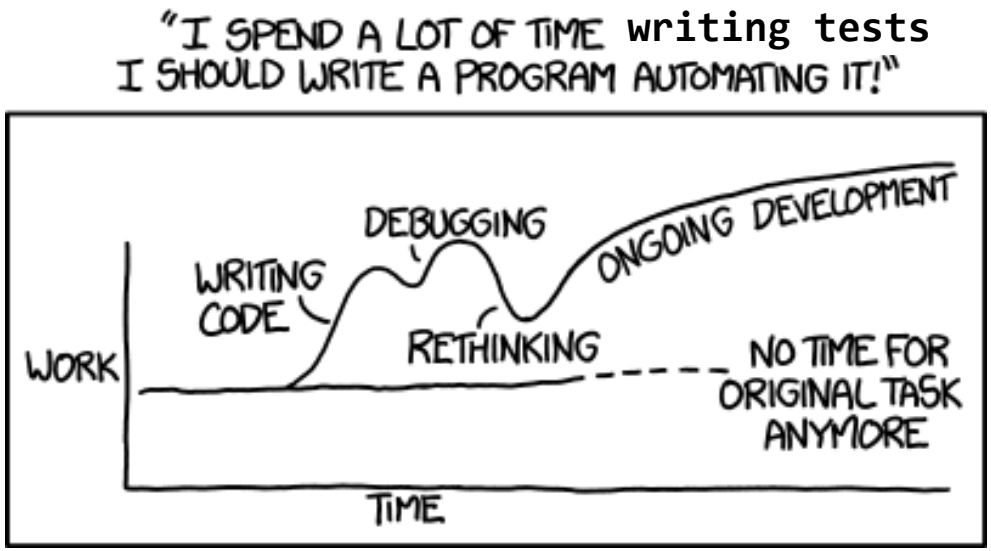
- Doing it in code is *repeatable*, not *automated*

Other kinds of tests

- Diff tests
  - Does new version reproduce known output?
- Doctests
  - Check that examples in docs still work
- Coverage tests
  - Identify unexecuted parts of your code
  - Use number, not percentage, of uncovered lines

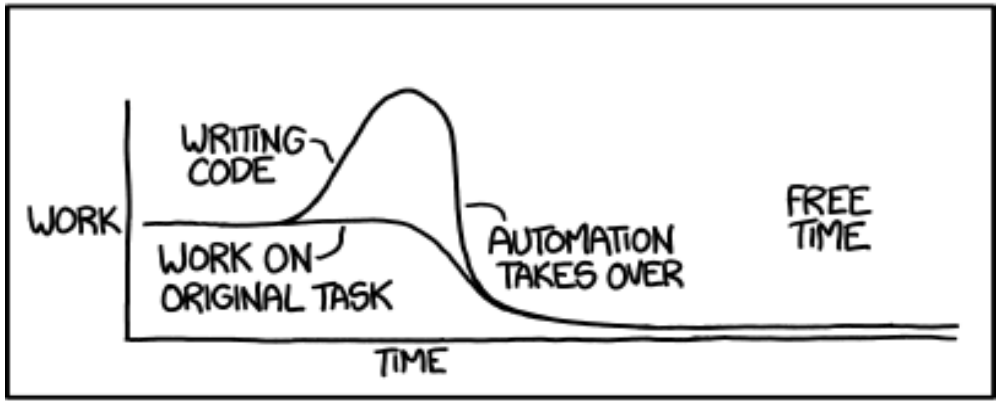
Property-based testing

- User:
  - Describes valid inputs
  - Writes a test that passes for any valid input
- Engine:
  - Generates many *test cases*
  - Runs your test for each input
  - Reports minimal failing inputs (usually)



PROPERTY-BASED TESTING 101

"I SPEND A LOT OF TIME writing tests
I SHOULD WRITE A PROGRAM AUTOMATING IT!"



PROPERTY-BASED TESTING 101

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

@given(
    st.lists(st.integers(), min_size=1)
)
def test_a_sort_function(ls):

    # we can compare to a trusted implementation,
    assert dubious_sort(ls) == sorted(ls)

    # or check the properties of dubious_sort directly.
    assert Counter(out) == Counter(ls)
    assert all(a<=b for a, b in zip(out, out[1:]))
```

Exercise #1

- Clone repo: tiny.cc/zhd-workshop
  github.com/Zac-HD/escape-from-automanual-testing
- `pip install pytest hypothesis`
  - In your preferred environment, py2 or py3
- `pytest pbt-101.py`
- Open file, edit per comments, re-run tests

DESCRIBING DATA

hypothesis.strategies

- Describes inputs for `@given` to generate

- Only construct strategies via the public API
  - SearchStrategy type is only public for type hints
  - Composing factories is nicer anyway!

Values

- Simplest strategies are for values
  - None, bools, numbers, Unicode or binary strings…
- Finer-grained than types
  - Optional bounds for value or length
  - Arguments like `allow_nan` or `timezones`

Collections

- Lists, sets, dicts, iterables, etc.
  - Take a strategy for `elements` (or keys/values)
  - Optional min_size and max_size

Map and Filter methods

`s.map(f)`

- applies function *f* to example
- shrinks *before* mapping

`s.filter(f)`

- retry unless *f(ex)*
- mostly for edge cases

```
s = integers()
s.map(str) # strings of digits

# even integers
s.map(lambda x: x * 2)
# odd ints, slowly
s.filter(lambda x: x % 2)

# Lists with some unique numbers
lists(s, 2).filter(
  lambda x: len(set(x)) >=2
)
```

Complicated data

- Got a list of values?
  - `sampled_from` or `permutations` can help
- Recursive strategies just work
  - At least three ways to define them
- Combine strategies:
  - `integers() | text()`
  - Can't take intersection though.
- Call anything with `builds()`

Inferring strategies

A schema is a machine-readable description:

- Used for validating input
- Can generate input instead!

This tests both validation and logic.

regex, array dtype, django model, attrs classes, type hints, database…

```
>>> from_regex(r'^[A-Z]\w+$')
'Fgjdfas'
'D榙譞Ć奇\n'


>>> from_dtype('f4,f4,f4')
(-9.00713e+15, 1.19209e-07, nan)
(0.5, 0.0, -1.9)


>>> def f(a: int): return str(a)
>>> builds(f)
'20091'
'-507'
```

Beyond the standard library

- hypothesis.extra
  - Django, Numpy, Pandas, Lark, pytz, dateutil…
- Also many third-party extensions, e.g.
  - Geojson, SQLAlchemy, networkx, jsonschema, Lollipop, Mongoengine, protobuf…

Data dependencies

Custom strategies

- Similar to interactive data in tests

```
@composite
def str_and_index(draw, min_size):
    s = draw(text(min_size=min_size))
    i = draw(integers(0, len(s) - 1)
    return (s, i)

str_and_index().example()
```

Interactive data

- Run part of a test, then get more input
- Useful with complex dataflow

```
@given(data())
def test_something(data):
    i = data.draw(integers(...))
```

Minimal examples

- Strategies shrink
  - From the inside out
    - i.e. before map or filter are applied
  - Towards the smallest and shortest example
  - Based on the strategy definition
- Multiple errors possible per test!

Inline `st.data()`

- Draw more data *within* the test function
  - Great for complex or stateful systems
  - Use `@composite` instead if you can

```
@given(st.data())
def a_test(data):
    x = data.draw(integers(0, 100), label="First number")
    y = data.draw(integers(x, 100), label="Second number")
    # Do something with `x` and `y`
```

TACTICS FOR TESTS

Tactics: what do we test?

- “Auto-manual” testing
  - output == expected
- Oracle tests (full specification)
  - Does a magic “oracle” function say output is OK?
- Partial specification
  - Can identify some but not all failures
- Metamorphic testing
- Hyper-properties

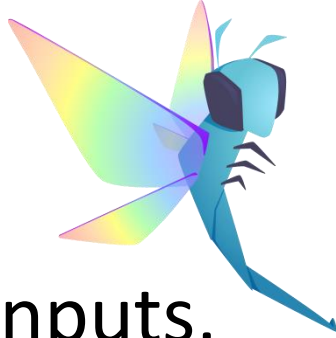
Oracles

- Fantastic for refactoring or testing performance optimisations
- “reverse oracles”
  - Generate an answer, ask the oracle for a matching question, test that code gets the answer
- You may need to test the Oracle too

Partial specification

- We don’t need an exact answer for tests!
  - min(xs) <= mean(xs) <= max(xs)
- Lots of serialisation specs are like this
  - In fact almost all specs are partial

Special-case oracles



- If your oracle only works for some valid inputs, that's still useful to test those inputs
- Or a more precise test for a subset of inputs
  - Monotonic functions, positive numbers, etc.
  - Varying just one parameter to simplify results

Common properties

- Shared by lots of code
  - Often good API design generally
  - Or worth it just for testability

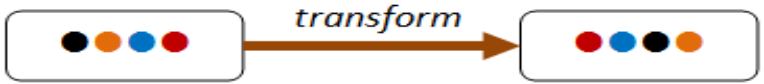
“Does not crash”

- Just call your function with valid input:

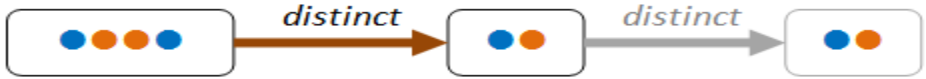
```
@given(lists(integers()))
def test_fuzz_max(xs):
    max(xs) # no assertions in the test!
```

- This is embarrassingly effective.

Invariants




```
Counter(ls) == Counter(sorted(ls))
```

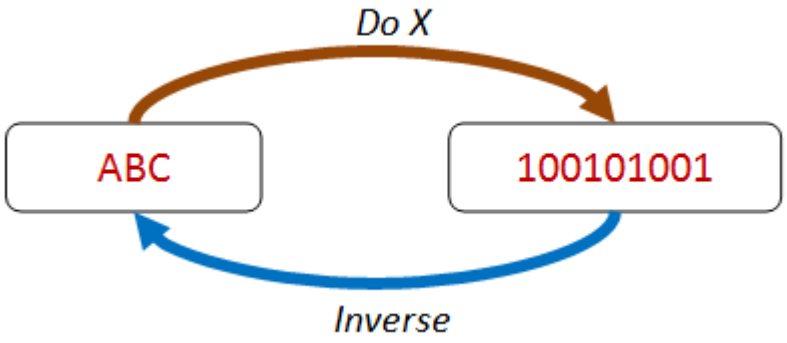



```
ls != set(ls) == set(set(ls))
```

Round-trips

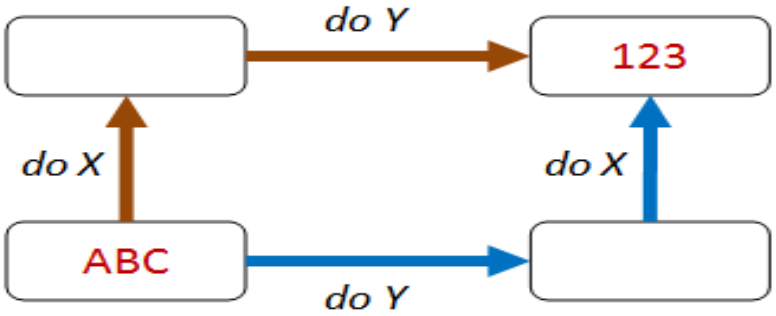
"inverse functions"

- `add` / `subtract`
- `json.dumps` / `json.loads`



or just related:

- `factorize` / `multiply`
- `set_x` / `get_x`
- `list.append` / `list.index`



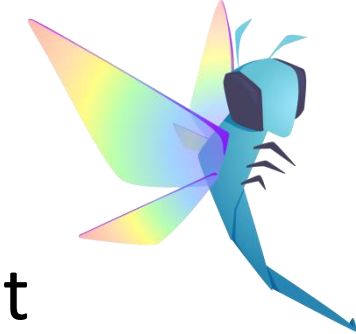
Exercise #2

- Same plan as last time:
  - `pytest strategies-and-tactics.py`
  - Open, edit, re-test

Metamorphic Relations

TESTING BLACK BOXES

Metamorphic relations?



- We don’t know how input relates to output
- BUT
  - Given an input and corresponding output
  - Make a known change to the input
  - We might know how the output should change (or not change)

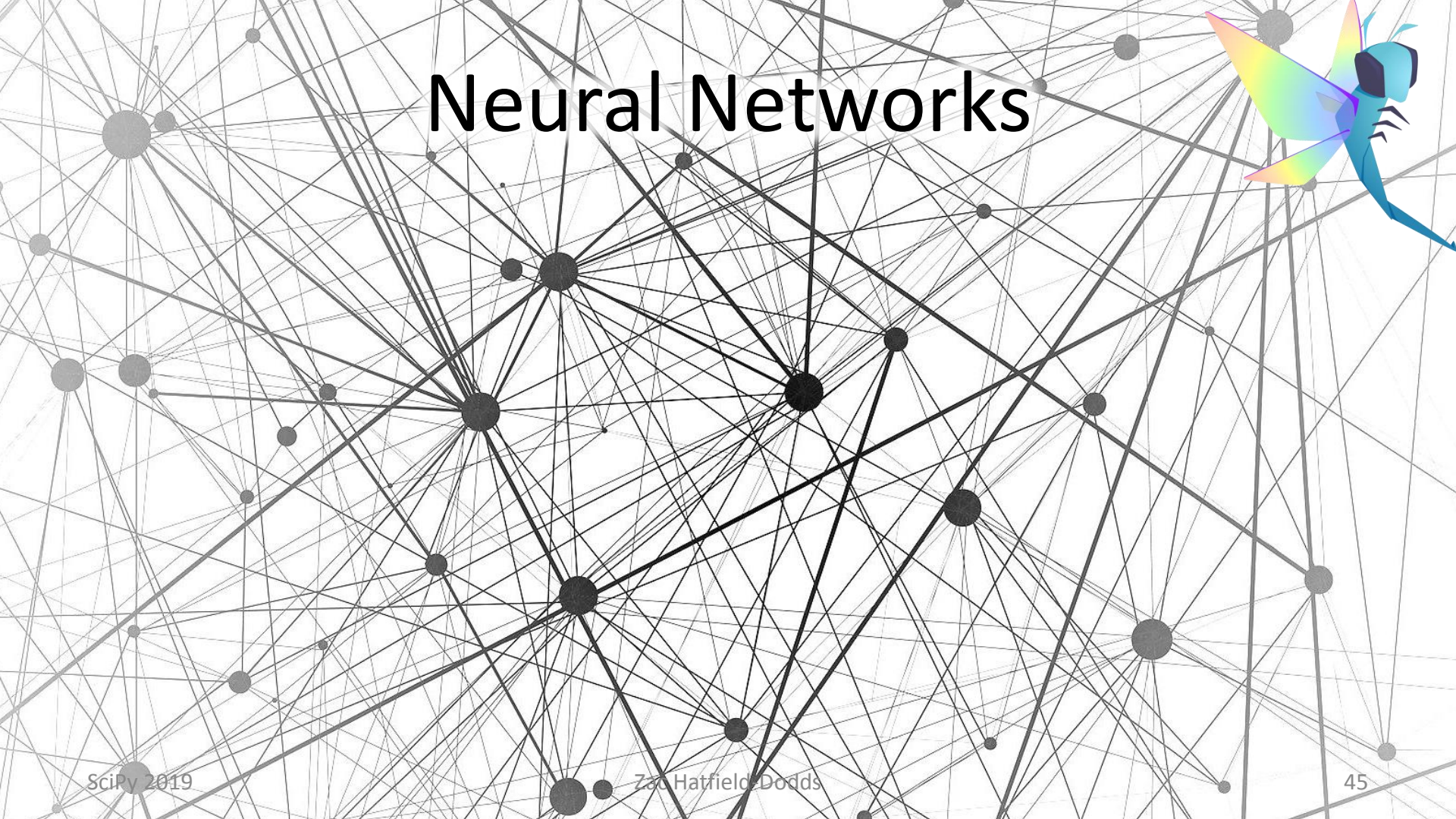
- That’s it – but this is really, really powerful

Compilers

- Very popular technique for compilers!
  - Generate valid program
  - Use many different compilers and settings
  - Run and compare results
  - Any difference == there's a bug somewhere

- A kind of differential testing

Neural Networks

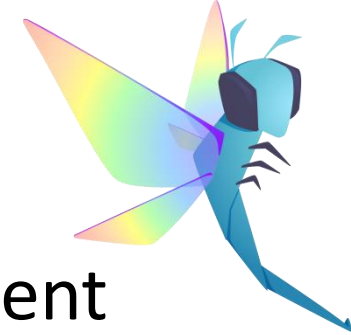


Neural Networks

- Testing for implementation errors is rare
  - Embed lots of assertions
  - Use simple properties across single steps

- Testing things like…
  - Training steps change neuron weights
  - Bounds on inputs and outputs
  - Converges when expected to

Computer vision



- Cars should not confidently choose a different action if the camera feed has
  - slight changes in contrast or brightness or scale
  - a small skew or offset or blur added
  - light rain or fog added

- https://deeplearningtest.github.io/deepTest/
  - Luckily there aren't many on the road (yet)

Exercise #3

- Same plan as last time, choice of two files:
  - `pytest scientific-hypothesis.py`
  - `pytest test-the-untestable.py`
  - Open, edit, re-test

In which we discuss all the other things that you might want to know.

PERFORMANCE, CONFIGURATION, AND COMMUNITY

Observability

`--hypothesis-show-statistics`

- Shows timing stats, perf breakdown, exit reasons
- Add custom entries by calling `event()` in a test

- Use `note()` if you like print-debugging
  - Only prints for minimal failing example
  - Details controlled by `verbosity` setting

Performance (generation)

- All pretty obvious in generation phase:
  - Calling slow things or many things is slow
  - Generating larger data takes longer
  - Filter more, and getting output takes longer
- Otherwise Hypothesis is pretty fast!

Performance (shrinking)

- Composition of shrinking
  - If any part shrinks, the whole should shrink
  - Order of recursive terms is important!
- Keep things local
  - Put filters (or assume) as far in as possible
  - Avoid drawing a size, then that many things
- Don’t waste more tuning than you save!

Configuration

- `hypothesis.settings`
- Per-test decorator or whole-suite profiles
- Lots of options
  - deadline, max_examples, report_multiple_bugs, database, etc.

Reproducing failures

- Hypothesis tests should **never** be flaky.
  - We detect most user-caused flakiness too
- Failures cached and retried until fixed
  - for local dev, reproducibility is automatic
- Printed seed to re-run failures from CI
- Explicit decorator for really tough cases

Update early & often!

- Hypothesis releases every pull request.
  - All bug fixes are available in ~30 minutes
  - As are features, performance improvements, …
  - We use strict semver and code review
  - (and have a fantastic test suite ☺)

- So stay up to date – for your own sake!

Who uses Hypothesis?

- 4% of all Pythonistas (PSF survey)
- Many companies
- ~2000 open source projects (github stats)

Consulting Services

- Want exciting new features?
- Want Hypothesis training for your team?
- Want your tests (and code) reviewed?

- Zac Hatfield-Dodds and David MacIver
  - Say hi via hello@hypothesis.works

About the project

- MPL-2.0 license
- New contributors welcome!
  - most remaining issues are non-trivial
  - using or extending Hypothesis is valued too

- Tries to be *legible*
  - we design APIs and errors to teach users
  - does what you expect; or explains why not