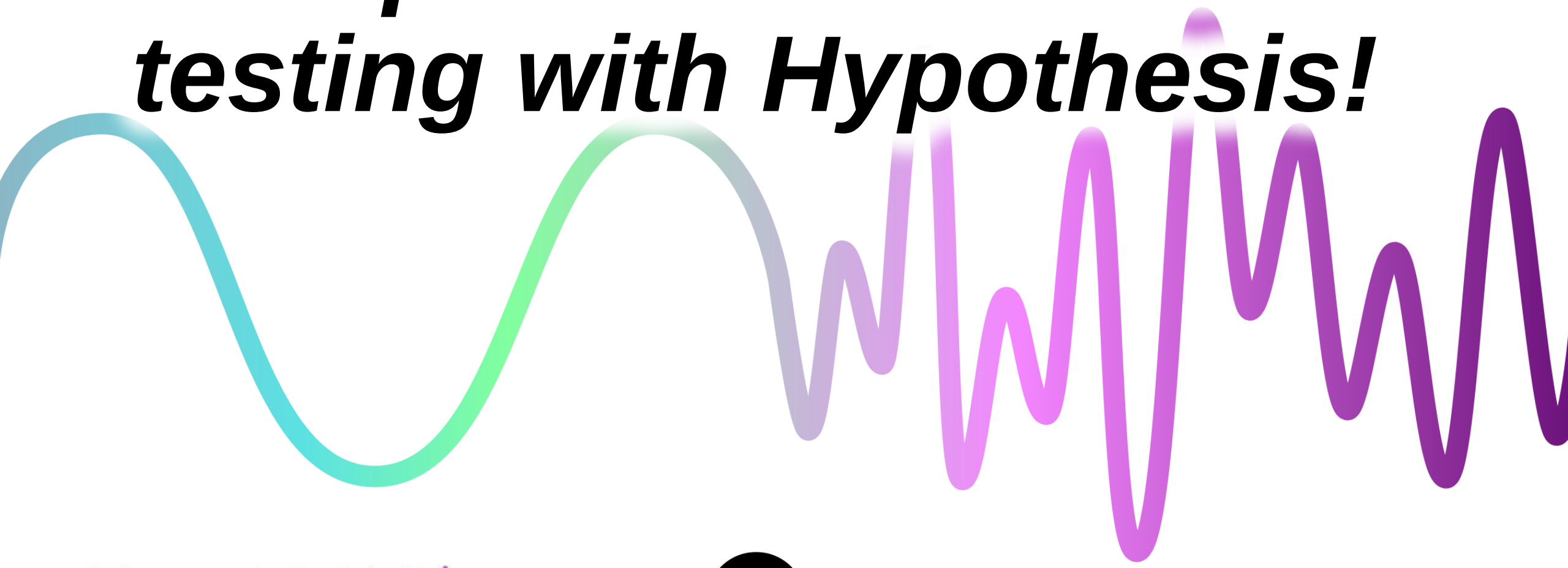
Escape from automanual testing with Hypothesis!



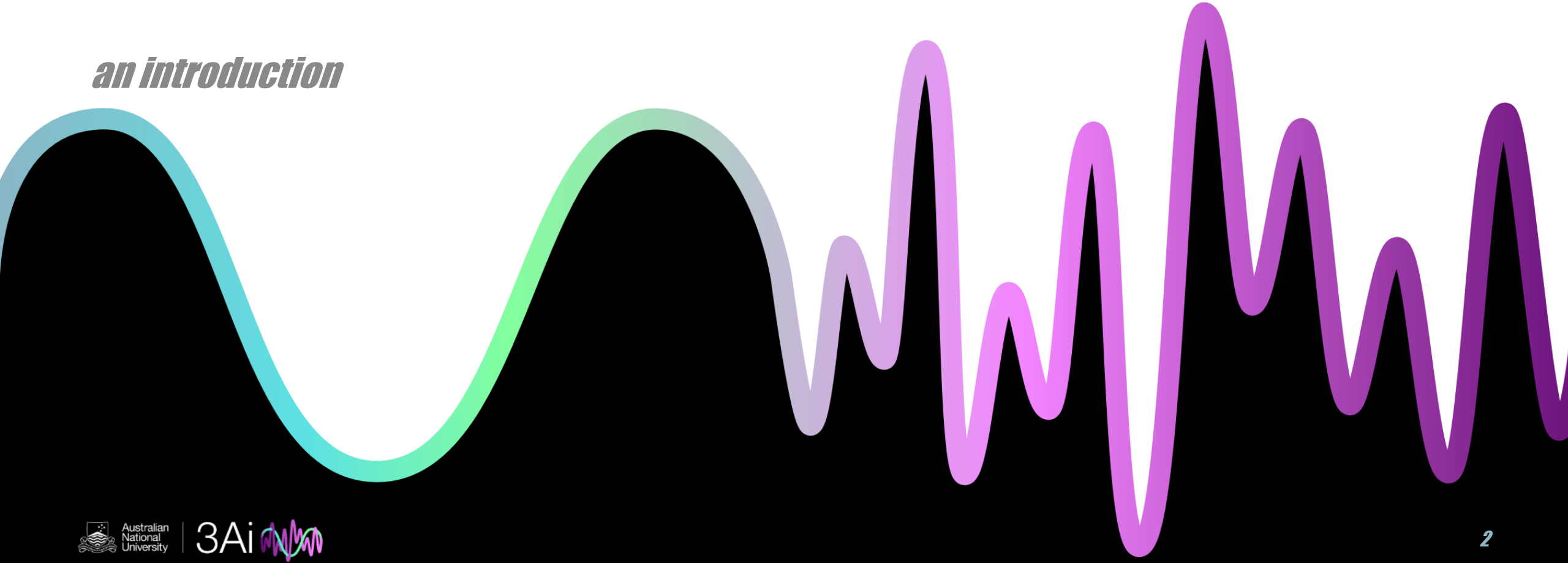
Australian National University | 3Ai

Zac-HD/slides

3AINSTITUTE.ORG

Different kinds of tests

an introduction



Australian National University | 3Ai

What is Auto-manual testing?

“manual test”

= a human *executes* the test

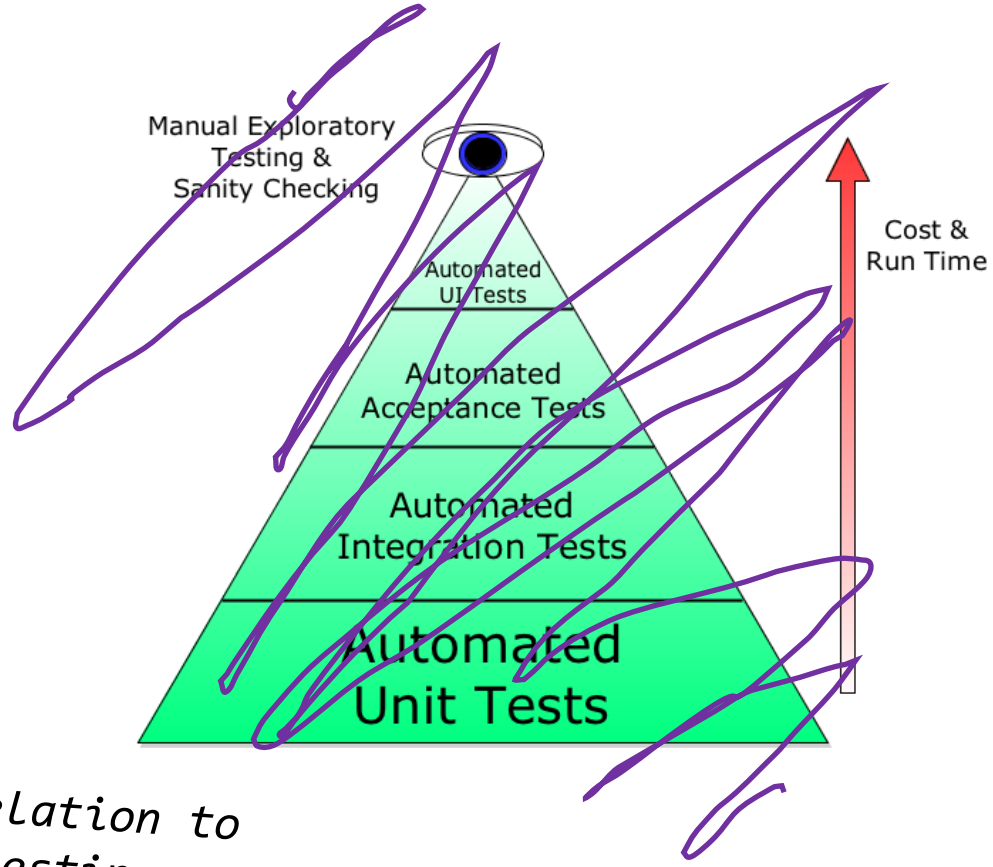
“automated test”

= a program that tests a program

“auto-manual test”

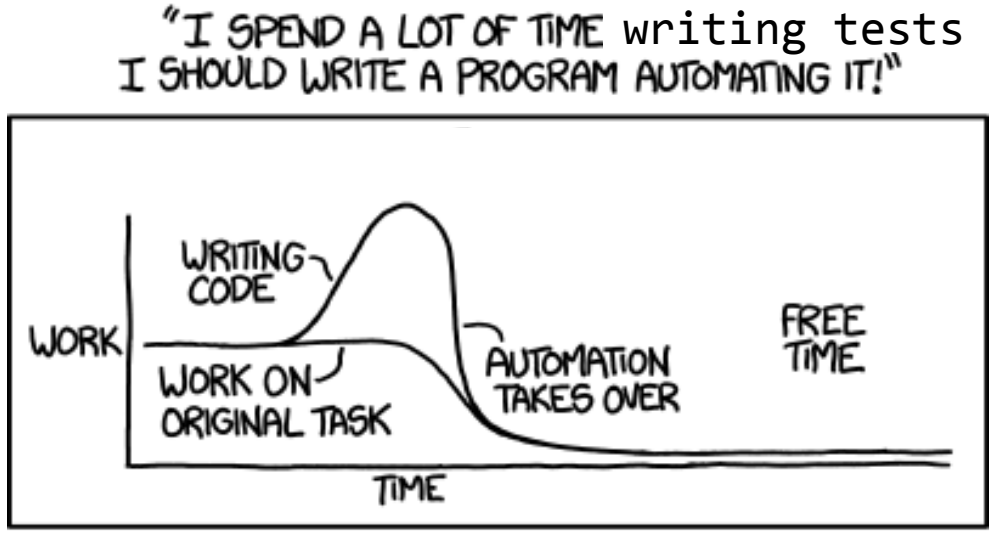
= hand-written by a human

unit tests, integration tests, parametrised tests, etc.



No relation to the testing pyramid ↑

Generative Tests



Using a library lets you skip the 'ongoing development' panel from the original comic

Basically automating automated testing!

Hypothesis can generate…

- arguments to a test function
- entire test programs!

Other kinds of tests

Diff tests

compare output to golden record

Mutation tests

add bugs to tests your tests

Coverage tests

find untested code

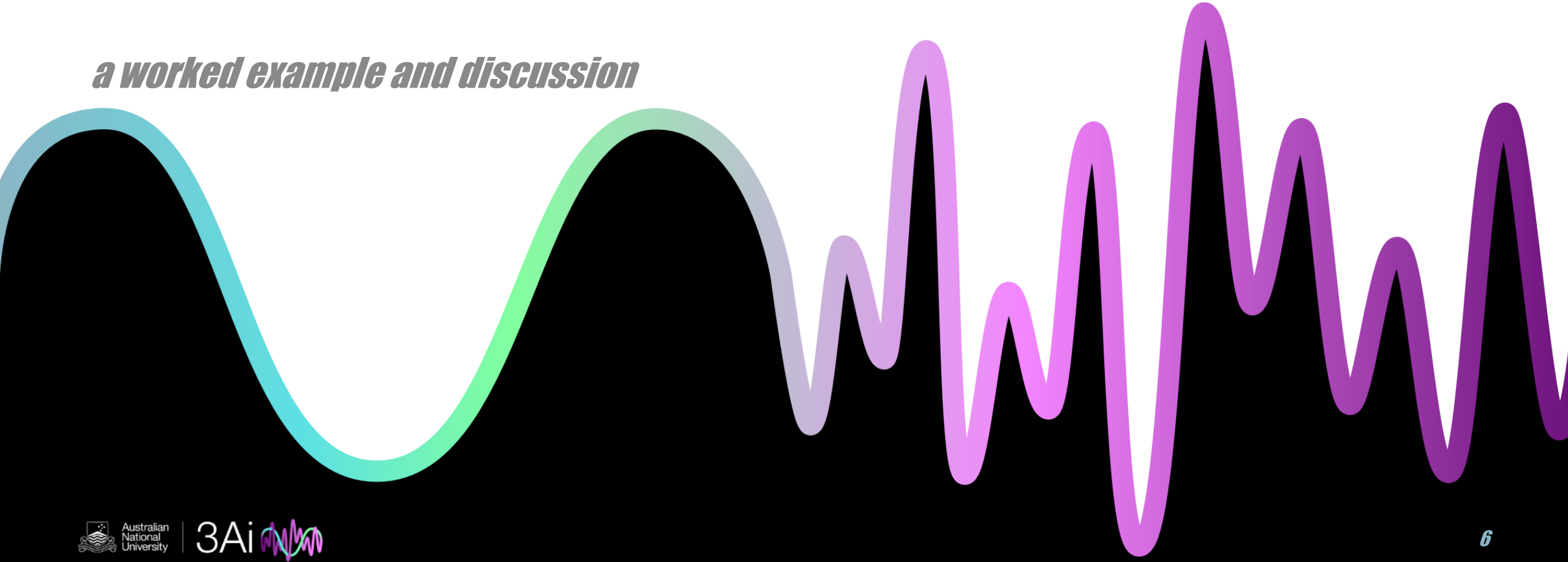
***NOT** a percentage measure!*

Static analysis

analyse code without running it

Your first property-based test

a worked example and discussion



Australian National University | 3Ai

*A Simple Example**

"The sum of a list of integers is greater than the largest element in the list"

```
def test_sum_above_max_small():
    xs = [1, 2, 3]
    assert sum(xs) > max(xs), ...

def test_sum_above_max_large():
    xs = [10, 20, 30]
    assert sum(xs) > max(xs), ...
```

** of **how** to test, not **what** to test!*

A Simple Example

“The sum of a list of integers is greater than the largest element in the list”

```
import pytest

@pytest.mark.parametrize('xs', [
    [1, 2, 3], [10, 20, 30], ...
])
def test_sum_above_max(xs):
    assert sum(xs) > max(xs), ...
```

A Simple Example

“The sum of a list of integers is greater than the largest element in the list”

What do we learn?

```
from hypothesis import given
from hypothesis.strategies import lists, integers


@given(lists(integers()))
def test_sum_above_max(xs):
    assert sum(xs) > max(xs), ...
```

A Simple Example

“The sum of a list of integers is greater than the largest element in the list”

What do we learn?

- Error to call `max([])`

```
from hypothesis import given
from hypothesis.strategies import lists, integers


@given(lists(integers()))
def test_sum_above_max(xs):
    assert sum(xs) > max(xs), ...
```

```
Falsifying example: test_sum_above_max(xs=[])

Traceback (most recent call last):
    ...
ValueError: max() arg is an empty sequence
```

A Simple Example

“The sum of a list of integers is greater than the largest element in the list”

What do we learn?

- Error to call `max([])`

```
from hypothesis import given
from hypothesis.strategies import lists, integers


@given(lists(integers(), min_size=1))
def test_sum_above_max(xs):
    assert sum(xs) > max(xs), ...
```

A Simple Example

“The sum of a list of integers is greater than the largest element in the list”

What do we learn?

- Error to call `max([])`
- Need greater *or equal* for 1-lists

```
from hypothesis import given
from hypothesis.strategies import lists, integers


@given(lists(integers(), min_size=1))
def test_sum_above_max(xs):
    assert sum(xs) > max(xs), ...
```

```
Falsifying example: test_sum_above_max(xs=[0])

Traceback (most recent call last):
    ...
AssertionError: xs=[0], sum(xs)=0, max(xs)=0
```

A Simple Example

“The sum of a list of integers is greater than the largest element in the list”

What do we learn?

- Error to call `max([])`
- Need greater *or equal* for 1-lists

```
from hypothesis import given
from hypothesis.strategies import lists, integers


@given(lists(integers(), min_size=1))
def test_sum_above_max(xs):
    assert sum(xs) >= max(xs), ...
```

A Simple Example

“The sum of a list of integers is greater than the largest element in the list”

What do we learn?

- Error to call `max([])`
- Need greater *or equal* for 1-lists
- Can’t have negative integers

```
from hypothesis import given
from hypothesis.strategies import lists, integers


@given(lists(integers(), min_size=1))
def test_sum_above_max(xs):
    assert sum(xs) >= max(xs), ...
```

```
Falsifying example: test_sum_above_max(xs=[0, -1])

Traceback (most recent call last):
    ...
AssertionError: xs=[0, -1], sum(xs)=-1, max(xs)=0
```

A Simple Example

“The sum of a list of integers is greater than the largest element in the list”

What do we learn?

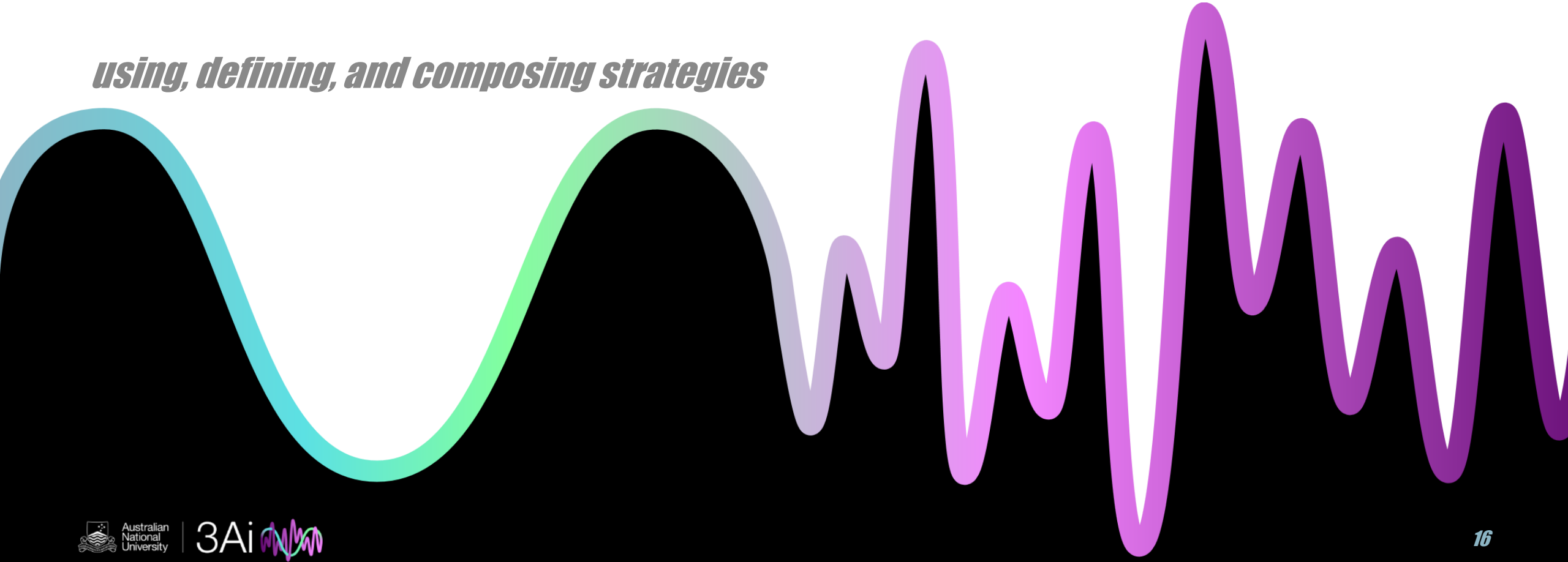
- Error to call `max([])`
- Need greater *or equal* for 1-lists
- Can’t have negative integers

And this version works!

```
from hypothesis import given
from hypothesis.strategies import lists, integers


@given(lists(integers(min_value=0), min_size=1))
def test_sum_above_max(xs):
    assert sum(xs) >= max(xs), ...
```

Describing inputs

using, defining, and composing strategies



Values and Collections

First-party support for common types

- None, bool, numbers, strings, times…
- `min_value` and `max_value` args
- type-specific args

```
floats(allow_infinity=False)
times(..., timezones=...)
```

Collections are composed:

- From elements and sizes
- From keys or indices

fancier options are later in the talk

Transforming Strategies

`.map(f)`
- applies function *`f`* to example
- minimises *before* mapping

`.filter(f)`
- retry unless *`f(ex)`* truthy
- mostly for edge cases

```
s = integers()

s.map(str)  # strings of digits

s.map(lambda x: x * 2)  # even integers

s.filter(lambda x: x % 2)  # odd ints, slowly

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

Advanced Options

Recursive data

- Several ways to recur, e.g.:

Interactive data

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

Custom strategies

- Similar to interactive data in tests

```
strat = deferred(
    lambda: integers() | lists(strat))

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

@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)
```

Infer from Schema

A schema is a machine-readable description of valid data

- Used for validating input
- Can generate input instead!

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

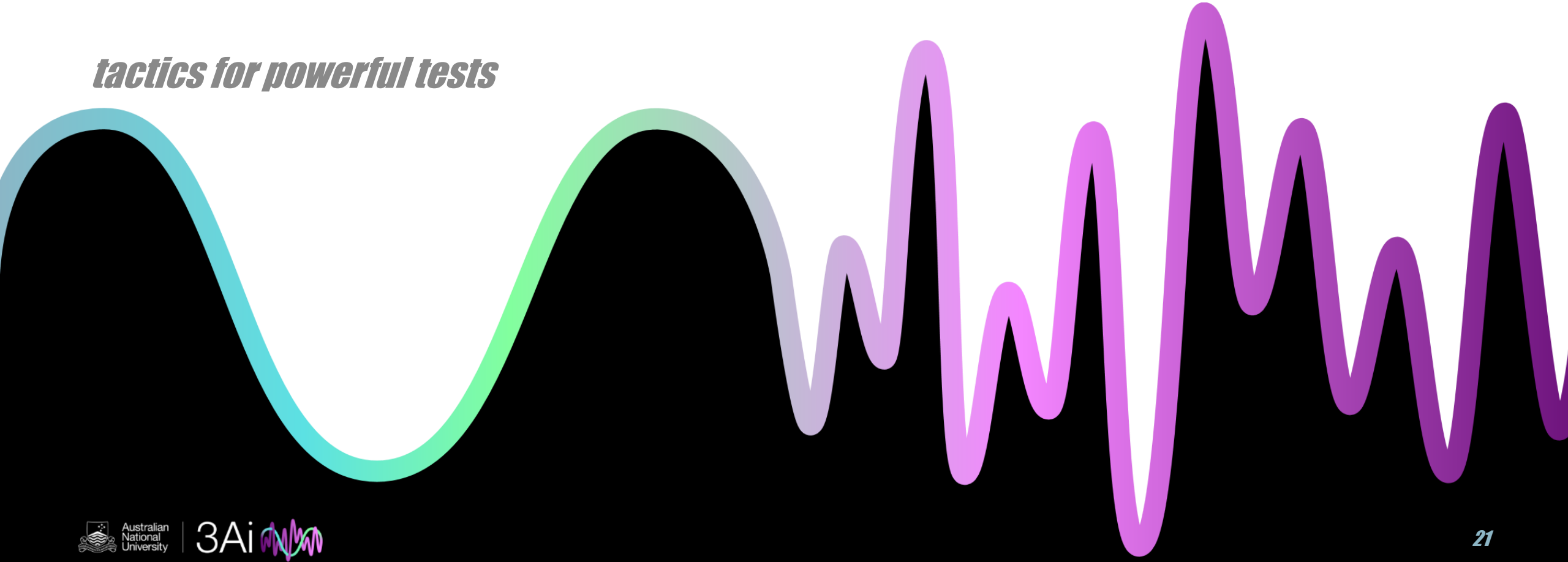
```
>>> from_regex(r'^[A-Z]\w+$')
'Fgjdfas'
'D楮譭Ć斋\n'
```

```
>>> 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'
```

Choosing Properties

tactics for powerful tests



Fuzzing

Just call your function with valid input:

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

This is *embarrassingly* effective.

An aside on Assertions

An assertion is an expression which must be true *unless* there is a bug in the program.

Not for error handling –
the expression might not be evaluated!

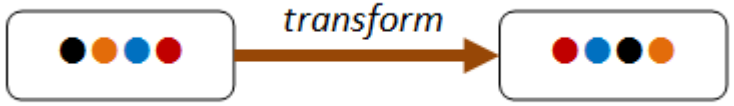
Great for localising bugs!

“Design by Contract”
= putting assertions in the main code
-> free integration tests!

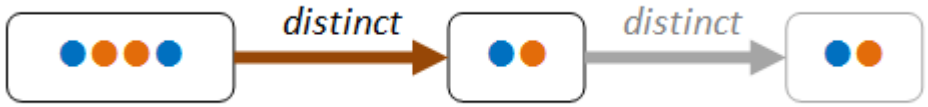
“Property based testing”
= more specific assertions in tests
-> makes fuzzing more powerful!

See https://blog.regehr.org/archives/1091
and https://www.hillelwayne.com/talks/beyond-unit-tests/

Some things never change




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




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

Invariants are great…

If your code should have them, test them!

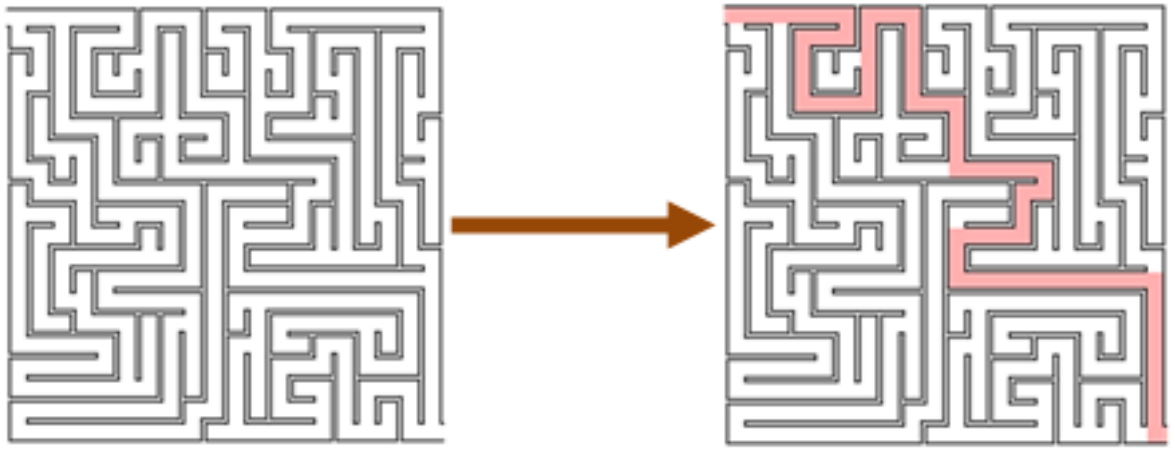
Hard to prove, easy to verify

Find prime factors, then…

- multiply factors to get input

Tokenise a string, then…

- check each token is valid
- concatenate to get input

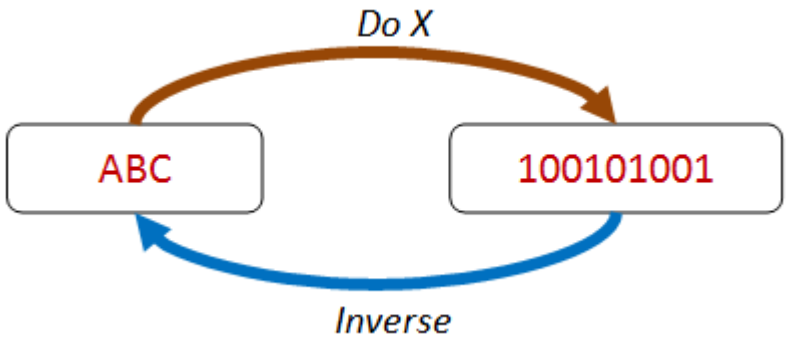


See https://fsharpforfunandprofit.com/posts/property-based-testing-2/

There and back again

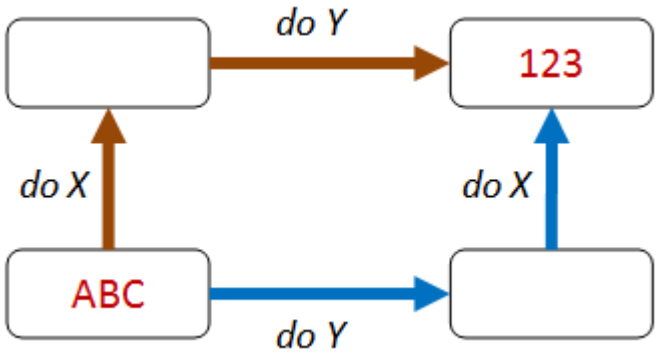
“inverse functions”

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



or just related:

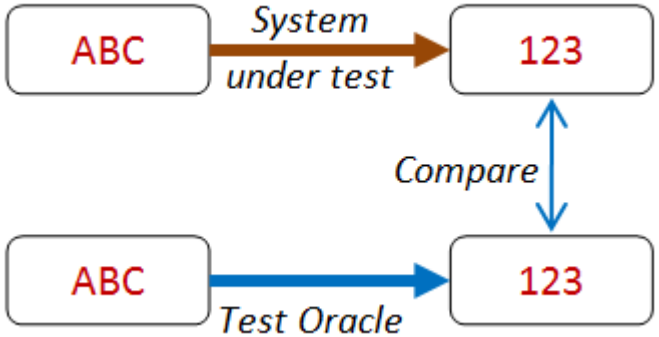
- `set_x` / `get_x`
- `list.append` / `list.index`



The Test Oracle

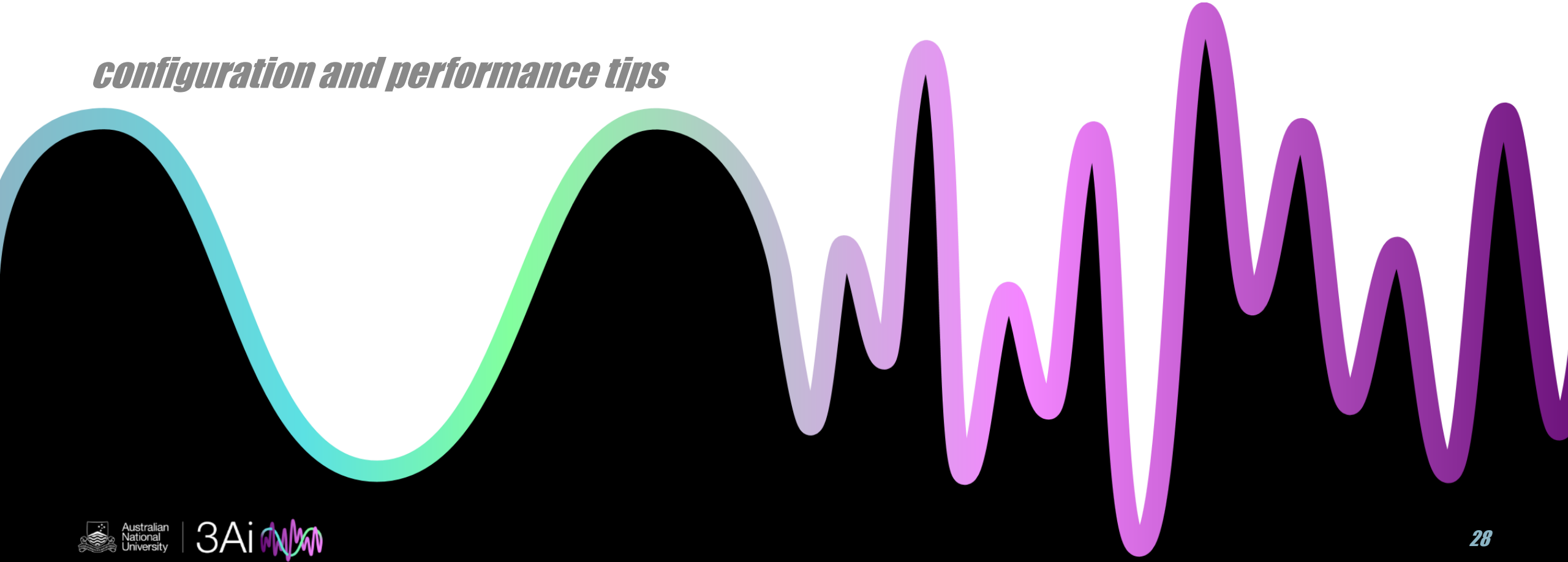
Compare to another version:

```
new_hotness(x)      == legacy(x)

fancy_algorithm(x) == brute_force(x)

foo(x, threads=10) == foo(x, threads=1)
```



Advanced Usage

configuration and performance tips



Configuring Hypothesis

- Verbosity and debug info
- How many inputs to test
- Cache location
- Deterministic mode
- *and much more!*

Configure in code or from pytest

Define and load custom profiles

```python
from hypothesis import given, settings


# Change global settings
settings.derandomize = True


# Override for a single test
@settings(max_examples=1000))
@given(...)
def test_with_settings(x):
    ...
```

Failures are always reproducible

Local Development

- Every failure is stored in a database
- Try all known failures before new tests

DB format is sharable and git-friendly

(but better not to share it!)

From CI log only

- Print a function call if that will work
- Print seed if needed
- Dump internal state if required

Less fun, but it always works

Performance tips

Hypothesis is fast!

```
> pytest --hypothesis-show-statistics
```

- mean time per example
- % time generating data (~0 to ~20%)
- define custom events

Hypothesis runs a *lot* of tests!

If your test is slow, running it hundreds of times will be *really* slow. (but you need fewer tests)

Hypothesis runs your code under coverage

- ~2x to ~5x slowdown
- still improves bugs per unit time
- can be disabled in pathological cases

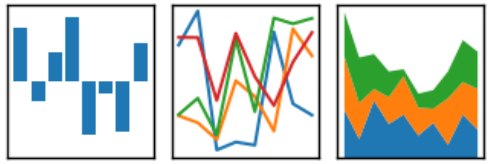
Users and use-cases

OPEN API INITIATIVE

Use `swagger-conformance` to check your REST API matches the spec – in any language

pandas

$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$



Date and time calculations with timezones, serialisation round-trips, reshaping, etc.

stripe

Complex ML features for fraud detection pipeline

Testing beginner code

from logic bugs to typos

blog.jrheard.com/hypothesis-and-pexpect

Ethereum client, VM implementation

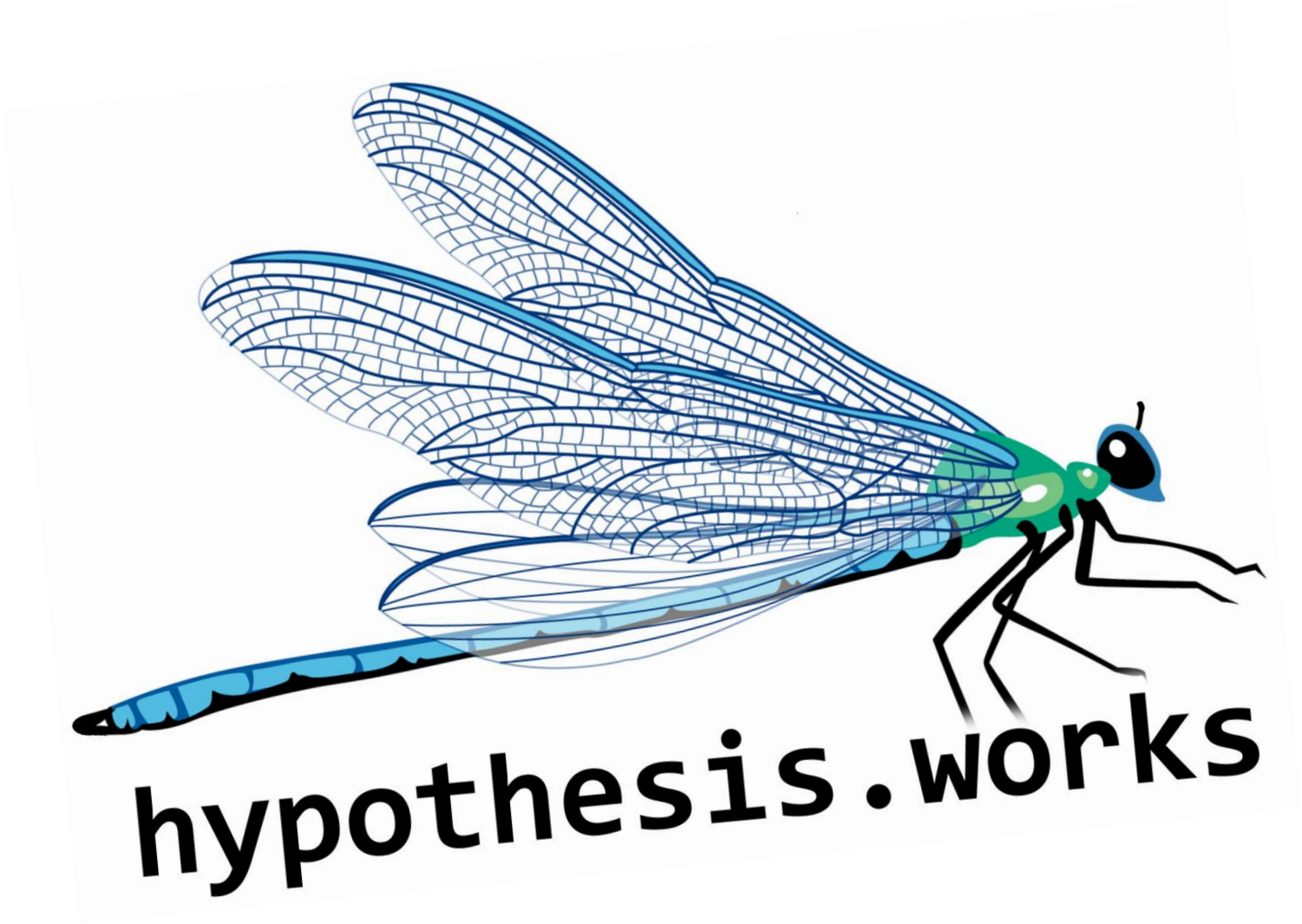
What kind of talk doesn't have blockchain in 2018?

The Project

- Mozilla Public License
- Contributors mentored!

- Training available
- Sponsor new features

- **Join us at the Sprints!**



Australian National University | 3Ai

Zac-HD/slides