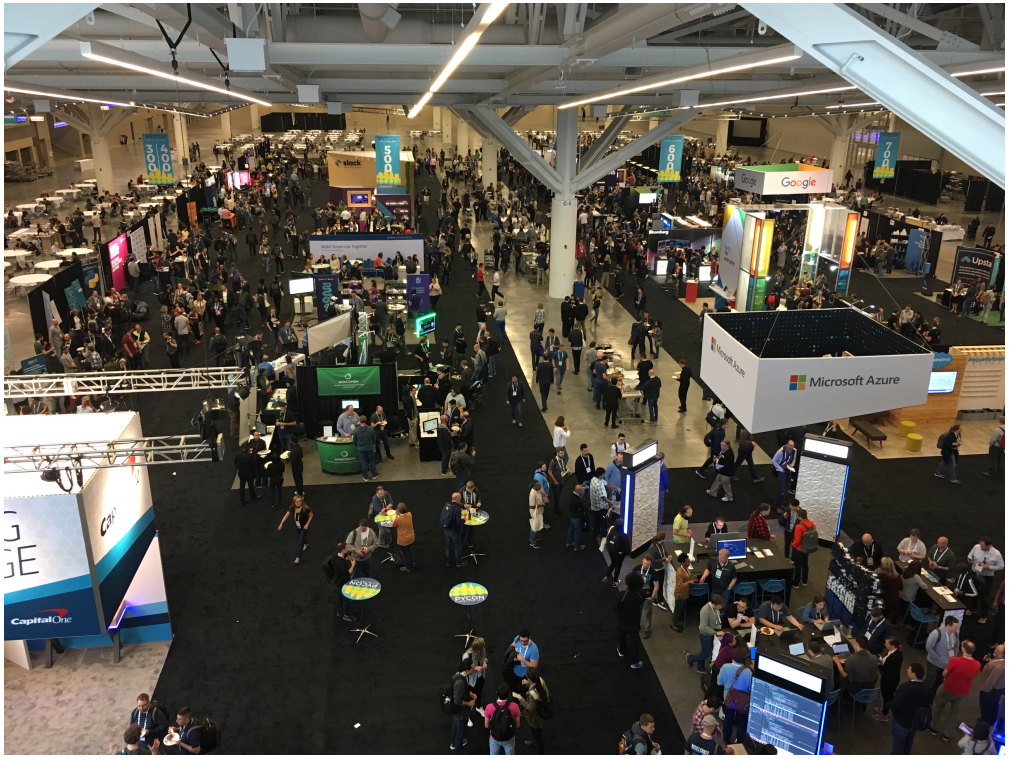
A Recap of Cool Things Learned in Cleveland.

What is PyCon?

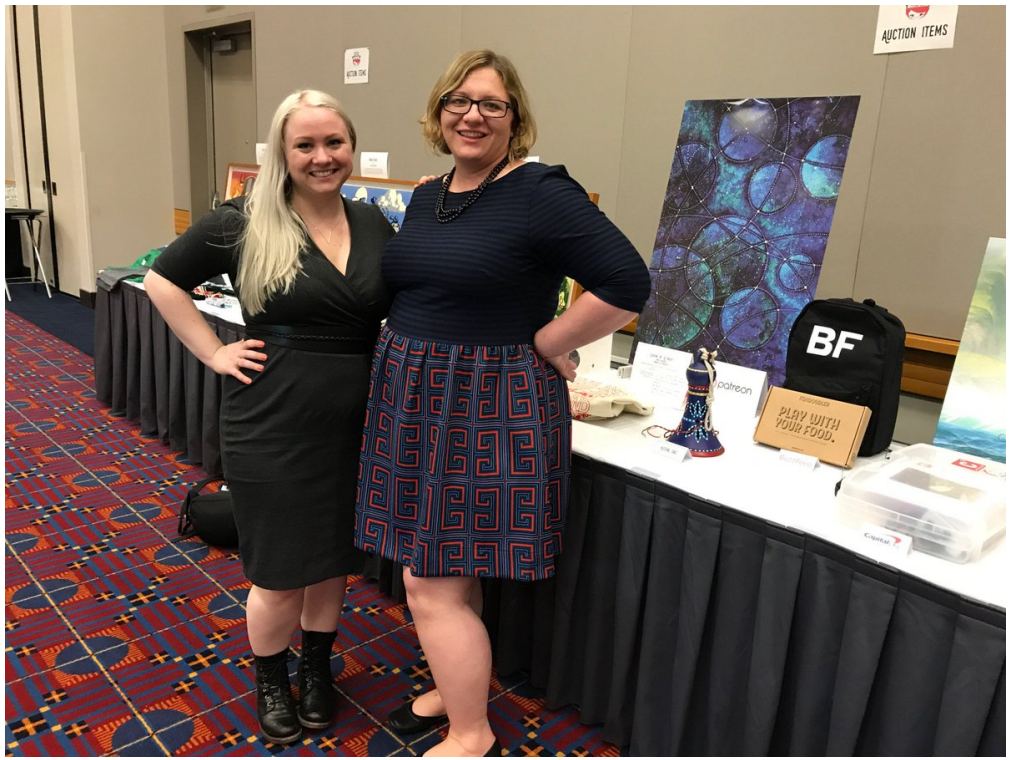
PyCon = 2 days tutorials + 3 days talks + 4 days developer sprints

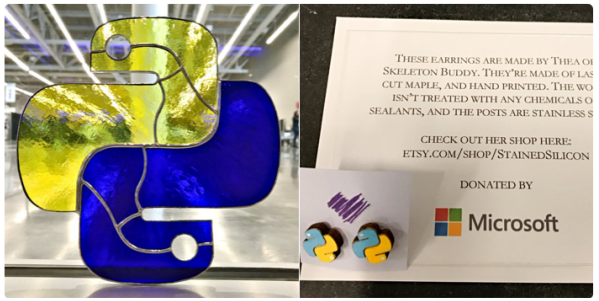
also, many parallel open spaces for informal meetups on diverse topics, the Annual PyLadies Auction, Fun Run, and more...

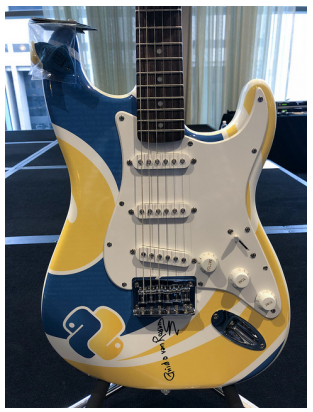


PyLadies Auction

The annual auction raised over $40K for PyLadies to continue supporting women who code!

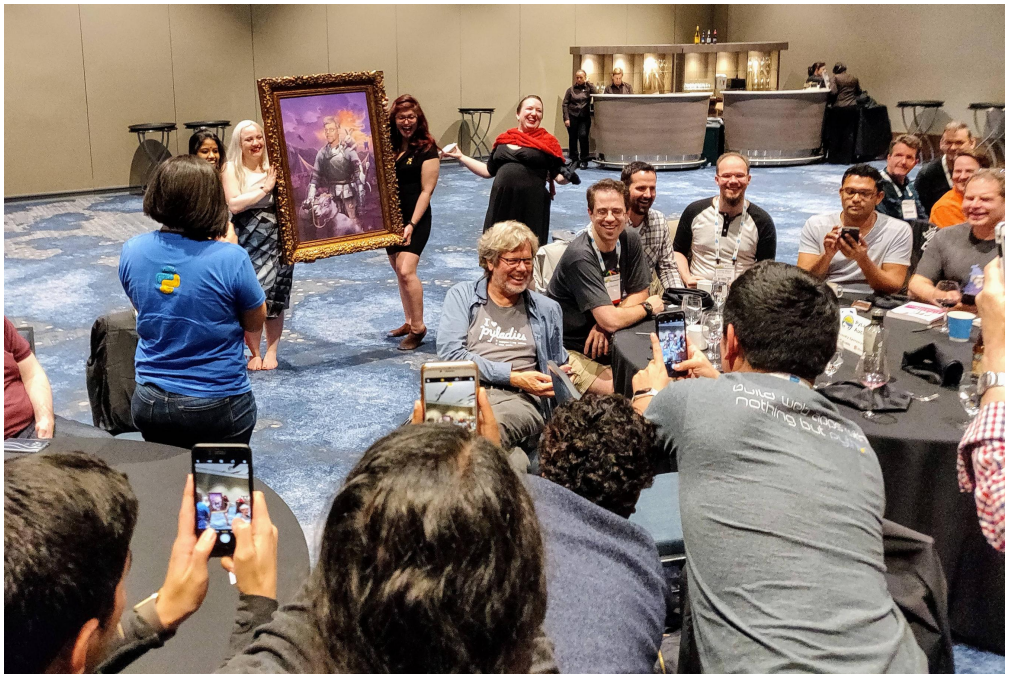






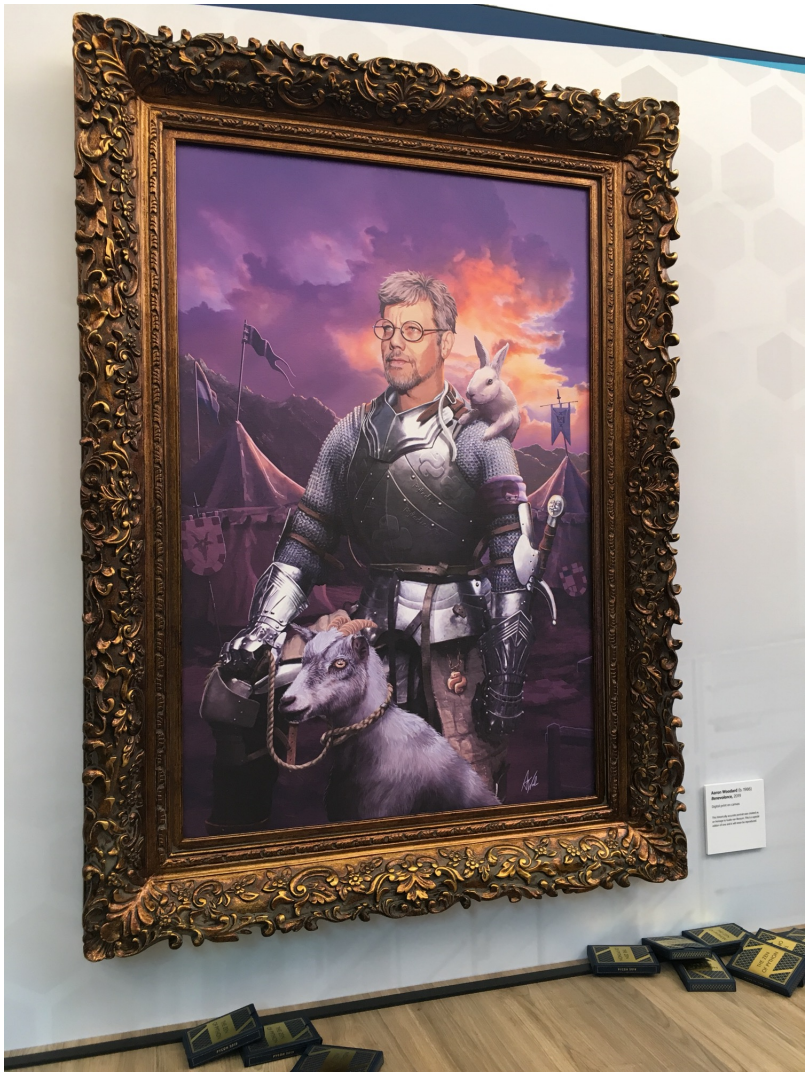
PyLadies Auction

The prize of the night was this painting of Guido, donated by capital one...



PyLadies Auction

The prize of the night was this painting of Guido, donated by capital one... which went for $9,001 after an epic auctioning battle.



API Evolution the Right Way

A. Jesse Jiryu Davis

Staff Engineer at MongoDB in New York City specializing in C, Python, and async. Author of Motor, an async MongoDB driver for Tornado and asyncio. Contributor to Python, MongoDB, Tornado, and asyncio. Co-author with Guido van Rossum of "A Web Crawler With asyncio Coroutines", a chapter in the "500 Lines or Less" book in the Architecture of Open Source Applications series.

Covenant 1. Avoid Adding Bad Features

Example: in Python 2 `datetime.time` evaluated to True at all times except at time 0 (midnight).

In [1]:
```
import datetime

if datetime.time(9, 30):
    print('9:30am is true')

if datetime.time(0, 0):
    print('midnight is true')  # True in Python 3
```

```
9:30am is true
midnight is true
```

Covenant 2. Minimize Features

- "Features are like children: conceived of in a moment of passion, they must be supported for years"
- Do not add features unless they are well motivated/necessary

Covenant 3. Keep Features Narrow

Contradicts common teachings that your code should solve very general problems. In his experience its best to only solve very specific user issues, so features don't become too powerful and get used in the wrong ways.

Covenant 4. Mark Experimental Features "Provisional"

Controversial practice to release features like this to get user feedback. Some people hate this.

Deleting Features

You discovered it was a bad idea or the ecosystem around your package changes and you need to adapt.

Cons

- User code must change
- User logic must change

Pros

- Change is mechanical
- Feature is dangerous

Deleting Features

In [2]:
```
# Example: we want to delete the legs feature so our
# lizard creature will be a snake and can no longer walk...

class Reptile:
    def walk(self):
        print('step step step')
```

In [3]:
```
# Do we just replace walk with slither?

class Reptile:
    def slither(self):
        print('slide slide slide')
```

In [4]:
```
reptile = Reptile()
reptile.walk()  # user code
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-811d5caa045b> in <module>
      1 reptile = Reptile()
----> 2 reptile.walk()  # user code

AttributeError: 'Reptile' object has no attribute 'walk'
```

Covenant 5. Delete Features Gently

Step 1. Introduce new API while keeping the old API in place.

In [5]:
```
class Reptile:
    def walk(self):
        print('step step step')

    def slither(self):
        print('slide slide slide')
```

Covenant 5. Delete Features Gently

Step 2. Mark old feature as deprecated.

In [6]:
```
import warnings


class Reptile:
    def walk(self):
        warnings.warn("walk is depracated, use slither",
                      DeprecationWarning, stacklevel=2)
        print('step step step')

    def slither(self):
        print('slide slide slide')
```

note: `stacklevel=2` tells users what line in their code they need to change

We are still not ready to delete the `slither` method though. We need to teach our users how to upgrade safely...

Covenant 6. Maintain a Change Log

Document new features and decrecations, and when features will be removed in the future.

Covenant 7. Choose a versioning scheme

Semantic versioning [semver.org] most popular - see PEP 440 (but sometimes time-based versioning makes sense).

- 1.0 First "stable" release
- 1.1 Add slither(), deprecate walk()
- 2.0 Delete walk()

Major releases are where breaking changes, like removing a feature, should occur.

Covenant 8. Write an Update Guide

Instructing users that they should upgrade to the last minor version before the new major version (the "bridge release"), and test their code by running it with `python -W::DeprecationWarning` , to make sure they are ready for the upgrade.

Adding/Changing Parameters

Covenant 9. Add Parameters Compatibly

In [7]:
```
def move(direction):
    print(f'slither {direction}')

# user's application
move('north')
```

```
slither north
```

Suppose you've added wings to your reptile, so now it can move by slithering *or* flying. We want to add a mode parameter to our function to indicate how it should move.

Covenant 9. Add Parameters Compatibly

In [8]:
```
def move(direction, mode):
    assert mode in ('slither', 'fly')
    print(f'{mode} {direction}')

# user's application
move('north')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-f158085caa36> in <module>
      4
      5 # user's application
----> 6 move('north')

TypeError: move() missing 1 required positional argument: 'mode'
```

Covenant 9. Add Parameters Compatibly

New parameters should be given defaults that preserve the old behavior.

In [9]:
```
def move(direction, mode='slither'):
    assert mode in ('slither', 'fly')
    print(f'{mode} {direction}')

# user's application
move('north')
```

```
slither north
```

Covenant 9. Add Parameters Compatibly

List new parameters in the order in which they were added... However there is still a risk because users *can* pass their parameters without specifying names.

In [ ]:
```
def move(direction,
         mode='slither',
         turbo=False,
         extra_sinuous=False,
         hail_lyft=False):
    # ...

# poorly written user application could be
move('north', 'slither', False, True)
```

Suppose `turbo` parameter is outdated and we want to delete it. User's code will still run but it will be doing something different than intended.

Explicit deprecation pattern

Will raise a warning if user is setting a parameter that will be deleted in the future. Then remove in next major release.

In [10]:
```
_turbo_default = object()

def move(direction,
         mode='slither',
         turbo=_turbo_default,
         extra_sinuous=False,
         hail_lyft=False):
    if turbo is not _turbo_default:
        warnings.warn('turbo is deprecated', DeprecationWarning, stacklevel=2)
    else:
        # the old default
        turbo = False
```

Require users to pass parameters by name

Even better approach if you can plan ahead and implement before you ever need to remove parameters.

In [11]:
```
def move(direction,
         *,
         mode='slither',
         turbo=False,
         extra_sinuous=False,
         hail_lyft=False):
    # ...
    return
```

All parameters after asterisk can only be passed by name (Python 3). Now if you delete parameters you know it will either raise an error for your user or will not cause any problems.

Require users to pass parameters by name

Even better approach if you can plan ahead and implement before you ever need to remove parameters.

In [12]:
```
def move(direction,
         *,
         mode='slither',
         turbo=False,
         extra_sinuous=False,
         hail_lyft=False):
    # ...
    return
```

In [13]:
```
move('northeast', mode='slither')
```

In [14]:
```
move('northeast', 'slither')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-42bec378ff2d> in <module>
----> 1 move('northeast', 'slither')

TypeError: move() takes 1 positional argument but 2 were given
```

Covenant 10. Change Behavior Gradually

Changing Behavior, without changing the API:

1. Add a flag to opt-in to new behavior (in minor release)
    - default False
    - warn if its False
2. Change default to True, deprecate flag entirely (next major release)
3. Remove the flag (in next major release after that)

API Evolution the Right Way

A. Jesse Jiryu Davis

- Read the written version in his blog: empti.ly/api-evolution (https://emptysqua.re/blog/api-evolution-the-right-way/)
- Watch the PyCon talk: https://www.youtube.com/watch?v=dqDnB6jKzcE (https://www.youtube.com/watch?v=dqDnB6jKzcE)

Escape from Auto-manual Testing with Hypothesis

Zac Hatfield-Dodds

Zac is a researcher at the Australian National University's 3A Institute, which is building a new applied science to 'manage the machines' - AI, cyber-physical systems, and other new technologies. He started using Python to analyse huge environmental datasets, and contributing to libraries like Xarray to make such analysis easier for all scientists. Now, as a maintainer of Hypothesis, Pytest, and Trio, Zac is still passionate about making it easy to write software you can understand and rely on. When not at a computer he can usually be found surrounded by books of all kinds, the Australian bush, or both.

`Hypothesis` is a Python package for **property based testing**: instead of testing exact inputs and outputs you describe the types of inputs that your function should accept, and assert things that should be true about the output.

https://hypothesis.works/ (https://hypothesis.works/)

Hypothesis will generate many examples of input data fitting your description, and run them all through your test, reporting back minimal (usually) failing examples.

In [15]:

```
from hypothesis import given, strategies as st


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

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

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

Testing Neural Networks

State of the art of testing NN is terrible. Advice:

- Use lots of assertions in your code
- check that model weights get updated after a training step
- check bounds on inputs/outputs of different steps
- check that the model converges when you expect

python `optimize` flag (or environment variable) lets you run your code completely ignoring assertions, so you can have them run only when you are testing and not everytime you are training a model.

```
$ python -o train.py
```

Print debugging

`note()` is like `print()` but will only print stuff in the final minimal failing test case.

Stateful Testing

The Water Jug Problem from Die Hard 3

Original example from Nicholas Chammas: http://nchammas.com/writing/how-not-to-die-hard-with-hypothesis (http://nchammas.com/writing/how-not-to-die-hard-with-hypothesis)

In the movie Die Hard with a Vengeance (aka Die Hard 3), there is a famous scene where John McClane (Bruce Willis) and Zeus Carver (Samuel L. Jackson) have to solve a problem or be blown up: **Given a 3 gallon jug and a 5 gallon jug, how do you measure out exactly 4 gallons of water?** https://www.youtube.com/watch?v=6cAbgAaEOVE (https://www.youtube.com/watch?v=6cAbgAaEOVE)

Stateful Testing

Given a 3 gallon jug and a 5 gallon jug, how do you measure out exactly 4 gallons of water?

We don't have a bomb handy, but we *can* make Hypothesis solve this for us! We just need to set up the state and possible actions... then we can claim that making random moves never leads to the "solved" state, and let Hypothesis find a counter-example. Fortunately, Hypothesis will also shrink what it finds to a minimal sequence of actions!

This pattern, where all the state lives on the `RuleBasedStateMachine`, is the easiest way to get started with stateful testing.

```python
# Given a 3 gallon jug and a 5 gallon jug, how do you measure
# out exactly 4 gallons of water?

from hypothesis import note, settings
from hypothesis.stateful import RuleBasedStateMachine, rule, invariant


@settings(max_examples=2000)
class DieHardProblem(RuleBasedStateMachine):
    small = 0
    big = 0

    @rule()
    def fill_small(self):
        self.small = 3

    @rule()
    def fill_big(self):
        self.big = 5

    @rule()
    def empty_small(self):
        self.small = 0

    @rule()
    def empty_big(self):
        self.big = 0

    @rule()
    def pour_small_into_big(self):
        old_big = self.big
        self.big = min(5, self.big + self.small)
        self.small = self.small - (self.big - old_big)

    @rule()
    def pour_big_into_small(self):
        old_small = self.small
        self.small = min(3, self.small + self.big)
        self.big = self.big - (self.small - old_small)

    @invariant()
    def physics_of_jugs(self):
        assert 0 <= self.small <= 3
        assert 0 <= self.big <= 5

    @invariant()
    def die_hard_problem_not_solved(self):
        note("> small: {s} big: {b}".format(s=self.small, b=self.big))
        assert self.big != 4


DieHardTest = DieHardProblem.TestCase
```

```
(test-demo) jes.ford@macbook:~/recap-pycon2019 $ pytest die_hard_example.py
============================================================ test session starts =============================================================
platform darwin -- Python 3.7.2, pytest-4.2.1, py-1.7.0, pluggy-0.8.1
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/Users/jes.ford/recap-pycon2019/.hypothesis/examples')
rootdir: /Users/jes.ford/recap-pycon2019, inifile:
plugins: hypothesis-4.17.2
collected 1 item

die_hard_example.py F                                                                                                                  [100%]

================================================================= FAILURES ==================================================================
_____________________________________________________ DieHardTest.runTest ______________________________________________________

self = <hypothesis.stateful.DieHardProblem.TestCase testMethod=runTest>

    def runTest(self):
>       run_state_machine_as_test(state_machine_class)

/anaconda3/envs/test-demo/lib/python3.7/site-packages/hypothesis/stateful.py:232:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
/anaconda3/envs/test-demo/lib/python3.7/site-packages/hypothesis/stateful.py:131: in run_state_machine_as_test
    run_state_machine(state_machine_factory)
/anaconda3/envs/test-demo/lib/python3.7/site-packages/hypothesis/stateful.py:89: in run_state_machine
    @given(st.data())
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = DieHardProblem({})

    @invariant()
    def die_hard_problem_not_solved(self):
        note("> small: {s} big: {b}".format(s=self.small, b=self.big))
>       assert self.big != 4
E       AssertionError: assert 4 != 4
E        +  where 4 = DieHardProblem({}).big

die_hard_example.py:50: AssertionError
---------------------------------------------------------- Hypothesis ----------------------------------------------------------
Falsifying example: run_state_machine(factory=DieHardProblem, data=data(...))
state = DieHardProblem()
> small: 0 big: 0
state.fill_big()
> small: 0 big: 5
state.pour_big_into_small()
> small: 3 big: 2
state.empty_small()
> small: 0 big: 2
state.pour_big_into_small()
> small: 2 big: 0
state.fill_big()
> small: 2 big: 5
state.pour_big_into_small()
> small: 3 big: 4
state.teardown()

You can reproduce this example by temporarily adding @reproduce_failure('4.17.2', b'AAECAQQBAQEEAQIBBA==') as a decorator on your test case
========================================================= 1 failed in 0.31 seconds ==========================================================
```

```
---------------------------------------------------------- Hypothesis ----------
Falsifying example: run_state_machine(factory=DieHardProblem, data=data(...))
state = DieHardProblem()
> small: 0 big: 0
state.fill_big()
> small: 0 big: 5
state.pour_big_into_small()
> small: 3 big: 2
state.empty_small()
> small: 0 big: 2
state.pour_big_into_small()
> small: 2 big: 0
state.fill_big()
> small: 2 big: 5
state.pour_big_into_small()
> small: 3 big: 4
state.teardown()
```

Escape from Auto-manual Testing with Hypothesis

Zac Hatfield-Dodds

- Watch the Pycon Talk: https://www.youtube.com/watch?v=KcyGUVzL7HA (https://www.youtube.com/watch?v=KcyGUVzL7HA)
- or the PyCon Tutorial (3 hours): https://www.youtube.com/watch?v=SmBAl34RV4M&list=PLPbTDk1hBo3xof51R8pk3kP1BVBuMYP9c&index=6&t=4 (https://www.youtube.com/watch?v=SmBAl34RV4M&list=PLPbTDk1hBo3xof51R8pk3kP1BVBuMYP9c&index=6&t=4
    - with supporting GitHub repo here: https://github.com/Zac-HD/escape-from-automanual-testing (https://github.com/Zac-HD/escape-from-automanual-testing)

PEP 572: The Walrus Operator

Dustin Ingram

Dustin is a Developer Advocate at Google, focused on supporting the Python community on the Google Cloud Platform. He's also a member of the Python Packaging Authority, maintainer of PyPI, and organizer for the PyTexas conference.

What is the Walrus Operator?

"Named Expression Operator" for assignment expressions using the newly defined `:=` operator, available in Python 3.8.

Example 1: Balancing Lines of Code vs complexity

In [ ]:
```
foo = [f(x), f(x)**2, f(x)**3]
```

If `f(x)` is expensive, we probably don't want to compute it three times. So we would do

In [ ]:
```
y = f(x)
foo = [y, y**2, y**3]
```

But this is one two lines, and maybe we don't like that...

With the walrus operator you can do this

In [ ]:
```
foo = [y := f(x), y**2, y**3]
```

Example 2: Avoiding Inefficient Comprehensions

In [ ]:
```
results = []
for x in data:
    result = f(x)
    if result:
        results.append(result)
```

In [ ]:
```
# probably nicer/cleaner in a list comprehension
result = [f(x) for x in data if f(x)]

# but this is more inefficient...
```

In [ ]:
```
# with the walrus operator we can have it both ways
result = [y for x in data if (y := f(x))]
```

Example 3: Unnecessary Variables in Scope

In [ ]:
```
# old way
chunk = file.read(8192)
while chunk:
    process(chunk)
    chunk = file.read(8192)
```

In [ ]:
```
# new way
while chunk := file.read(8192):
    process(chunk)
```

Reception of PEP 572

- This was a particularly controversial PEP, around May 2018
- polls, lengthly mailing list threads, discussions, arguments...
- eventually Guido Van Rossum accepted and merged this proposal in July 2018
    - *and then stepped down from his position as BDFL*

"Now that PEP 572 is done, I don't ever want to have to fight so hard for a PEP and find that so many people despise my decisions."

New Python Governance Model

Without a BDFL, there was no way to move forward with changes to the language.

- PEP 8000: Python Language Governance Proposal Overview
- PEP 8001: Python Governance Voting Process

... lots of proposals...

- PEP 8016: Steering Council Model was accepted
    - Elected 5 person committee who can accept or reject PEPs, plus some other limited powers.