Strategies For Testing Async Code

Neil Chazin

Who am I?

— Programming in Python since the early 2000s
— Believe that testing software is important
— @neilathotep

Work

— Currently Senior Software Engineer/Tech Lead at Agari
  — Agari secures digital communications to ensure humanity prevails over evil

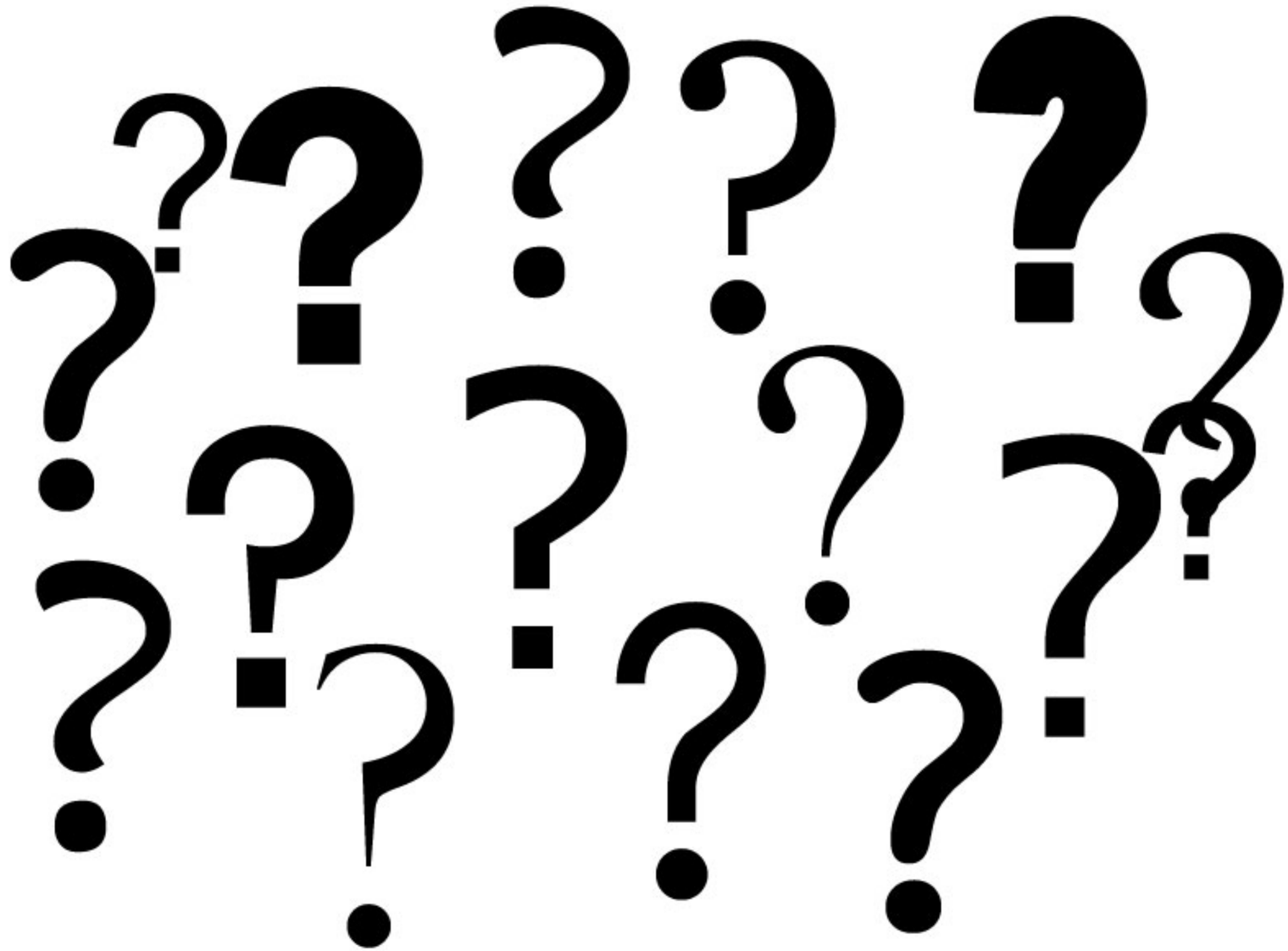
Brief Outline

1. Why?
2. Quick intro to async/`asyncio`
3. Challenges of testing
   — Some solutions I've found
4. Wrap up

Why?

— Testing is important

— Performance is often important



Async in Python

A very brief intro

neil chazin - @neilathotep

General Async Basics

- Concurrency through cooperation
  - yield control when 'waiting' - asynchronous results

asyncio

- framework for asynchronous computing
- available in stdlib as of 3.4
  - Improved syntax in 3.5
  - Incremental changes in 3.6 and 3.7

Two primary concepts

— Coroutines: perform asynchronous work

— Event loops: schedule asynchronous work

Key syntax

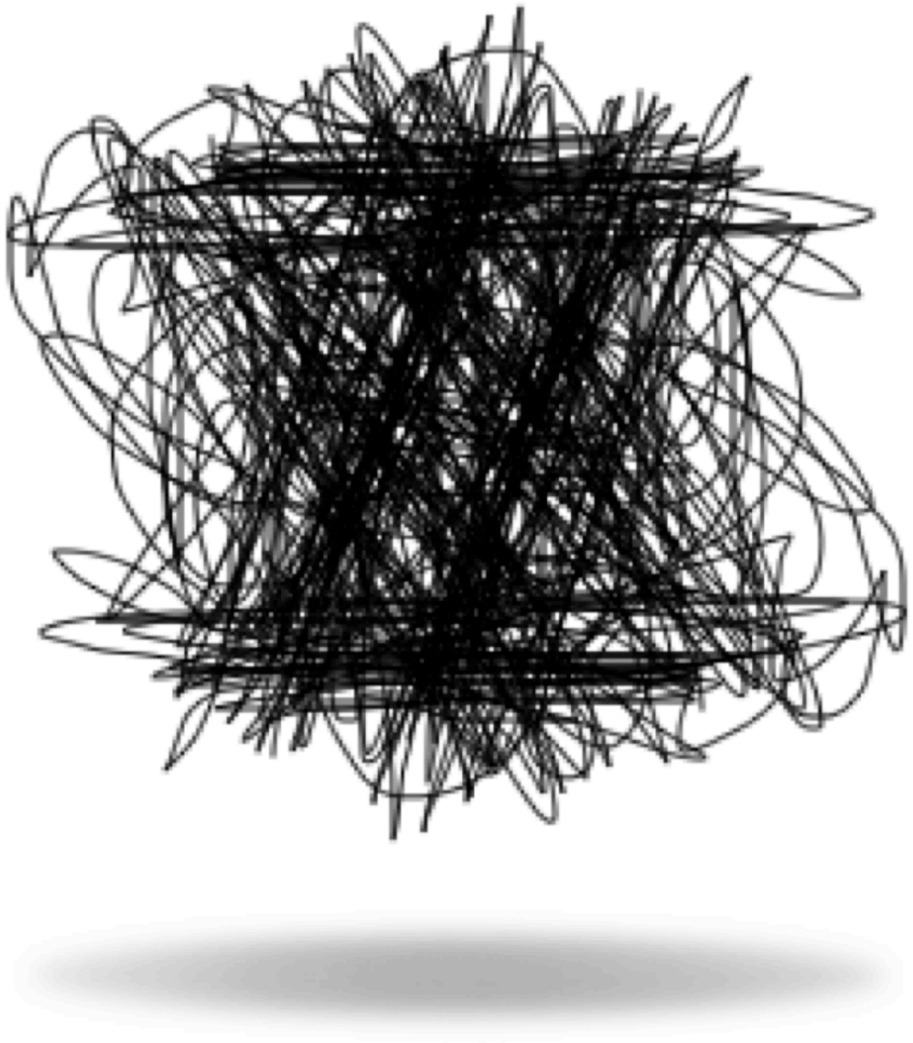
— `async` - define a coroutine

— `await` - 'call' a coroutine

```
1  async def my_coroutine():
2      return await another_coroutine()
```

Example use case

We need something simple...



```python
import asyncio
import random

class Cat:
    def __init__(self, name):
        self.name = name
        self.mood = DISPLEASED

    async def move(self, direction):
        if self.mood == PLEASED:
            await asyncio.sleep(random.uniform(0, 5))
            self.mood = DISPLEASED
            return True
        return False

    def pet(self):
        self.mood = PLEASED
```

```python
import asyncio
import random

class Cat:
    def __init__(self, name):
        self.name = name
        self.mood = DISPLEASED

    async def move(self, direction):
        if self.mood == PLEASED:
            await asyncio.sleep(random.uniform(0, 5))
            self.mood = DISPLEASED
            return True
        return False

    def pet(self):
        self.mood = PLEASED
```

```python
import asyncio
import random

class Cat:
    def __init__(self, name):
        self.name = name
        self.mood = DISPLEASED

    async def move(self, direction):
        if self.mood == PLEASED:
            await asyncio.sleep(random.uniform(0, 5))
            self.mood = DISPLEASED
            return True
        return False

    def pet(self):
        self.mood = PLEASED
```

Challenge

Testing a coroutine

Call as a function

```
1   from unittest import TestCase
2
3   import cat
4
5   async def herd(cat, direction):
6       cat.pet()
7       return await cat.move(direction)
8
9   class Tests(TestCase):
10
11      def test_forward(self):
12          garfield = cat.Cat('Garfield')
13          self.assertTrue(herd(garfield, 'forward'))
```

Call as a function

```
1   from unittest import TestCase
2
3   import cat
4
5   async def herd(cat, direction):
6       cat.pet()
7       return await cat.move(direction)
8
9   class Tests(TestCase):
10
11      def test_forward(self):
12          garfield = cat.Cat('Garfield')
13          self.assertTrue(herd(garfield, 'forward'))
```

Tests pass but...

```
(.env) nchazin:tests nchazin$ python -m unittest test_naive.py
/Users/nchazin/github/presentations/pycon2019/tests/test_naive.py:13
: RuntimeWarning: coroutine 'herd' was never awaited
  self.assertTrue(herd(garfield, 'forward'))
RuntimeWarning: Enable tracemalloc to get the object allocation trac
eback
.
----------------------------------------------------------------------
--
Ran 1 test in 0.148s

OK
```

Tests pass but...

```
(.env) nchazin:tests nchazin$ python -m unittest test_naive.py
/Users/nchazin/github/presentations/pycon2019/tests/test_naive.py:13
: RuntimeWarning: coroutine 'herd' was never awaited
  self.assertTrue(herd(garfield, 'forward'))
RuntimeWarning: Enable tracemalloc to get the object allocation trac
eback
.
----------------------------------------------------------------------
--
Ran 1 test in 0.148s

OK
```

Await the coroutine

```
1   from unittest import TestCase
2
3   import cat
4
5   async def herd(cat, direction):
6       cat.pet()
7       return await cat.move(direction)
8
9
10  class Tests(TestCase):
11
12      def test_forward(self):
13          garfield = cat.Cat('Garfield')
14          result = await herd(garfield, 'forward')
15          self.assertTrue(result)
```

SyntaxError

```
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.7/lib/python
3.7/runpy.py", line 193, in _run_module_as_main
    "__main__", mod_spec)
...
  File "/Users/nchazin/github/presentations/pycon2019/tests/test_awa
it.py", line 14
    result = await herd(garfield, 'forward')
             ^
SyntaxError: 'await' outside async function
```

Run in an event loop

```
1   import asyncio
2   from unittest import TestCase
3
4   import cat
5
6   async def herd(cat, direction):
7       cat.pet()
8       return await cat.move(direction)
9
10
11  class Tests(TestCase):
12
13      def test_forward(self):
14          garfield = cat.Cat('Garfield')
15          loop = asyncio.get_event_loop()
16          result = loop.run_until_complete(herd(garfield, 'forward'))
17          loop.close()
18          self.assertTrue(result)
```

Run in an event loop

```
1   import asyncio
2   from unittest import TestCase
3
4   import cat
5
6   async def herd(cat, direction):
7       cat.pet()
8       return await cat.move(direction)
9
10
11  class Tests(TestCase):
12
13      def test_forward(self):
14          garfield = cat.Cat('Garfield')
15          loop = asyncio.get_event_loop()
16          result = loop.run_until_complete(herd(garfield, 'forward'))
17          loop.close()
18          self.assertTrue(result)
```

Success

```
(.env) nchazin:tests nchazin$ python -m unittest test_run_until_comp
lete.py
.
----------------------------------------------------------------------
Ran 1 test in 2.368s

OK
```

Use pytest-asyncio

— pip install pytest-asyncio

```python
import pytest

import cat


async def herd(cat, direction):
    cat.pet()
    return await cat.move(direction)


@pytest.mark.asyncio
async def test_forward():
    garfield = cat.Cat('Garfield')
    result = await herd(garfield, 'forward')
    assert result
```

```python
import pytest

import cat


async def herd(cat, direction):
    cat.pet()
    return await cat.move(direction)


@pytest.mark.asyncio
async def test_forward():
    garfield = cat.Cat('Garfield')
    result = await herd(garfield, 'forward')
    assert result
```

```python
import pytest

import cat


async def herd(cat, direction):
    cat.pet()
    return await cat.move(direction)


@pytest.mark.asyncio
async def test_forward():
    garfield = cat.Cat('Garfield')
    result = await herd(garfield, 'forward')
    assert result
```

```
(.env) nchazin:tests nchazin$ PYTHONPATH=. pytest test_pytest.py
============================= test session starts ==============================
platform darwin -- Python 3.7.2, pytest-4.4.1, py-1.8.0, pluggy-0.9.
0
rootdir: /Users/nchazin/github/presentations/pycon2019/tests
plugins: asyncio-0.10.0
collected 1 item

test_pytest.py .                                                         [100%]

========================== 1 passed in 0.56 seconds ===========================
```

Python 3.7

New function `asyncio.run()` simplifies things

```python
import asyncio
from unittest import TestCase

import cat

async def herd(cat, direction):
    cat.pet()
    return await cat.move(direction)


class Tests(TestCase):

    def test_forward(self):
        garfield = cat.Cat('Garfield')
        result = asyncio.run(herd(garfield, 'forward'))
        self.assertTrue(result)
```

Challenge

Mocks and patching

Mocks often relied upon

— Unittest.mock.Mock won't mock a coroutine

```python
import asyncio
from unittest import TestCase
from unittest.mock import Mock, patch

import cat

async def herd(cat, direction):
    cat.pet()
    return await cat.move(direction)


class Tests(TestCase):

    @patch('cat.Cat.move')
    def test_forward(self, move_mock):
        garfield = cat.Cat('Garfield')
        loop = asyncio.get_event_loop()
        result = loop.run_until_complete(herd(garfield, 'forward'))
        move_mock.assert_called_with('forward')
```

```
1   import asyncio
2   from unittest import TestCase
3   from unittest.mock import Mock, patch
4
5   import cat
6
7   async def herd(cat, direction):
8       cat.pet()
9       return await cat.move(direction)
10
11
12  class Tests(TestCase):
13
14      @patch('cat.Cat.move')
15      def test_forward(self, move_mock):
16          garfield = cat.Cat('Garfield')
17          loop = asyncio.get_event_loop()
18          result = loop.run_until_complete(herd(garfield, 'forward'))
19          move_mock.assert_called_with('forward')
```

```
TypeError: object MagicMock can't be used in 'await' expression
----------------------------------------------------------------------
Ran 1 test in 0.128s
FAILED (errors=1)
```

```python
import asyncio
from unittest import TestCase
from unittest.mock import MagicMock, patch

import cat


class AsyncMock(MagicMock):
    async def __call__(self, *args, **kwargs):
        return super().__call__(*args, **kwargs)


class Tests(TestCase):

    @patch('cat.Cat.move', new_callable=AsyncMock)
    def test_forward(self, move_mock):
        garfield = cat.Cat('Garfield')
        loop = asyncio.get_event_loop()
        result = loop.run_until_complete(herd(garfield, 'forward'))
        move_mock.assert_called_with('forward')
```

```python
import asyncio
from unittest import TestCase
from unittest.mock import MagicMock, patch

import cat


class AsyncMock(MagicMock):
    async def __call__(self, *args, **kwargs):
        return super().__call__(*args, **kwargs)


class Tests(TestCase):

    @patch('cat.Cat.move', new_callable=AsyncMock)
    def test_forward(self, move_mock):
        garfield = cat.Cat('Garfield')
        loop = asyncio.get_event_loop()
        result = loop.run_until_complete(herd(garfield, 'forward'))
        move_mock.assert_called_with('forward')
```

Challenge

Context managers

```
1  class AsyncContextManager(mock.MagicMock):
2      async def __aenter__(self, *args, **kwargs):
3          return self.__enter__(*args, **kwargs)
4
5      async def __aexit__(self, *args, **kwargs):
6          return self.__exit__(*args, **kwargs)
```

Challenge

More complex use cases

- Previous patterns all work great but some short comings I've found:
  - test coroutines that do not run in the `MainThread` when executing
  - test syncronous functions that call into event loops ()
  - functional tests
  - Event Loop variants (e.g. uvloop)

```python
import asyncio
import threading

class LoopRunner(threading.Thread):

    def __init__(self, loop):
        threading.Thread.__init__(self, name='runner')
        self.loop = loop

    def run(self):
        asyncio.set_event_loop(self.loop)
        try:
            self.loop.run_forever()
        finally:
            if self.loop.is_running():
                self.loop.close()

    def run_coroutine(self, coroutine):
        result = asyncio.run_coroutine_threadsafe(coroutine, self.loop)
        return result.result()
```

```python
import asyncio
import threading

class LoopRunner(threading.Thread):

    def __init__(self, loop):
        threading.Thread.__init__(self, name='runner')
        self.loop = loop

    def run(self):
        asyncio.set_event_loop(self.loop)
        try:
            self.loop.run_forever()
        finally:
            if self.loop.is_running():
                self.loop.close()

    def run_coroutine(self, coroutine):
        result = asyncio.run_coroutine_threadsafe(coroutine, self.loop)
        return result.result()
```

```python
import asyncio
from unittest import TestCase

import cat
from loop_runner import LoopRunner


class Tests(TestCase):

    def setUp(self):
        self.runner = LoopRunner(asyncio.new_event_loop())
        self.runner.start()

    def tearDown(self):
        self.runner.stop()
        self.runner.join()

    def test_forward(self):
        garfield = cat.Cat('Garfield')
        result = self.runner.run_coroutine(herd(garfield, 'forward'))
        self.assertTrue(result)
```

```python
import asyncio
from unittest import TestCase

import cat
from loop_runner import LoopRunner


class Tests(TestCase):

    def setUp(self):
        self.runner = LoopRunner(asyncio.new_event_loop())
        self.runner.start()

    def tearDown(self):
        self.runner.stop()
        self.runner.join()

    def test_forward(self):
        garfield = cat.Cat('Garfield')
        result = self.runner.run_coroutine(herd(garfield, 'forward'))
        self.assertTrue(result)
```

Calling across threads

```python
import asyncio
import threading

class HerdRouter(threading.Thread):

    def __init__(self, loop):
        self._loop = loop
        self._rlock = threading.RLock()
        self.commands = set()

    # thread safe
    async def _add_command(self, key):
        with self._rlock:
            self.commands.add(key)
        return True

    # actual work is done in a thread safe manner
    def add_command(self, key):
        result = asyncio.run_coroutine_threadsafe(self._add_command(key), self._loop)
        result.result()
```

```python
import asyncio
import threading

class HerdRouter(threading.Thread):

    def __init__(self, loop):
        self._loop = loop
        self._rlock = threading.RLock()
        self.commands = set()

    # thread safe
    async def _add_command(self, key):
        with self._rlock:
            self.commands.add(key)
        return True

    # actual work is done in a thread safe manner
    def add_command(self, key):
        result = asyncio.run_coroutine_threadsafe(self._add_command(key), self._loop)
        result.result()
```

```python
import asyncio
import threading

class HerdRouter(threading.Thread):

    def __init__(self, loop):
        self._loop = loop
        self._rlock = threading.RLock()
        self.commands = set()

    # thread safe
    async def _add_command(self, key):
        with self._rlock:
            self.commands.add(key)
        return True

    # actual work is done in a thread safe manner
    def add_command(self, key):
        result = asyncio.run_coroutine_threadsafe(self._add_command(key), self._loop)
        result.result()
```

```python
import asyncio
from unittest import TestCase

import herd_router
from loop_runner import LoopRunner


class Tests(TestCase):

    def setUp(self):
        self.loop = asyncio.new_event_loop()
        self.herd_router = herd_router.HerdRouter(self.loop)
        self.runner = LoopRunner(self.loop)
        self.runner.start()

    def tearDown(self):...

    def test_command(self):
        self.herd_router.add_command('test')
        assert 'test' in self.herd_router.commands
```

```python
import asyncio
from unittest import TestCase

import herd_router
from loop_runner import LoopRunner


class Tests(TestCase):

    def setUp(self):
        self.loop = asyncio.new_event_loop()
        self.herd_router = herd_router.HerdRouter(self.loop)
        self.runner = LoopRunner(self.loop)
        self.runner.start()

    def tearDown(self):...

    def test_command(self):
        self.herd_router.add_command('test')
        assert 'test' in self.herd_router.commands
```

```python
import asyncio
from unittest import TestCase

import herd_router
from loop_runner import LoopRunner


class Tests(TestCase):

    def setUp(self):
        self.loop = asyncio.new_event_loop()
        self.herd_router = herd_router.HerdRouter(self.loop)
        self.runner = LoopRunner(self.loop)
        self.runner.start()

    def tearDown(self):...

    def test_command(self):
        self.herd_router.add_command('test')
        assert 'test' in self.herd_router.commands
```

Wrap up

— Try out `asyncio` programming
— Keep testing
— Find sample code and presentation at:

http://bit.ly/nchazin-pycon2019