

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

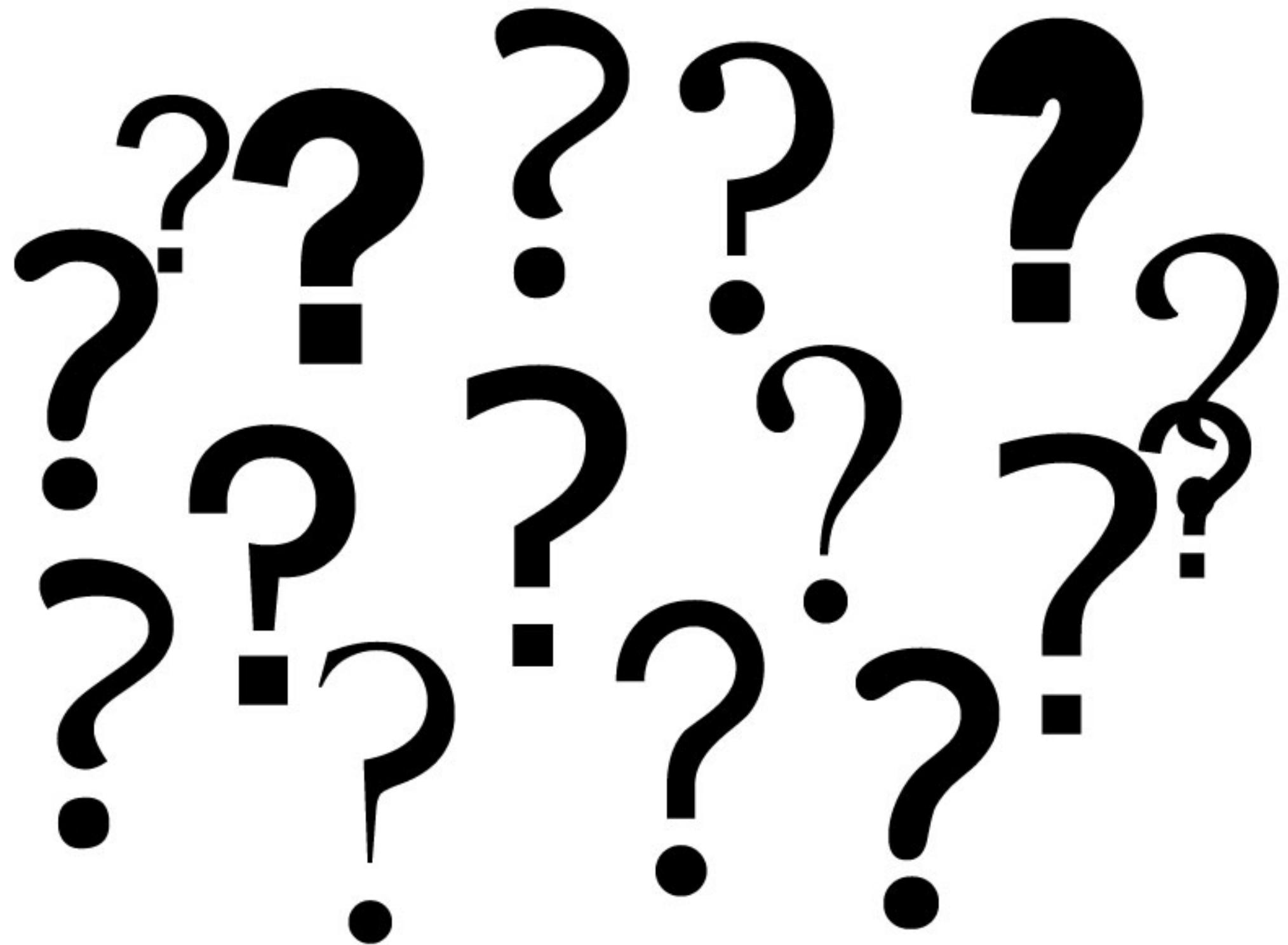


Brief Outline

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

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()
```

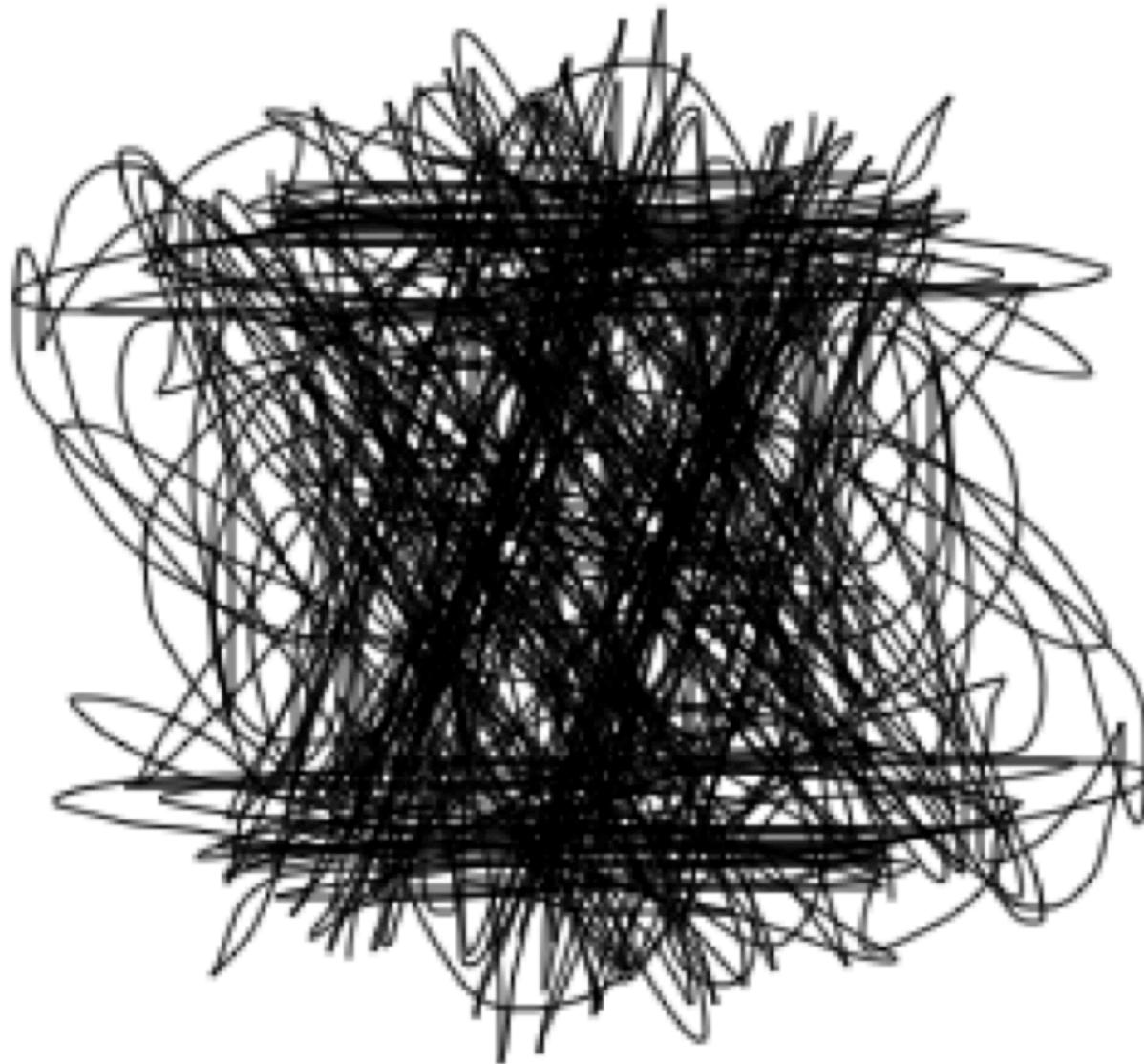
Example use case

We need something simple...



neil chazin - @neilathotep





neil chazin - @neilathotep



```
1  import asyncio
2  import random
3
4  class Cat:
5      def __init__(self, name):
6          self.name = name
7          self.mood = DISPLEASED
8
9      async def move(self, direction):
10         if self.mood == PLEASED:
11             await asyncio.sleep(random.uniform(0, 5))
12             self.mood = DISPLEASED
13         return True
14
15     def pet(self):
16         self.mood = PLEASED
```

```
1  import asyncio
2  import random
3
4  class Cat:
5      def __init__(self, name):
6          self.name = name
7          self.mood = DISPLEASED
8
9      async def move(self, direction):
10         if self.mood == PLEASED:
11             await asyncio.sleep(random.uniform(0, 5))
12             self.mood = DISPLEASED
13         return True
14
15     def pet(self):
16         self.mood = PLEASED
```

```
1 import asyncio
2 import random
3
4 class Cat:
5     def __init__(self, name):
6         self.name = name
7         self.mood = DISPLEASED
8
9     async def move(self, direction):
10        if self.mood == PLEASED:
11            await asyncio.sleep(random.uniform(0, 5))
12            self.mood = DISPLEASED
13            return True
14        return False
15
16    def pet(self):
17        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
```

```
0K
```

Use pytest-asyncio

— pip install pytest-asyncio

```
1 import pytest
2
3 import cat
4
5
6 async def herd(cat, direction):
7     cat.pet()
8     return await cat.move(direction)
9
10
11 @pytest.mark.asyncio
12 async def test_forward():
13     garfield = cat.Cat('Garfield')
14     result = await herd(garfield, 'forward')
15     assert result
```

```
1 import pytest
2
3 import cat
4
5
6 async def herd(cat, direction):
7     cat.pet()
8     return await cat.move(direction)
9
10
11 @pytest.mark.asyncio
12 async def test_forward():
13     garfield = cat.Cat('Garfield')
14     result = await herd(garfield, 'forward')
15     assert result
```

```
1 import pytest
2
3 import cat
4
5
6 async def herd(cat, direction):
7     cat.pet()
8     return await cat.move(direction)
9
10
11 @pytest.mark.asyncio
12
13 async def test_forward():
14     garfield = cat.Cat('Garfield')
15     result = await herd(garfield, 'forward')
16
17     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

```
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         result = asyncio.run(herd(garfield, 'forward'))
16         self.assertTrue(result)
```



challenge

Mocks and patching

Mocks often relied upon

- `Unittest.mock.Mock` won't mock a coroutine

```
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')
```

```
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)

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

```
1 import asyncio
2 from unittest import TestCase
3 from unittest.mock import MagicMock, patch
4
5 import cat
6
7
8 class AsyncMock(MagicMock):
9     async def __call__(self, *args, **kwargs):
10         return super().__call__(*args, **kwargs)
11
12
13 class Tests(TestCase):
14
15     @patch('cat.Cat.move', new_callable=AsyncMock)
16     def test_forward(self, move_mock):
17         garfield = cat.Cat('Garfield')
18         loop = asyncio.get_event_loop()
19         result = loop.run_until_complete(herd(garfield, 'forward'))
20         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 synchronous functions that call into event loops ()
 - functional tests
 - Event Loop variants (e.g. uvloop)

```
1 import asyncio
2 import threading
3
4 class LoopRunner(threading.Thread):
5
6     def __init__(self, loop):
7         threading.Thread.__init__(self, name='runner')
8         self.loop = loop
9
10    def run(self):
11        asyncio.set_event_loop(self.loop)
12        try:
13            self.loop.run_forever()
14        finally:
15            if self.loop.is_running():
16                self.loop.close()
17
18    def run_coroutine(self, coroutine):
19        result = asyncio.run_coroutine_threadsafe(coroutine, self.loop)
20        return result.result()
```

```
1 import asyncio
2 import threading
3
4 class LoopRunner(threading.Thread):
5
6     def __init__(self, loop):
7         threading.Thread.__init__(self, name='runner')
8         self.loop = loop
9
10    def run(self):
11        asyncio.set_event_loop(self.loop)
12        try:
13            self.loop.run_forever()
14        finally:
15            if self.loop.is_running():
16                self.loop.close()
17
18    def run_coroutine(self, coroutine):
19        result = asyncio.run_coroutine_threadsafe(coroutine, self.loop)
20        return result.result()
```

```
1 import asyncio
2 from unittest import TestCase
3
4 import cat
5 from loop_runner import LoopRunner
6
7
8 class Tests(TestCase):
9
10    def setUp(self):
11        self.runner = LoopRunner(asyncio.new_event_loop())
12        self.runner.start()
13
14    def tearDown(self):
15        self.runner.stop()
16        self.runner.join()
17
18    def test_forward(self):
19        garfield = cat.Cat('Garfield')
20        result = self.runner.run_coroutine(herd(garfield, 'forward'))
21        self.assertTrue(result)
```



```
1  import asyncio
2  from unittest import TestCase
3
4  import cat
5  from loop_runner import LoopRunner
6
7
8  class Tests(TestCase):
9
10 def setUp(self):
11     self.runner = LoopRunner(asyncio.new_event_loop())
12     self.runner.start()
13
14 def tearDown(self):
15     self.runner.stop()
16     self.runner.join()
17
18 def test_forward(self):
19     garfield = cat.Cat('Garfield')
20     result = self.runner.run_coroutine(herd(garfield, 'forward'))
21     self.assertTrue(result)
```



Calling across threads

```
1 import asyncio
2 import threading
3
4 class HerdRouter(threading.Thread):
5
6     def __init__(self, loop):
7         self._loop = loop
8         self._rlock = threading.RLock()
9         self.commands = set()
10
11     # thread safe
12     @async def _add_command(self, key):
13         with self._rlock:
14             self.commands.add(key)
15         return True
16
17     # actual work is done in a thread safe manner
18     def add_command(self, key):
19         result = asyncio.run_coroutine_threadsafe(self._add_command(key), self._loop)
20         result.result()
```



```
1 import asyncio
2 import threading
3
4 class HerdRouter(threading.Thread):
5
6     def __init__(self, loop):
7         self._loop = loop
8         self._rlock = threading.RLock()
9         self.commands = set()
10
11     # thread safe
12     @async def _add_command(self, key):
13         with self._rlock:
14             self.commands.add(key)
15         return True
16
17     # actual work is done in a thread safe manner
18     def add_command(self, key):
19         result = asyncio.run_coroutine_threadsafe(self._add_command(key), self._loop)
20         result.result()
```



```
1 import asyncio
2 import threading
3
4 class HerdRouter(threading.Thread):
5
6     def __init__(self, loop):
7         self._loop = loop
8         self._rlock = threading.RLock()
9         self.commands = set()
10
11     # thread safe
12     @async def _add_command(self, key):
13         with self._rlock:
14             self.commands.add(key)
15         return True
16
17     # actual work is done in a thread safe manner
18     def add_command(self, key):
19         result = asyncio.run_coroutine_threadsafe(self._add_command(key), self._loop)
20         result.result()
```



```
1  import asyncio
2  from unittest import TestCase
3
4  import herd_router
5  from loop_runner import LoopRunner
6
7
8  class Tests(TestCase):
9
10 def setUp(self):
11     self.loop = asyncio.new_event_loop()
12     self.herd_router = herd_router.HerdRouter(self.loop)
13     self.runner = LoopRunner(self.loop)
14     self.runner.start()
15
16 def tearDown(self):...
17
18
19 def test_command(self):
20     self.herd_router.add_command('test')
21     assert 'test' in self.herd_router.commands
```

```
1  import asyncio
2  from unittest import TestCase
3
4  import herd_router
5  from loop_runner import LoopRunner
6
7
8  class Tests(TestCase):
9
10 def setUp(self):
11     self.loop = asyncio.new_event_loop()
12     self.herd_router = herd_router.HerdRouter(self.loop)
13     self.runner = LoopRunner(self.loop)
14     self.runner.start()
15
16 def tearDown(self):...
17
18
19 def test_command(self):
20     self.herd_router.add_command('test')
21     assert 'test' in self.herd_router.commands
```

```
1  import asyncio
2  from unittest import TestCase
3
4  import herd_router
5  from loop_runner import LoopRunner
6
7
8  class Tests(TestCase):
9
10 def setUp(self):
11     self.loop = asyncio.new_event_loop()
12     self.herd_router = herd_router.HerdRouter(self.loop)
13     self.runner = LoopRunner(self.loop)
14     self.runner.start()
15
16 def tearDown(self):...
17
18
19
20 def test_command(self):
21     self.herd_router.add_command('test')
22     assert 'test' in self.herd_router.commands
```

Wrap up

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

<https://bit.ly/nchazin-pycon2019>