Make You an Async!

For Great Good?

@Judy2k

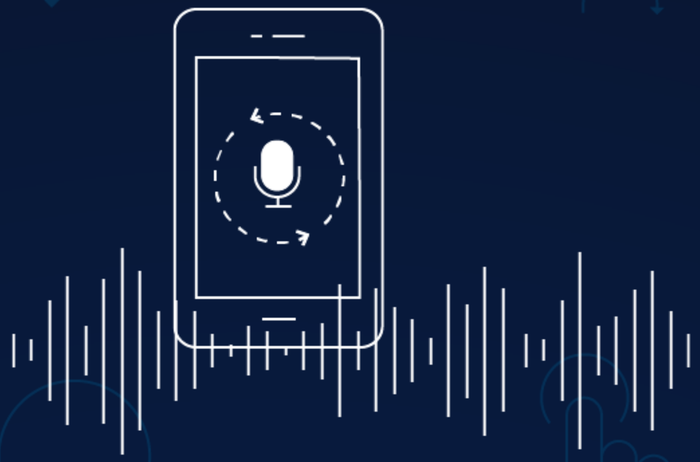
Mark Smith

Developer Advocate

Nexmo

nexmo®

The Vonage® API Platform



Non-Agenda

Non-Agenda

- Thread safety

Non-Agenda

- Thread safety
- Exception handling

Non-Agenda

- Thread safety
- Exception handling
- No cancellation of futures

Non-Agenda

- Thread safety
- Exception handling
- No cancellation of futures
- No validation

Non-Agenda

- Thread safety
- Exception handling
- No cancellation of futures
- No validation
- No stack reconstruction

Non-Agenda

- Thread safety
- Exception handling
- No cancellation of futures
- No validation
- No stack reconstruction
- No debugging

Non-Agenda

- Thread safety
- Exception handling
- No cancellation of futures
- No validation
- No stack reconstruction
- No debugging
- No clever scheduling

Non-Agenda

- Thread safety
- Exception handling
- No cancellation of futures
- No validation
- No stack reconstruction
- No debugging
- No clever scheduling
- No I/O code

Asyncio

Asyncio

Recap

Getting URLs in Parallel

```
async def main():
    await asyncio.gather(*[fetch(url) for url in [
        'http://python.org',
        'http://www.yahoo.com',
        'http://www.google.com',
        'http://www.nexmo.com',
    ]])

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Result

```
Fetching: http://python.org
Fetching: http://www.yahoo.com
Fetching: http://www.google.com
Fetching: http://www.nexmo.com
Received: http://www.google.com
Received: http://python.org
Received: http://www.nexmo.com
Received: http://www.yahoo.com
```

Python 3.4

```
@asyncio.coroutine
def main():
    yield from asyncio.gather(*[fetch(url) for url in [
        'http://python.org',
        'http://www.yahoo.com',
        'http://www.google.com',
        'http://www.nexmo.com',
    ]])


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

mysyncio

mysyncio

A Simple Event Loop

```
class MyEventLoop:
    def __init__(self):
        self._ready = collections.deque()
        self._stopping = False

    def call_soon(self, callback):
        self._ready.append(callback)

    ...
```

A Simple Event Loop

```
class MyEventLoop:
    def __init__(self):
        self._ready = collections.deque()
        self._stopping = False

    def call_soon(self, callback):
        self._ready.append(callback)

    ...
```

A Simple Event Loop

```
class MyEventLoop:
    def __init__(self):
        self._ready = collections.deque()
        self._stopping = False

    def call_soon(self, callback):
        self._ready.append(callback)

    ...
```

A Simple Event Loop

```
class MyEventLoop:
    def __init__(self):
        self._ready = collections.deque()
        self._stopping = False

    def call_soon(self, callback):
        self._ready.append(callback)

    ...
```

A Simple Event Loop

```
class MyEventLoop:
    ...
    def run_forever(self):
        while True:
            self._run_once()
            if self._stopping:
                break

    def _run_once(self):
        while self._ready:
            callback = self._ready.popleft()
            callback()
            time.sleep(0.1)
```

A Simple Event Loop

```
class MyEventLoop:
    ...
    def run_forever(self):
        while True:
            self._run_once()
            if self._stopping:
                break

    def _run_once(self):
        while self._ready:
            callback = self._ready.popleft()
            callback()
            time.sleep(0.1)
```

A Simple Event Loop

```
class MyEventLoop:
    ...
    def run_forever(self):
        while True:
            self._run_once()
            if self._stopping:
                break

    def _run_once(self):
        while self._ready:
            callback = self._ready.popleft()
            callback()
            time.sleep(0.1)
```

A Simple Event Loop

```
class MyEventLoop:
    ...
    def run_forever(self):
        while True:
            self._run_once()
            if self._stopping:
                break

    def _run_once(self):
        while self._ready:
            callback = self._ready.popleft()
            callback()
            time.sleep(0.1)
```

A Simple Event Loop

```
class MyEventLoop:
    ...
    def run_forever(self):
        while True:
            self._run_once()
            if self._stopping:
                break

    def _run_once(self):
        while self._ready:
            callback = self._ready.popleft()
            callback()
            time.sleep(0.1)
```

A Simple Event Loop

```
class MyEventLoop:
    ...
    def run_forever(self):
        while True:
            self._run_once()
            if self._stopping:
                break

    def _run_once(self):
        while self._ready:
            callback = self._ready.popleft()
            callback()
            time.sleep(0.1)
```

A Simple Event Loop

```
class MyEventLoop:
    ...
    def run_forever(self):
        while True:
            self._run_once()
            if self._stopping:
                break

    def _run_once(self):
        while self._ready:
            callback = self._ready.popleft()
            callback()
            time.sleep(0.1)
```

A Simple Event Loop

```
class MyEventLoop:
    ...
    def run_forever(self):
        while True:
            self._run_once()
            if self._stopping:
                break

    def _run_once(self):
        while self._ready:
            callback = self._ready.popleft()
            callback()
            time.sleep(0.1)
```

A Simple Event Loop

```
class MyEventLoop:
    ...
    def stop(self):
        self._stopping = True
```

A Simple Event Loop

```
class MyEventLoop:
    def __init__(self):
        self._ready = collections.deque()
        self._stopping = False

    def stop(self):
        self._stopping = True

    def call_soon(self, callback):
        self._ready.append(callback)

    def run_forever(self):
        while True:
            self._run_once()
            if self._stopping:
                break

    def _run_once(self):
        while self._ready:
            callback = self._ready.popleft()
            callback()
            time.sleep(0.1)
```

Hello ... World!

```
import mysyncio

def hello():
    print("Hello...")

def world():
    print("... World!")

loop = mysyncio.MyEventLoop()
loop.call_soon(hello)
loop.call_soon(world)
loop.run_forever()
```

Hello ... World!

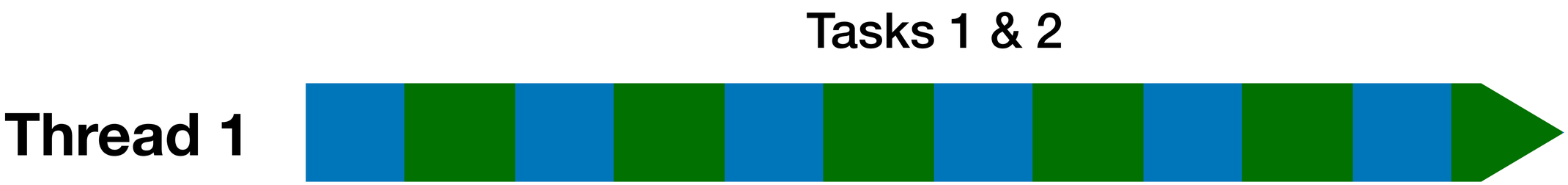
```
import mysyncio

def hello():
    print("Hello...")

def world():
    print("... World!")

loop = mysyncio.MyEventLoop()
loop.call_soon(hello)
loop.call_soon(world)
loop.run_forever()
```

```
Hello...
...World!
```

Threads & Processes

Single Threaded

Cooperative Multitasking



Running a Function

```
def hello():
    print("Hello")
    print("Goodbye")
    return "Result"
```

```
result = hello()
print("Result:", result)
```

Running It

```
Hello
Goodbye
Result: Result
```

Driving a Coroutine

```
def hello():
    print("Hello")
    yield
    print("Goodbye")
    return "Result"
```

```
try:
    gen = hello()
    next(gen)
    print(".")
    next(gen)
    print("Unreachable")
except StopIteration as result:
    print(
        "Result:", result.value)
```

Driving a Coroutine

```
def hello():
    print("Hello")
    yield
    print("Goodbye")
    return "Result"
```

```
try:
    gen = hello()
    next(gen)
    print(".")
    next(gen)
    print("Unreachable")
except StopIteration as result:
    print(
        "Result:", result.value)
```

Driving a Coroutine

```
def hello():
    print("Hello")
    yield
    print("Goodbye")
    return "Result"
```

```
try:
    gen = hello()
    next(gen)
    print(".")
    next(gen)
    print("Unreachable")
except StopIteration as result:
    print(
        "Result:", result.value)
```

Driving a Coroutine

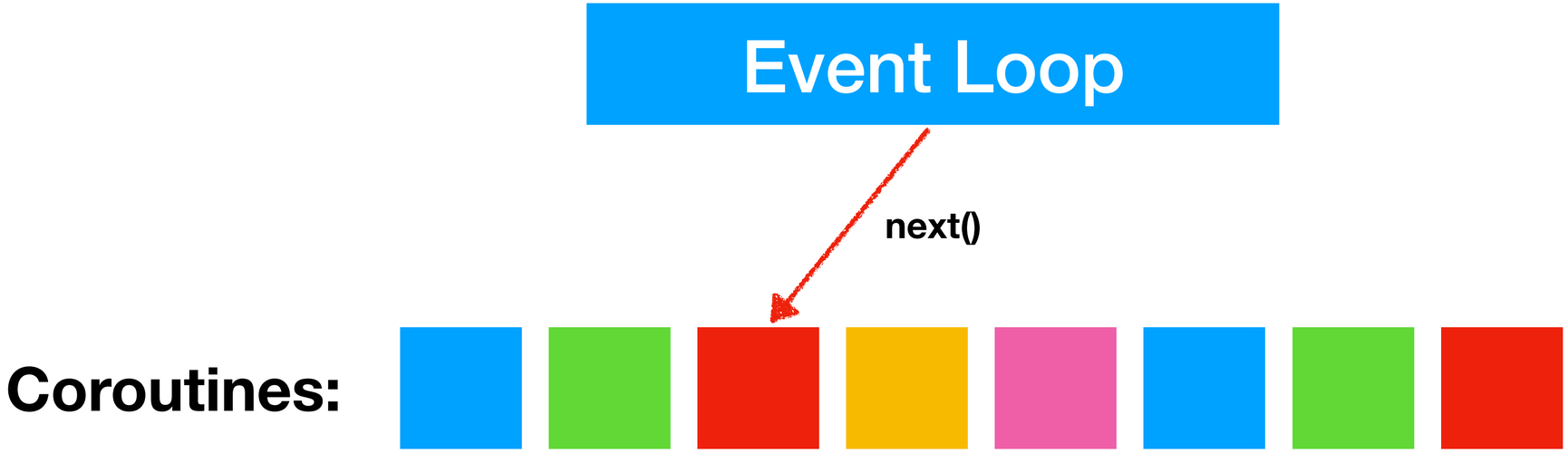
```
def hello():
    print("Hello")
    yield
    print("Goodbye")
    return "Result"
```

```
try:
    gen = hello()
    next(gen)
    print(".")
    next(gen)
    print("Unreachable")
except StopIteration as result:
    print(
        "Result:", result.value)
```

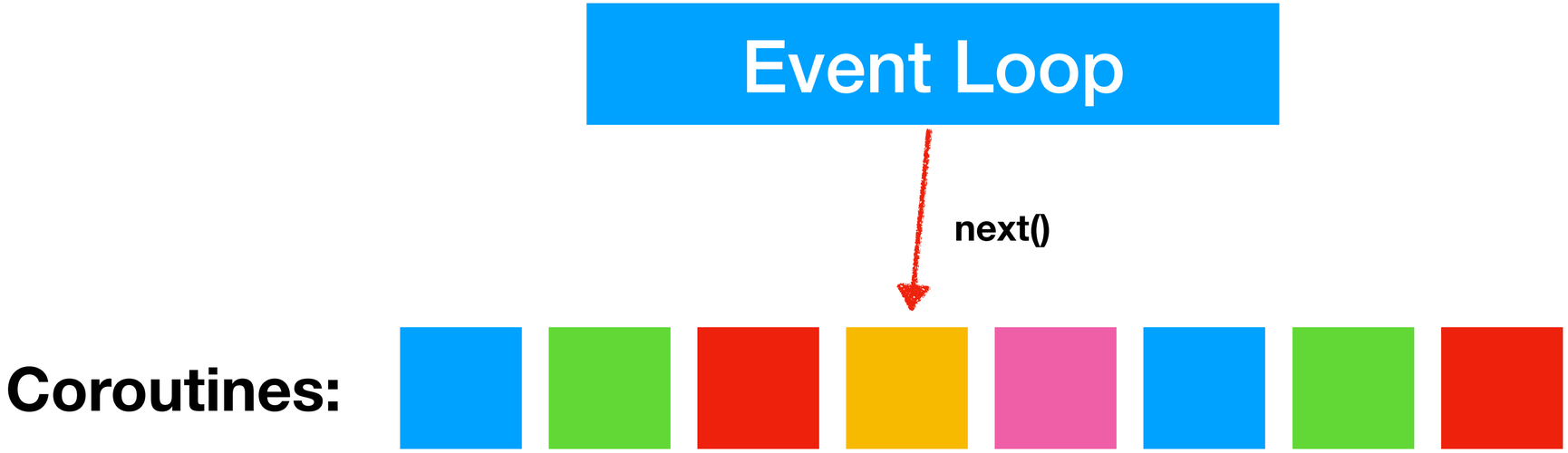
Running It

```
Hello
.
Goodbye
Result: Result
```

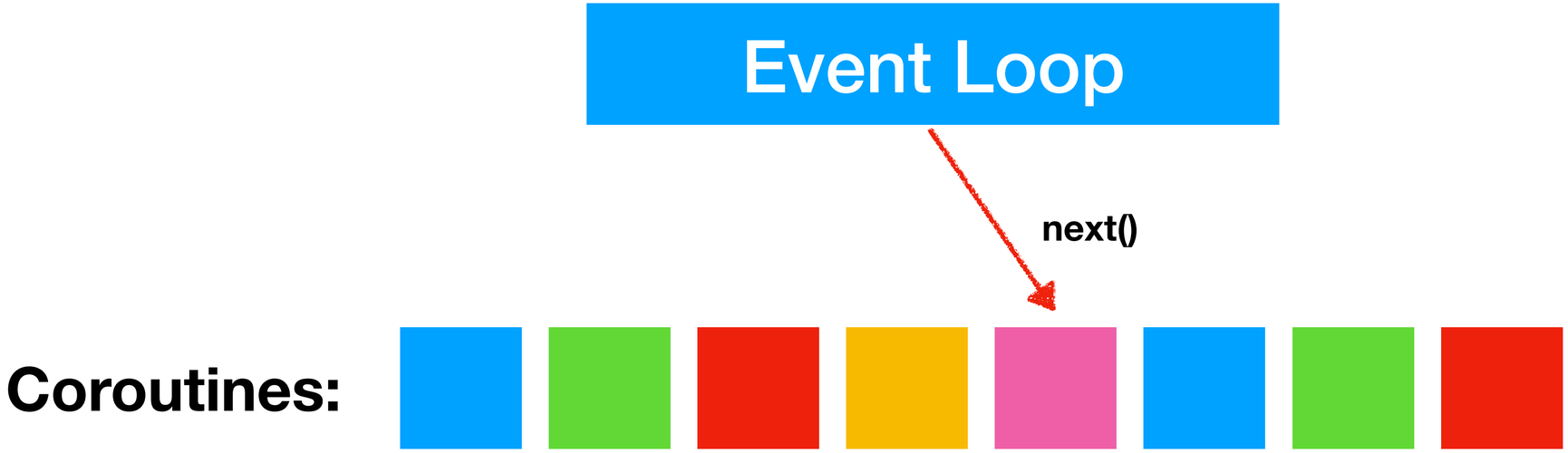
Cooperative Multitasking



Cooperative Multitasking



Cooperative Multitasking



mysyncio

```
@mysyncio.coroutine
def hello():
    for _ in range(3):
        yield
        print("Hello")

@mysyncio.coroutine
def hello_world():
    for _ in range(3):
        print("Introducing")
    yield from hello()
    for _ in range(3):
        print("World")

loop = mysyncio.get_event_loop()
loop.create_task(hello_world())
loop.run_forever()
```

@coroutine

```
def coroutine(func):
    return func
```

yield from == 'blocking'

```
def hello():          # coroutine
    for _ in range(3):
        print("Hello")
        yield

def hello_world():  # coroutine
    for _ in range(3):
        print("Introducing")
    yield from hello()
    for _ in range(3):
        print("World")

hw = hello_world()
while True:
    next(hw)
```

yield from == 'blocking'

```python
def hello():          # coroutine
    for _ in range(3):
        print("Hello")
        yield

def hello_world():  # coroutine
    for _ in range(3):
        print("Introducing")
    yield from hello()
    for _ in range(3):
        print("World")

hw = hello_world()
while True:
    next(hw)
```

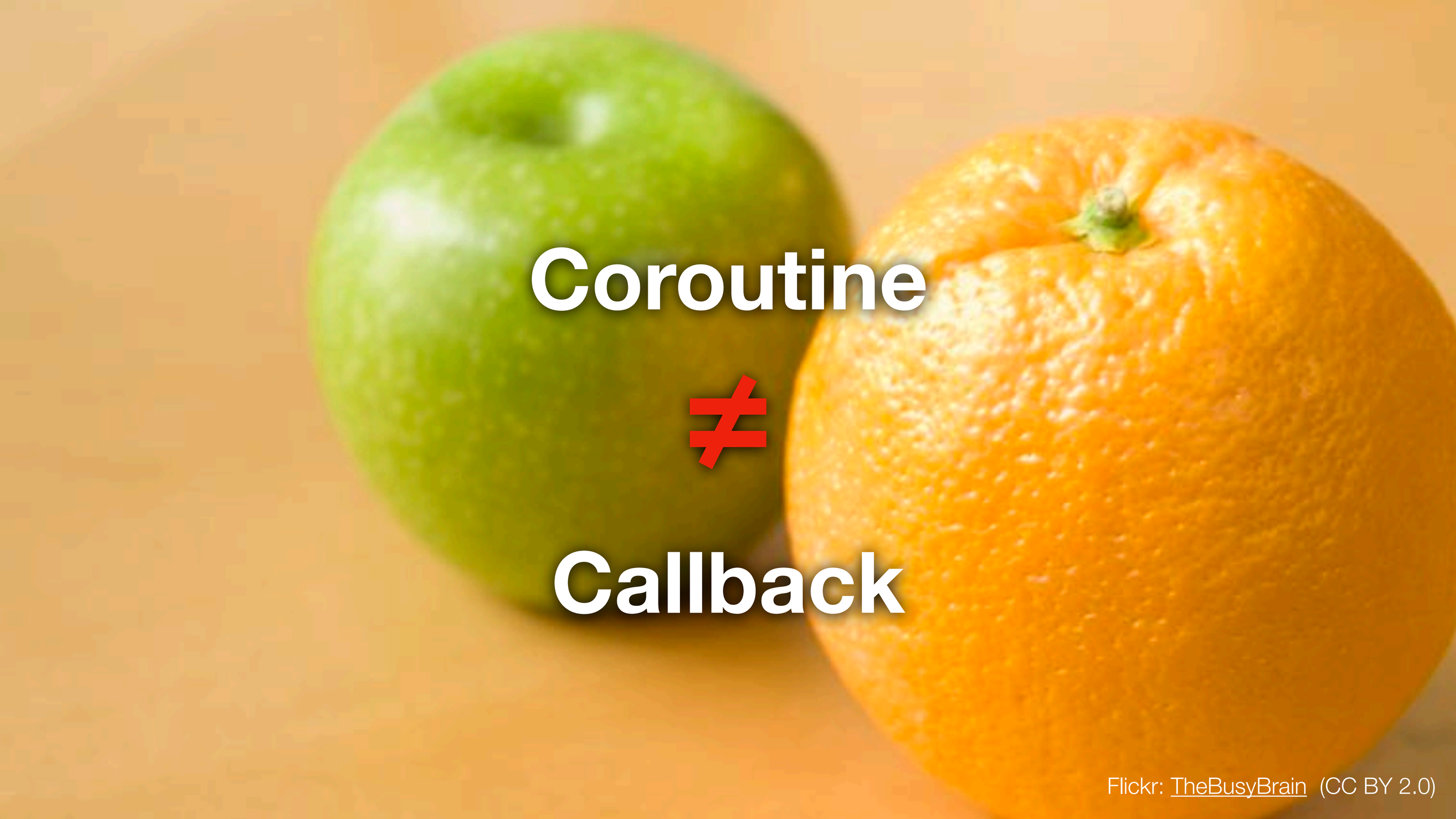
```
Introducing
Introducing
Introducing
Hello
Hello
Hello
World
World
World
Traceback (most recent call last):
  File "demo_generator.py"
    next(hw)
StopIteration
```

mysyncio.sleep

```
@coroutine
def sleep(seconds):
    then = time.time() + seconds
    while time.time() < then:
        yield


@coroutine
def hello_world():
    print("Hello ...")
    yield from sleep(1)
    print("... World")
```

Future

```
class Future:
    def __init__(self, loop):
        self._state = _PENDING
        self._loop = loop

    def done(self):
        return self._state != _PENDING

    def set_result(self, result):
        self.result = result
        self._state = _FINISHED

    def __iter__(self):
        while not self.done():
            yield
        return self.result
```



Coroutine *Inside* Callback

Flickr: Th

Task

```
class Task(Future):
    def __init__(self, coro, *, loop=None):
        super().__init__(loop=loop)
        self._coro = coro
        self._loop.call_soon(self._step)

    def _step(self):
        try:
            result = next(self._coro)
        except StopIteration as exc:
            self.set_result(exc.value)
        else:
            self._loop.call_soon(self._step)
```

Task

```
class Task(Future):
    def __init__(self, coro, *, loop=None):
        super().__init__(loop=loop)
        self._coro = coro
        self._loop.call_soon(self._step)

    def _step(self):
        try:
            result = next(self._coro)
        except StopIteration as exc:
            self.set_result(exc.value)
        else:
            self._loop.call_soon(self._step)
```

Task

```
class Task(Future):
    def __init__(self, coro, *, loop=None):
        super().__init__(loop=loop)
        self._coro = coro
        self._loop.call_soon(self._step)

    def _step(self):
        try:
            result = next(self._coro)
        except StopIteration as exc:
            self.set_result(exc.value)
        else:
            self._loop.call_soon(self._step)
```

Task

```
class Task(Future):
    def __init__(self, coro, *, loop=None):
        super().__init__(loop=loop)
        self._coro = coro
        self._loop.call_soon(self._step)

    def _step(self):
        try:
            result = next(self._coro)
        except StopIteration as exc:
            self.set_result(exc.value)
        else:
            self._loop.call_soon(self._step)
```

Task

```
class Task(Future):
    def __init__(self, coro, *, loop=None):
        super().__init__(loop=loop)
        self._coro = coro
        self._loop.call_soon(self._step)

    def _step(self):
        try:
            result = next(self._coro)
        except StopIteration as exc:
            self.set_result(exc.value)
        else:
            self._loop.call_soon(self._step)
```

Task

```
class Task(Future):
    def __init__(self, coro, *, loop=None):
        super().__init__(loop=loop)
        self._coro = coro
        self._loop.call_soon(self._step)

    def _step(self):
        try:
            result = next(self._coro)
        except StopIteration as exc:
            self.set_result(exc.value)
        else:
            self._loop.call_soon(self._step)
```

Task

```
class Task(Future):
    def __init__(self, coro, *, loop=None):
        super().__init__(loop=loop)
        self._coro = coro
        self._loop.call_soon(self._step)

    def _step(self):
        try:
            result = next(self._coro)
        except StopIteration as exc:
            self.set_result(exc.value)
        else:
            self._loop.call_soon(self._step)
```

Task

```
class Task(Future):
    def __init__(self, coro, *, loop=None):
        super().__init__(loop=loop)
        self._coro = coro
        self._loop.call_soon(self._step)

    def _step(self):
        try:
            result = next(self._coro)
        except StopIteration as exc:
            self.set_result(exc.value)
        else:
            self._loop.call_soon(self._step)
```

Utility Methods

```
class MyEventLoop:
    def create_future(self):
        return Future(self)

    def create_task(self, coro):
        return Task(coro, loop=self)
```

Trying It Out

```
@mysyncio.coroutine
def set_after(delay, value):
    yield from mysyncio.sleep(delay)
    return value

@mysyncio.coroutine
def main():
    task = loop.create_task(set_after(1, '... world'))
    print('hello ...')
    print((yield from task))

loop = mysyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

Trying It Out

```
@mysyncio.coroutine
def set_after(delay, value):
    yield from mysyncio.sleep(delay)
    return value

@mysyncio.coroutine
def main():
    task = loop.create_task(set_after(1, '... world'))
    print('hello ...')
    print((yield from task))

loop = mysyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

Trying It Out

```
@mysyncio.coroutine
def set_after(delay, value):
    yield from mysyncio.sleep(delay)
    return value

@mysyncio.coroutine
def main():
    task = loop.create_task(set_after(1, '... world'))
    print('hello ...')
    print((yield from task))

loop = mysyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

Trying It Out

```
@mysyncio.coroutine
def set_after(delay, value):
    yield from mysyncio.sleep(delay)
    return value

@mysyncio.coroutine
def main():
    task = loop.create_task(set_after(1, '... world'))
    print('hello ...')
    print((yield from task))

loop = mysyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

Trying It Out

```
@mysyncio.coroutine
def set_after(delay, value):
    yield from mysyncio.sleep(delay)
    return value

@mysyncio.coroutine
def main():
    task = loop.create_task(set_after(1, '... world'))
    print('hello ...')
    print((yield from task))

loop = mysyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

Trying It Out

```
@mysyncio.coroutine
def set_after(delay, value):
    yield from mysyncio.sleep(delay)
    return value

@mysyncio.coroutine
def main():
    task = loop.create_task(set_after(1, '... world'))
    print('hello ...')
    print((yield from task))

loop = mysyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

Using Futures

```
@mysyncio.coroutine
def set_after(delay, fut, value):
    yield from mysyncio.sleep(delay)
    fut.set_result(value)

@mysyncio.coroutine
def main():
    fut = loop.create_future()
    task = loop.create_task(set_after(1, fut, '... world'))

    print('hello ...')
    print((yield from fut))

loop = mysyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

Using Futures

```
@mysyncio.coroutine
def set_after(delay, fut, value):
    yield from mysyncio.sleep(delay)
    fut.set_result(value)

@mysyncio.coroutine
def main():
    fut = loop.create_future()
    task = loop.create_task(set_after(1, fut, '... world'))

    print('hello ...')
    print((yield from fut))

loop = mysyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

Using Futures

```
@mysyncio.coroutine
def set_after(delay, fut, value):
    yield from mysyncio.sleep(delay)
    fut.set_result(value)

@mysyncio.coroutine
def main():
    fut = loop.create_future()
    task = loop.create_task(set_after(1, fut, '... world'))

    print('hello ...')
    print((yield from fut))

loop = mysyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

Using Futures

```
@mysyncio.coroutine
def set_after(delay, fut, value):
    yield from mysyncio.sleep(delay)
    fut.set_result(value)

@mysyncio.coroutine
def main():
    fut = loop.create_future()
    task = loop.create_task(set_after(1, fut, '... world'))

    print('hello ...')
    print((yield from fut))

loop = mysyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

Using Futures

```
@mysyncio.coroutine
def set_after(delay, fut, value):
    yield from mysyncio.sleep(delay)
    fut.set_result(value)

@mysyncio.coroutine
def main():
    fut = loop.create_future()
    task = loop.create_task(set_after(1, fut, '... world'))

    print('hello ...')
    print((yield from fut))

loop = mysyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()
```

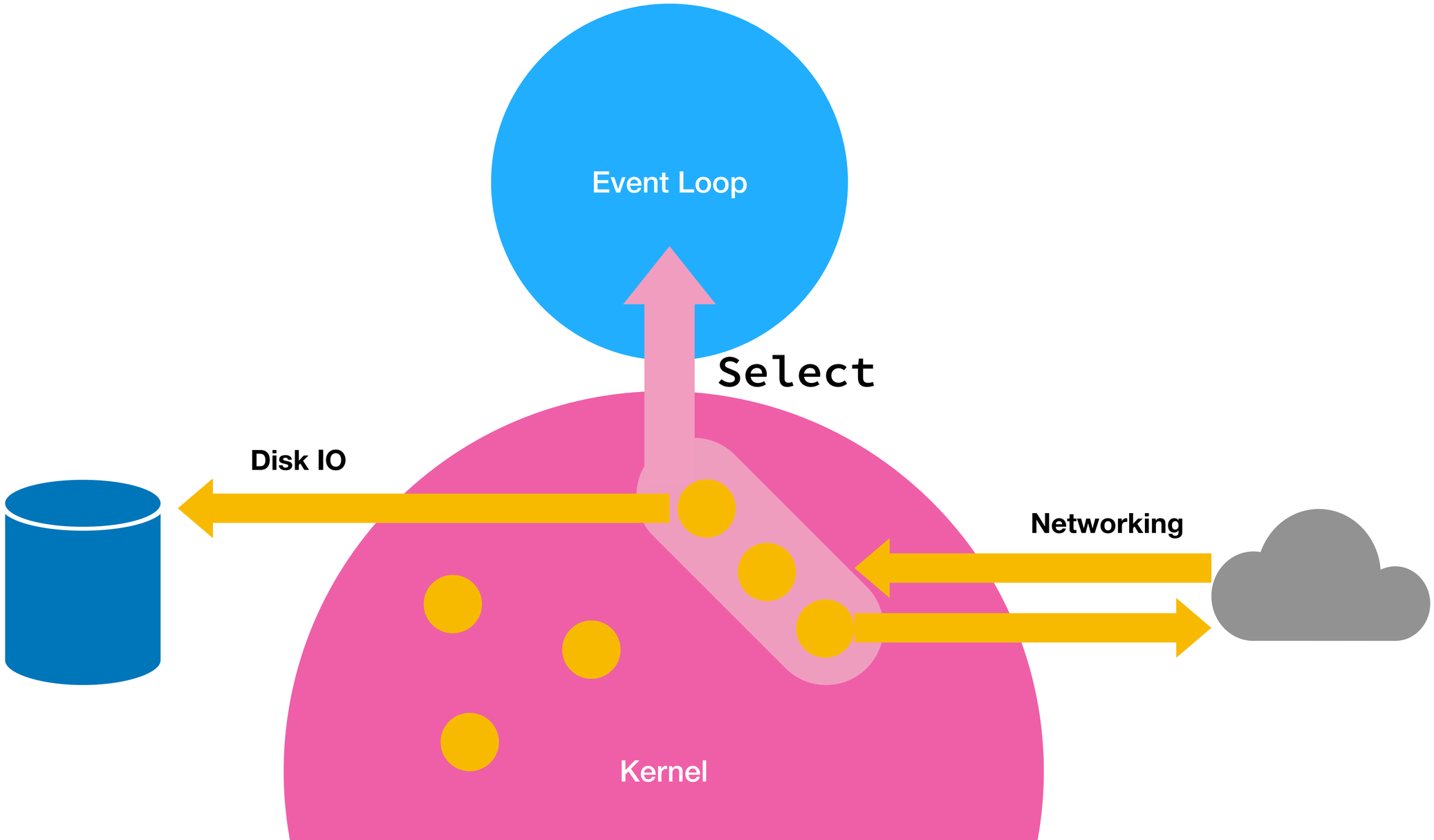
Why?

The IO in AsyncIO

IO in AsyncIO

```
ready_to_read, ready_to_write, _ \
    = select(rlist, wlist, xlist)
```

Select



Selectors

kqueue	OSX/BSD
epoll	Linux
proactor	Windows
devpoll	Solaris

Async IO IO IO

- Register socket & callback with event loop
- Each iteration:
  - **select** is asked which registered sockets are ready.
  - All ready sockets have their associated callbacks called.
  - Callbacks complete associated futures.
  - Coroutines awaiting 'read' return.

There's More

- All the things we skipped
- Coordination methods like **gather**
- Higher-level IO
- Task Scheduling
- Optimizations

Slides & Code

bit.ly/
make-you-an-async

@Judy2k