

A BRIEF INTRODUCTION TO CONCURRENCY AND COROUTINES

Eric Appelt



Thank you to my concurrency mentors:

Sophia and Jacob



Thank you to my concurrency mentors:

Sophia and Jacob

Please report to them any technical inaccuracies or errors in this talk.

GENERAL GOALS

- Concurrency versus parallelism
- Understand iteration protocol, generators, and coroutines
- Use coroutines to add concurrency to a simple example script with asyncio

CODE EXAMPLES FOR THE TALK

- All examples are in github
- <https://github.com/appeltel/PythonConcurrencyTalk2016>
- All live coding starts with modules in the examples
- We'll run in the REPL using "python -i"
- Requires pip install of "requests" and "aiohttp".

THANKSGIVING DINNER

THANKSGIVING DINNER

Roast Turkey
Recipe

THANKSGIVING DINNER

Roast Turkey
Recipe

Mashed
Potatoes
Recipe

THANKSGIVING DINNER

Roast Turkey
Recipe

Mashed
Potatoes
Recipe

Stuffing
Recipe

THANKSGIVING DINNER

Roast Turkey
Recipe

Mashed
Potatoes
Recipe

Stuffing
Recipe

Glazed
Carrot
Recipe

THANKSGIVING DINNER

Roast Turkey
Recipe

Mashed
Potatoes
Recipe

Stuffing
Recipe

Glazed
Carrot
Recipe

Green Bean
Casserole
Recipe

THANKSGIVING DINNER

Roast Turkey
Recipe

Mashed
Potatoes
Recipe

Stuffing
Recipe

Glazed
Carrot
Recipe

Green Bean
Casserole
Recipe

Cranberry
Sauce
Recipe

PARALLELISM VS CONCURRENCY

- Parallelism: Doing (computing) multiple things at the same time
- Concurrency: Dealing with multiple things going on at the same time
- Concurrency may give the illusion of parallelism when quickly switching between tasks.

THE ANIMALS API

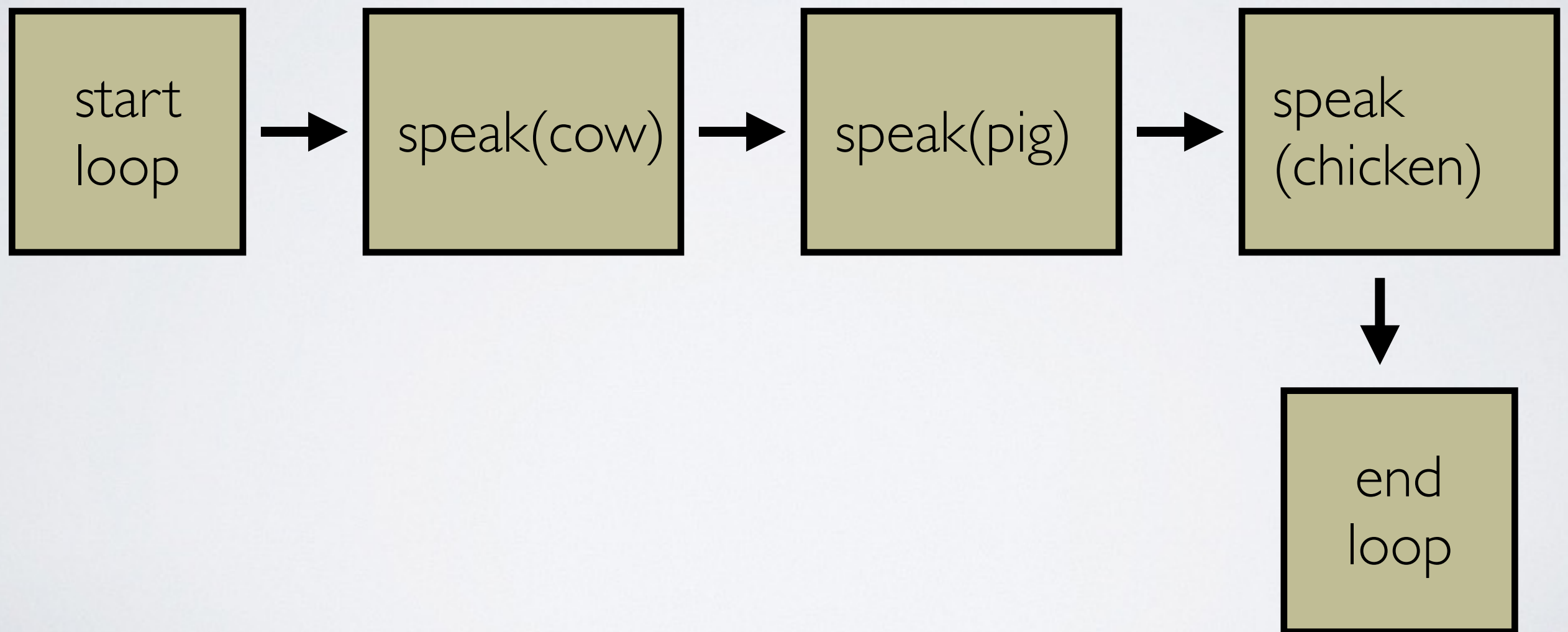


- Enterprise cloud solution for determining what the animals say.
- Compare to on-premises See 'n Say (tm) hardware.
- No capital expenditure requirement, capacity billing.
- Easy to use RESTful API
- Example: <https://www.ericappelt.com/animals/cow>

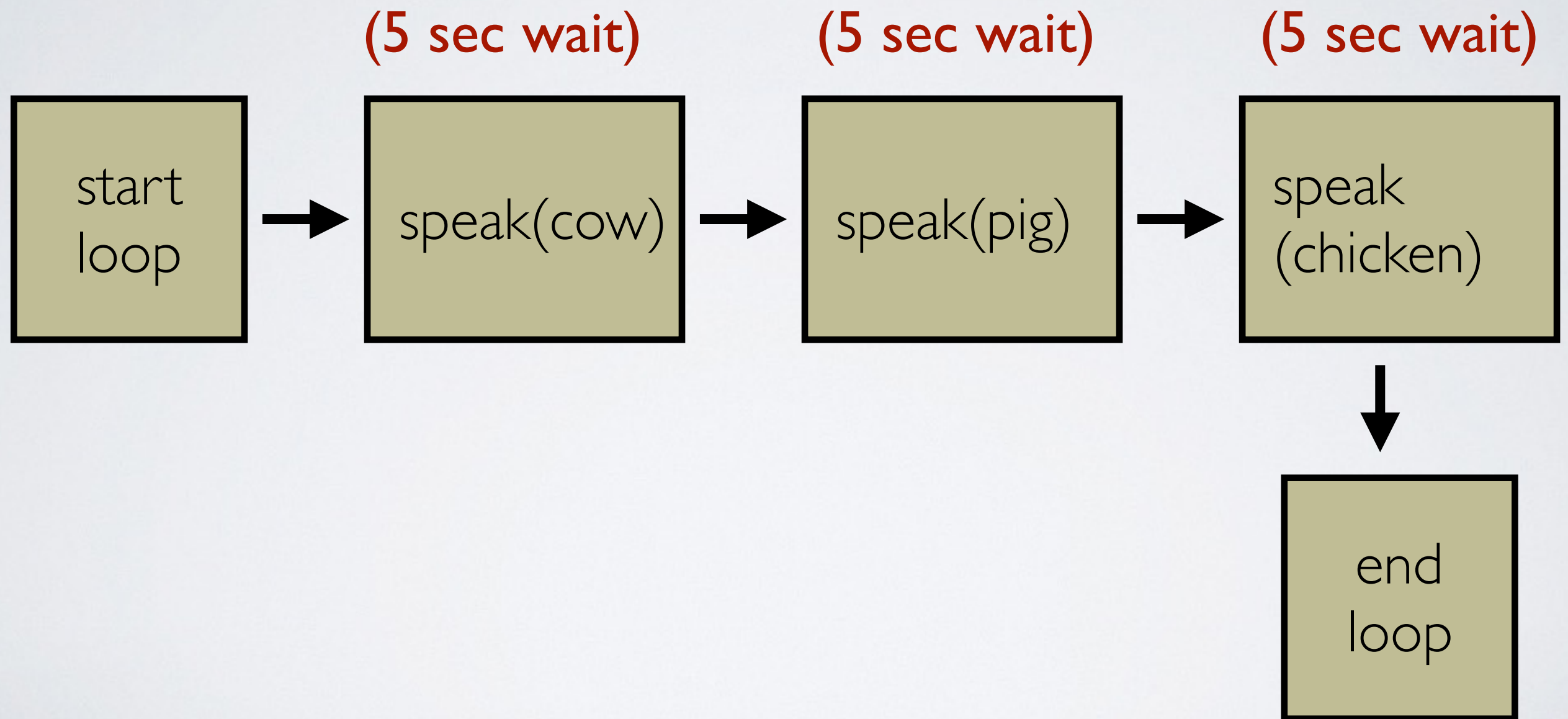
EXAMPLE APPLICATION

- Given a list of animals.
- Connect to animals API to get animal sounds.
- For each animal, animated print 'The X says "Y".'
- Program structured with subroutine to retrieve and print a given animal.
- Loop over animals and call subroutine for each.

SEQUENTIAL EXECUTION



SEQUENTIAL EXECUTION



SUBROUTINE:

- Single entry point.
- Only returns once.

COROUTINE:

- Multiple entry points.
- Can suspend execution, transfer control to caller, be resumed later.

SUBROUTINE:

- Single entry point.
- Only returns once.

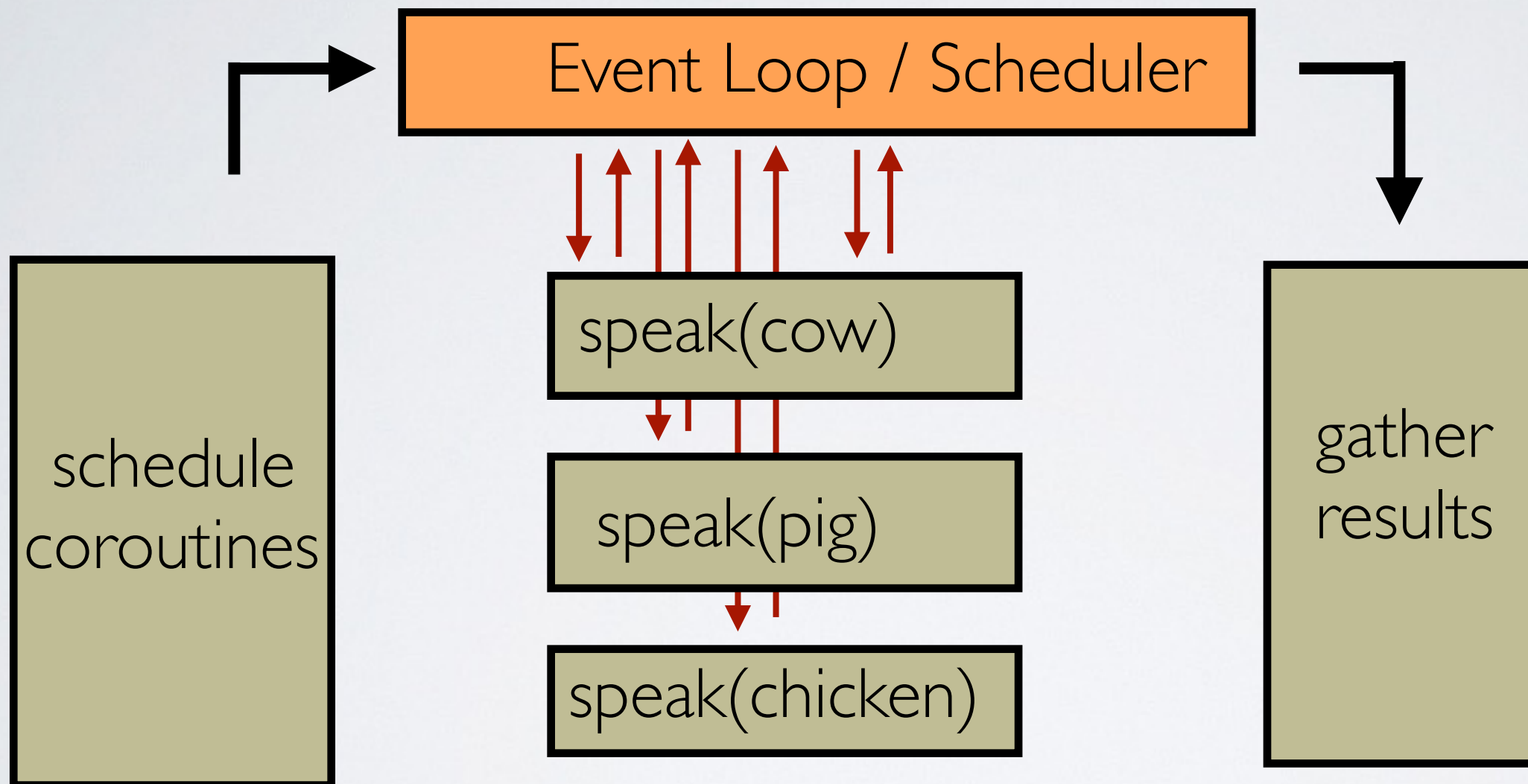
COROUTINE:



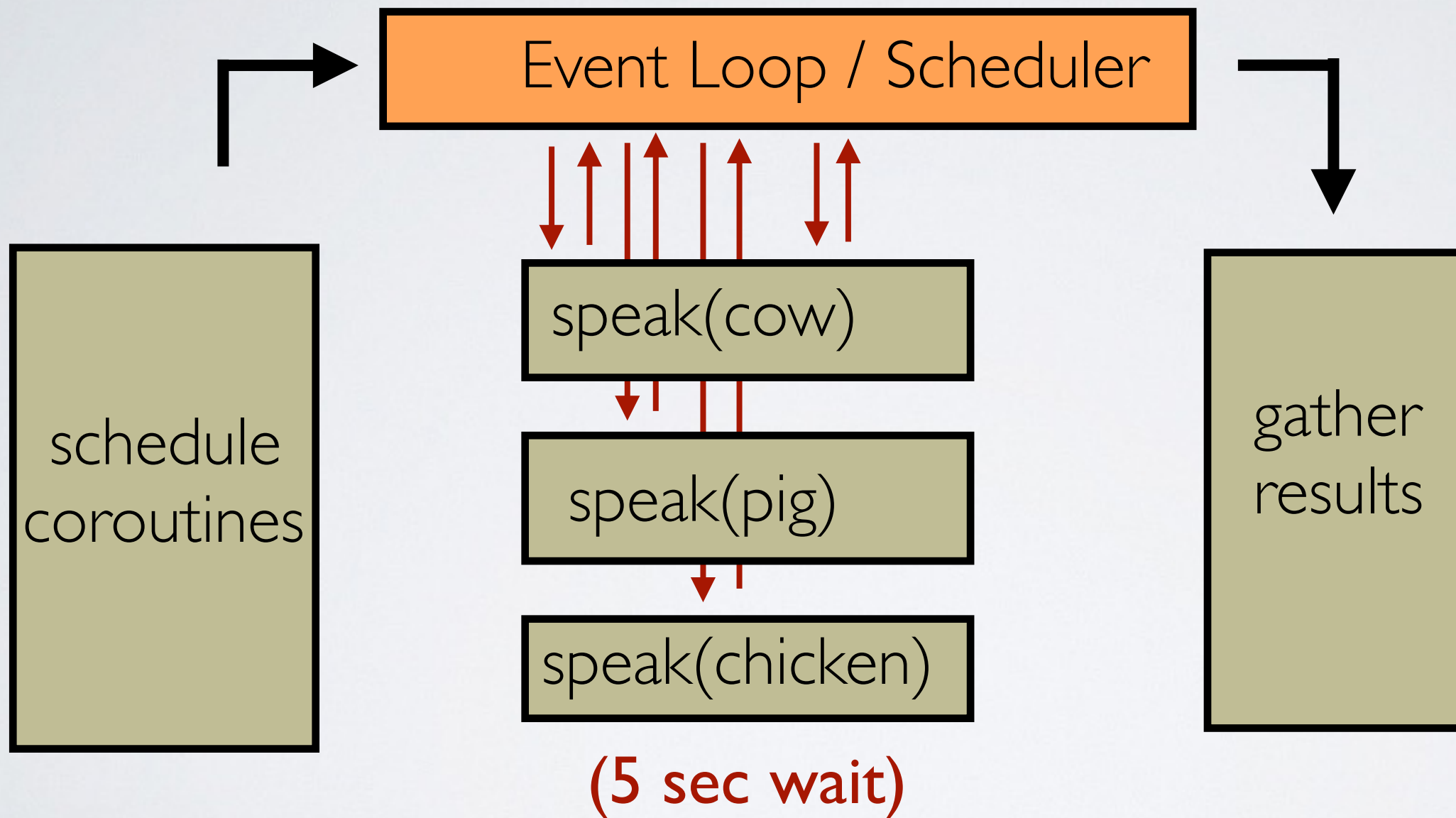
Well actually...

- Multiple entry points.
- Can suspend execution, transfer control to caller, be resumed later.

CONCURRENT EXECUTION



CONCURRENT EXECUTION



PYTHON ON THE ROAD TO COROUTINES...



MILESTONE I: ITERATORS

PEP232 - 2001

- Custom containers (*iterables*) useable in for loops
- An *iterable* defines an `__iter__` method that returns an *iterator*.
- An *iterator* defines an `__iter__` method that returns itself.
- An *iterator* defines a `__next__` method which:
 - Returns the next element from the container, or...
 - Raises a `StopIteration` exception if exhausted.

MILESTONE 2:

PEP255 - 2001

GENERATOR FUNCTIONS

- Look like functions, but include `yield` statements.
- When you call a *generator function*, you get a *generator* back - a type of *iterator*.
- Calling `next()` on a *generator* executes the body of the *generator function* until the next `yield`.
- The *generator* raises `StopIteration` when it completes execution of the *generator function*.

MILESTONE 3:

PEP342 - 2005

IMPROVED GENERATORS

- Want to feed back information when resuming
- Yield becomes an expression (i.e. "x = yield y")
- The result of the expression depends on how the *generator* is resumed after yielding.
- If next() is called, the result is None, or call send(x) and the generator resumes with x as the result
- ...or call throw(Ex) and the generator resumes with the yield statement raising Ex

MILESTONE 4: DELEGATION

PEP380 - 2011

Python 3 only!!!

- Want generators to "delegate" to a sub-generator.
- New "yield from *generator*" expression.
- Yield from will run the specified generator to exhaustion, yielding a value each time it is resumed.
- A *generator function* can now have a return statement.
- Return value becomes the value of StopIteration exception and the value of the yield from expression.

MILESTONE 5: SEPARATION FROM GENERATORS

PEP492 - 2015

Python 3.5+ only!!!

- Want to keep our iterators separate from our coroutines.
- A *coroutine function* is defined with "async def", returns a *coroutine* when called.
- A *coroutine* works like a *generator* but does not implement `__iter__` or `__next__`. To run a *coroutine* one may call `send`.
- A *coroutine function* can delegate with "await" expression, much like "yield from" but can only await from other coroutines or an *awaitable*
- A *coroutine* defined with "async def" cannot itself "yield", but may *await* on special *awaitables* that do.

MILESTONE 5: SEPARATION FROM GENERATORS

PEP492 - 2015

Python 3.5+ only!!!

- Want to keep our iterators separate from our coroutines.
- A *coroutine function* is defined with "async def", returns a *coroutine* when called.
- A *coroutine* works like a *generator* but does not implement `__iter__` or `__next__`. To run a *coroutine* one may call `send`.
- A *coroutine function* can delegate with "await" expressions like "yield from" but can only await from other coroutines.
awaitable



Well actually...

- A *coroutine* defined with "async def" cannot itself "yield", but may *await* on special *awaitables* that do.

WHAT COROUTINES NEED TO BE USEFUL:

- A library or framework providing:
 - An event loop or scheduler to run coroutines concurrently
 - Special awaitables for network I/O that return control to the event loop while waiting.
- Examples:
 - **asyncio**: part of the python standard library
 - **twisted**: mature, established. Can use coroutines with Deferreds.
 - **curio**: natively coroutine-based. New, clean approach.
 - **uvloop**: fast, plug-in compatible with asyncio

ONE-SLIDE INTRO TO ASYNCIO:

- Running a coroutine (or awaitable):

```
loop = asyncio.get_event_loop()  
loop.run_until_complete(coro)
```

- Scheduling multiple coroutines to run concurrently:

```
results = await asyncio.gather(coro1, coro2, ...)
```

- Returning control to the event loop:

```
async def my_coro():  
    ...  
    await asyncio.sleep(0)  
    ...
```

JUST A BIT MORE NEW
SYNTAX...


CONTEXT MANAGERS

manager.__enter__() called
upon entering block



```
with manager as m:  
    ...do stuff...  
    ...do stuff...  
    ...do stuff...
```

manager.__exit__() called
upon exiting block, even if
an exception is raised



ASYNCHRONOUS CONTEXT MANAGERS

manager.__aenter__() returns a
coroutine which is awaited on

```
async with manager as m:  
    ...do stuff...  
    ...do stuff...  
    ...do stuff...
```

manager.__aexit__() called
upon exiting block, returns
a coroutine which is
awaited on

ITERATORS


container.__iter__() called
and returns an iterator

```
for item in container:  
    ...do stuff...  
    ...do stuff...  
    ...do stuff...
```

iterator.__next__() called
each step to fetch each
item, until StopIteration is
raised

ASYNCHRONOUS ITERATORS

container.__aiter__() called
and returns an asynchronous
iterator




```
async for item in container:
```

```
    ...do stuff...
```

```
    ...do stuff...
```

```
    ...do stuff...
```

aiterator.__anext__() called each
step and returns a coroutine,
until StopAsyncIteration is
raised. The coroutine is awaited
on and returns item in each step.



NOW FINISH THE
ANIMALS EXAMPLE
TASK WITH ASYNCIO...

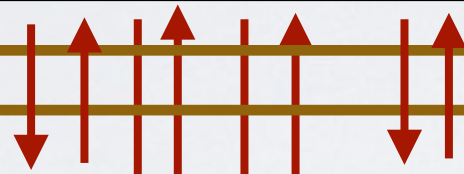
THANK YOU FOR
LISTENING!!!

BACKUP / EXTRAS

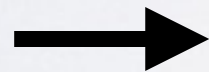
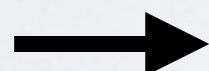
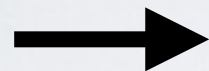
THREADED EXECUTION

OPERATING SYSTEM

Kernel Scheduler



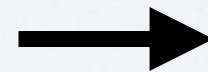
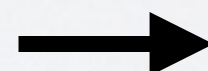
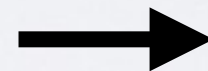
spawn
threads



speak(cow)

speak(pig)

speak(chicken)



join
threads

PYTHON INTERPRETER

WAIT - WHAT'S AN AWAITABLE?

- *A coroutine* is an awaitable.
- An object that implements an `__await__` method that returns an iterator is awaitable.
- *A generator* is not awaitable...
- ...unless decorated by `types.coroutine`.

WHAT DOES GEVENT DO?

- Runs "green threads" or "greenlets".
- Lightweight: Scheduling and task switching handled within the process, kernel sees a single thread.
- Implicit: Standard I/O calls monkey patched to yield to the event loop.
- Cooperative scheduling, but not explicit which calls may yield control.
- Looks much like threading to the programmer, works like coroutines at runtime. No parallel execution.

PYTHON 3.6 GOODIES

- PEP525: Asynchronous Generators
 - Better syntax for making objects for asynchronous iteration.
 - Roughly corresponds to how normal generators nicely implement iteration protocol
- PEP530: Asynchronous Comprehensions
 - `result = [x async for x in async_iterator()]`
 - `result = [await coro(x) async for x in async_iterator()]`
 - `result = {x: await coro(x) async for x in async_iterator()}`
 - etc...

PARALLELISM VS CONCURRENCY

Dear Friend:

*I am having a birthday party.
Please RSVP if you would like
to come and state if you would
like chocolate or vanilla ice
cream.*

(THE INVITATION EXAMPLE)

PARALLELISM VS CONCURRENCY

- Serial Method (Naive):
- Write a single letter.
- Mail it.
- Wait week for return.
- Mark ice cream preference, repeat...
- Very slow - 20 weeks for 20 guests!!!

(THE INVITATION EXAMPLE)

PARALLELISM VS CONCURRENCY

- Parallel Method (Absurd):
- Hire 3 people for 5 weeks.
- Each person mails a letter, waits for return, etc...
- Much faster, 4 weeks for 20 guests.

(THE INVITATION EXAMPLE)

PARALLELISM VS CONCURRENCY

- Concurrent Method (Sensible):
- Write all letters one at a time.
- Mail each letter, one at a time.
- Wait for letters to come in.
- When each letter comes in, mark the preference.
- Takes one week, no need to pay helpers!

(THE INVITATION EXAMPLE)

PARALLELISM VS CONCURRENCY

- Parallelism: Doing (computing) multiple things at the same time
- Concurrency: Dealing with multiple things going on at the same time
- Programming may involve both of these, but concurrency does not always require parallelism.

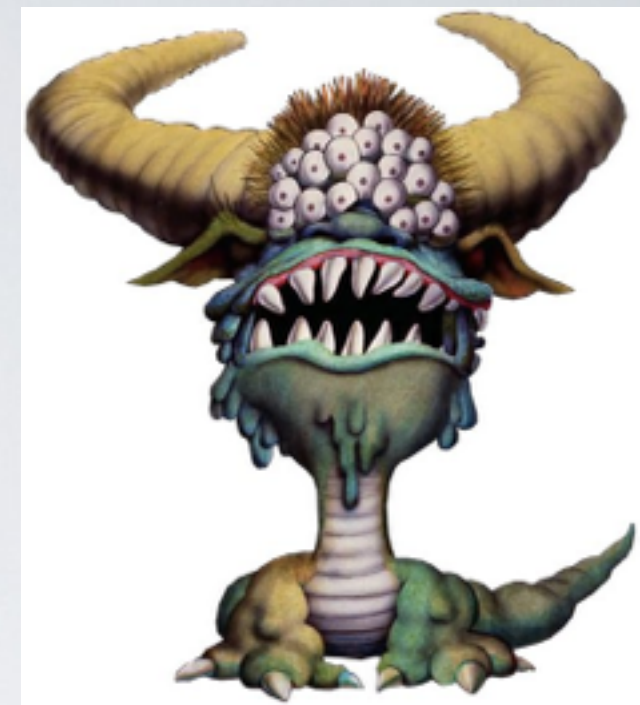
THREADS

- "Lightweight" processes share heap memory and resources.
- Separate execution stacks
- Preemptively scheduled by OS kernel
- Python has built-in high level "threading" model in standard library.
- Can achieve parallelism with multiple CPU cores (*)

THREADING OUR EXAMPLE (IN FOUR ACTS)

- Act 1 (a mess): `animals_thread_wrong.py`
- Act 2 (locks): `animals_thread_correct.py`
- Act 3 (deadlocks): `animals_thread_deadlock.py`
- Act 4 (fixing a deadlock):
`animals_thread_contextmanager.py`

THE GLOBAL INTERPRETER LOCK



- Think of the python interpreter itself as a resource.
- Any thread wanting to interpret a python instruction must acquire a lock.
- Threads waiting on I/O for an instruction may release the lock.
- Only one thread can interpret python bytecode at a time.
- Smart mathematical libraries (i.e. "numpy") will release the lock while running C-extensions.

TOWARDS COROUTINES: AN EXAMPLE RECAP

- We didn't want parallelism anyway.
- We wanted to switch tasks while waiting on the animals API.
- We didn't really want to switch tasks while animated printing.
- The OS kernel preempts our threads in places we can't predict - no control!
- What if we could internally schedule and control when we switch between tasks?