

A Brief Introduction to Concurrency and Coroutines

Eric Appelt
@appeltel



Thank you to Sophia and
Jacob, my concurrency
mentors

SETUP, OUTLINE,
AND WARMUP

Tutorial Setup

- Python 3.5 or later is required
- We will use the requests third-party library
- We will use the aiohttp third-party library
- Tutorial repo: <https://github.com/appeltel/CoroutineTutorial>

Tutorial Outline

- Warmup - the Animals API
- Cooking with Coroutines - a metaphor
- Coroutines by Hand - `async/await`, how does it work?
- Getting to Know Asyncio - your “batteries included” scheduler
- Animals and Aiohttp - putting coroutines to use
- Server Side Animals - a simple web service
- Publish and Subscribe - a streaming web service

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, print 'The X says "Y".'
- Program structured with subroutine to retrieve and print a given animal.
- Loop over animals and call subroutine for each.

AND NOW FOR SOMETHING
COMPLETELY DIFFERENT



COOKING WITH COROUTINES

ORANGE GINGER SALMON



- Preheat oven to 350 degrees F
- Arrange salmon filets on cooking sheet
- Slather each filet with 2 tbsp. of orange-ginger dressing
- Bake salmon in oven for 18 minutes

BOX OF RICE PILAF



- Put 1 3/4 cup of water and 2 tbsp. butter in 2 qt pot
- Bring pot to a boil
- Stir in spice package and rice pilaf, cover, set to low
- Let simmer for 20-25 minutes
- Fluff, let stand for 5 minutes

STEAM-IN-BAG GREEN BEANS



- Poke holes in bag, put on microwave-safe plate
- Microwave for 5 minutes

DINNER IN 35 MINUTES





LET'S AUTOMATE
DINNER!!!

ORANGE GINGER SALMON

```
from time import sleep

from kitchen import (
    oven, baking_sheet,
    salmon, orange_dressing
)

def cook_salmon():
    oven.preheat(350)
    baking_sheet.place(salmon)
    for filet in salmon:
        filet.slather(orange_dressing, tbsp=2)
    oven.insert(baking_sheet)
    sleep(18 * 60)
    return oven.extract_all()
```

RICE PILAF

```
from time import sleep

from kitchen import (
    pot, stovetop, box_rice, water, butter
)

def cook_rice():
    pot.insert(water, cups=1.75)
    pot.insert(butter, tbsp=2)
    stovetop.add(pot)
    stovetop.set_burner(5)
    pot.wait_for_boil()
    pot.insert(box_rice)
    sleep(22*60)
    stovetop.set_burner(0)
    pot.fluff_contents()
    sleep(5*60)
    return pot.extract_all()
```

GREEN BEANS

```
from kitchen import microwave, green_beans

def cook_beans():
    green_beans.poke()
    microwave.insert(green_beans)
    microwave.cook(5*60, power=10)
    return microwave.extract_all()
```

PUTTING IT ALL TOGETHER

```
def make_salmon_dinner():  
    meat = cook_fish()  
    starch = cook_rice()  
    veggie = cook_beans()  
  
    return (meat, starch, veggie)
```

PUTTING IT ALL TOGETHER

```
def make_salmon_dinner():  
    meat = cook_fish()  
    starch = cook_rice()  
    veggie = cook_beans()  
  
    return (meat, starch, veggie)
```

**SOME THOUGHTS ABOUT FUNCTIONS,
ORDER, AND ENCAPSULATION**

(STACK OF NOTECARDS EXAMPLE)

TIME ANALYSIS

```
def make_salmon_dinner():  
    meat = cook_fish()      # 20 minutes  
    starch = cook_rice()    # 30 minutes  
    veggie = cook_beans()   # 6 minutes  
  
    return (meat, starch, veggie)
```


TIME ANALYSIS

```
def make_salmon_dinner():  
    meat = cook_fish()          # 20 minutes  
    starch = cook_rice()        # 30 minutes  
    veggie = cook_beans()       # 6 minutes  
  
    return (meat, starch, veggie)
```

TOO LONG!

WHAT WE WANT: the modularity of functions,
with the ability to concurrently execute.

PARALLELISM VS CONCURRENCY

- Parallelism: DOING multiple things at the same time
- Concurrency: DEALING WITH multiple things going on at the same time

COROUTINES BY HAND

GOALS OF THIS SECTION


- Understand coroutine syntax in python
- Understand how coroutines are run
- Know some major libraries for scheduling coroutines

EXERCISES


- (If you get lost)
- `examples/byhand_functions.py`
- `examples/byhand_coroutines_time_sleep.py`
- `examples/byhand_coroutines_manual_sleep.py`
- `examples/byhand_coroutines_asyncio.py`

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.


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

SCHEDULER LIBRARIES

- **twisted** - well established, progenitor of asynchronous patterns in python, inspiration for other libraries
- **asyncio** - standard library, commitment to support, predates python coroutines, additional abstraction layers for futures and callbacks
- **curio** - newer library, simple design based on just running coroutines
- **trio** - very new, also simple async/await-native design
- others.... (my apologies)

GETTING TO KNOW ASYNCIO

WHAT YOU GOTTA KNOW

- `asyncio.gather(coro, coro, coro, ...)`
- `asyncio.get_event_loop()`
- `loop.run_until_complete(coro)`
- `loop.run_forever()`
- `loop.create_task(coro)`
- `loop.run_in_executor(executor, function, args...)`

HOW TO DO ACTUAL IO

- asyncio provides a streams and transport/protocol api for socket programming
- Not in scope of this tutorial
- Lots of third party add-on apps for popular application protocols


AIO-LIBS PROJECT



aio-lib 

The set of asyncio-based libraries built with high quality

<https://groups.google.com/forum/#!forum/aio-lib>

 **Repositories**

 **People** 17

Pinned repositories

aiohttp

Async http client/server framework (asyncio)

● Python ★ 3.5k 🍴 629

aiopg

aiopg is a library for accessing a PostgreSQL database from the asyncio

● Python ★ 457 🍴 67

aioredis

asyncio (PEP 3156) Redis support

● Python ★ 361 🍴 63

aiomysql

aiomysql is a library for accessing a MySQL database from the asyncio

● Python ★ 328 🍴 49

aiobotocore

asyncio support for boto3 library using aiohttp

● Python ★ 117 🍴 31

yarl

Yet another URL library

● Python ★ 185 🍴 24

AIO-LIBS PROJECT



aio-libs

The set of asyncio-based libraries built with high quality

<https://groups.google.com/forum/#!forum/aio-lib>

 **Repositories**

 **People** 17

Pinned repositories

aiohttp

Async http client/server framework (asyncio)

● Python ★ 3.5k 🍴 629

aiopg

aiopg is a library for accessing a PostgreSQL database from the asyncio

● Python ★ 457 🍴 67

aioredis

asyncio (PEP 3156) Redis support

● Python ★ 361 🍴 63

aiomysql

aiomysql is a library for accessing a MySQL database from the asyncio

● Python ★ 328 🍴 49

aiobotocore

asyncio support for boto3 library using aiohttp

● Python ★ 117 🍴 31

yarl

Yet another URL library

● Python ★ 185 🍴 24

ANIMALS AND AIOHTTP

rewrite the animals client
application to use
asyncio with aiohttp



A FEW WORDS OF
CAUTION!!!

SERVER SIDE
ANIMALS

write your own animals
api server

PUBLISH AND
SUBSCRIBE

QUEUES ARE GREAT

- Mechanism for communicating between coroutines (or threads, processes, etc...)
- One coroutine can put items on a queue, another can take them off in the order they were added
- `asyncio` provides a `Queue` class

ASYNCIO QUEUES

- `q = asyncio.Queue(maxsize=0)` [zero means no max]
- (coroutine) `q.get()` [get item or wait if empty]
- (coroutine) `q.put(item)` [put item or wait if empty]
- `q.get_nowait()` [get item or raise if empty]
- `q.put_nowait(item)` [put item or raise if full]

STREAMING WEBSERVICE EXAMPLE

- <http://localhost:8080/> GET - open a streaming connection, receive messages
- <http://localhost:8080/?msg=<<message>>> POST - send a message to all open streaming connections

BACKUP