Asyncio

A Really Gentle Introduction

About Me

- I've been programming in Python since 1995
- I've worked in financial services since 2003
- I'm currently a developer at Optiver

https://www.optiver.com/

Talk Outline

- Why Asyncio?
- How can 2 things happen at once without threads?
- Getting a Handle on the Future
- A Quick Detour: The Select Module
- Planes, Trains and Automobiles
- Protocols
- Putting it all Together

Why Asyncio?

Why do we need Asyncio?

- I/O is slow…
- Really slow!
- Millions of CPU cycles wasted!
- How can we use this waiting time effectively?

The Reactor Pattern

"The reactor design pattern is an event handling pattern for handling service requests delivered concurrently to a service handler by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to the associated request handlers."

–Wikipedia

Asyncio vs Threads

- Asyncio is (typically) single-threaded
- You don’t have to worry about synchronisation!
- Application controlled context switching
- Asyncio good for I/O bound tasks

Concurrency without Threads?

How Can this Be?

- Back in the days of yore when the world was young…

- Photo attribution: Bill Bertram (from Wikipedia)

- Single core CPU at 7MHz
- Pre-emptive multitasking in 1985!
- You don’t need a multi-core CPU
- Or even a particularly fast one

Live Demo

```
def annie():
    yield "Anything you can do, I can do better.\n" \
          "    I can do anything better than you."
    yield "Yes, I can!"

    yield "Yes, I can!"

    yield "Yes, I can. Yes, I can!"
```

```
def frank():
    yield "No you can't"

    yield "No you can't"
```

```
queue = [annie(), frank()]

while queue:

    singer = queue.pop(0)

    try:

        print(next(singer))

        queue.append(singer)

    except StopIteration:

        pass
```

What have we got?

- Two generators running (sort of) concurrently
- Application defined context switch points
- A loop to dispatch them

Why have we achieved?

- We could have simply printed the lines in order
- Instead we've arranged them logically by singer
  - with application specific sync points

- In asyncio the generators are called coroutines
- The loop to execute them is the event loop

Getting a Handle on the Future

What is a Future?

- “All problems in computer science can be solved by another level of indirection” **- David Wheeler**

- A future is an indirect reference to a forthcoming result.

- You can ask the future to “call back” when ready

Examples of Futures

"The cheque's in the mail!"

"Don't call us, we'll call you!"

Live Demo: Futures

Callbacks

- Callbacks aren’t the nicest way to do things…
- Wouldn’t it be great to write the code inline like before?

Live Demo: Tasks

Tasks

- A task executes a coroutine in an event loop
- At each step the coroutine either
  - awaits (yields) a future
  - awaits (yields) another coroutine
  - returns a result

A Quick Detour

The Select Module

The Select Module

- Select is an OS function to wait for I/O
- It tells you which, if any, I/O channels are ready
- I/O channels can be files, sockets or pipes
- It can wait a fixed length of time or indefinitely

Live Demo: Select

- Asyncio's event loop uses select (or something like it) under the hood

Planes, Trains & Automobiles

Asyncio Transports

- Transports are communication channels
- Responsible for performing I/O & buffering
- There are several types:
  - TCP, UDP, SSL, Pipes

Streaming Transports

- For example: TCP
- The API includes methods such as:
  - close, write, pause/resume reading
- Note: no read method
  - instead you get a call back as we'll see

Other Types of Transports

- Subprocess Pipes
  - The API includes methods such as:
    - get_pid, get_pipe_transport, terminate, kill
- Datagram Transports
  - API includes: sendto, abort

- You don’t create transports directly
- Instead the event loop supplies methods
- For example:
  - create_connection
  - create_server
  - subprocess_exec
- Each takes a protocol factory as its first argument

Protocols

Protocols

- Asyncio protocols process received data
  - and ask the transport to send data
- Again there a several types
  - Streaming, datagram, subprocess
- Your application will create a subclass of one

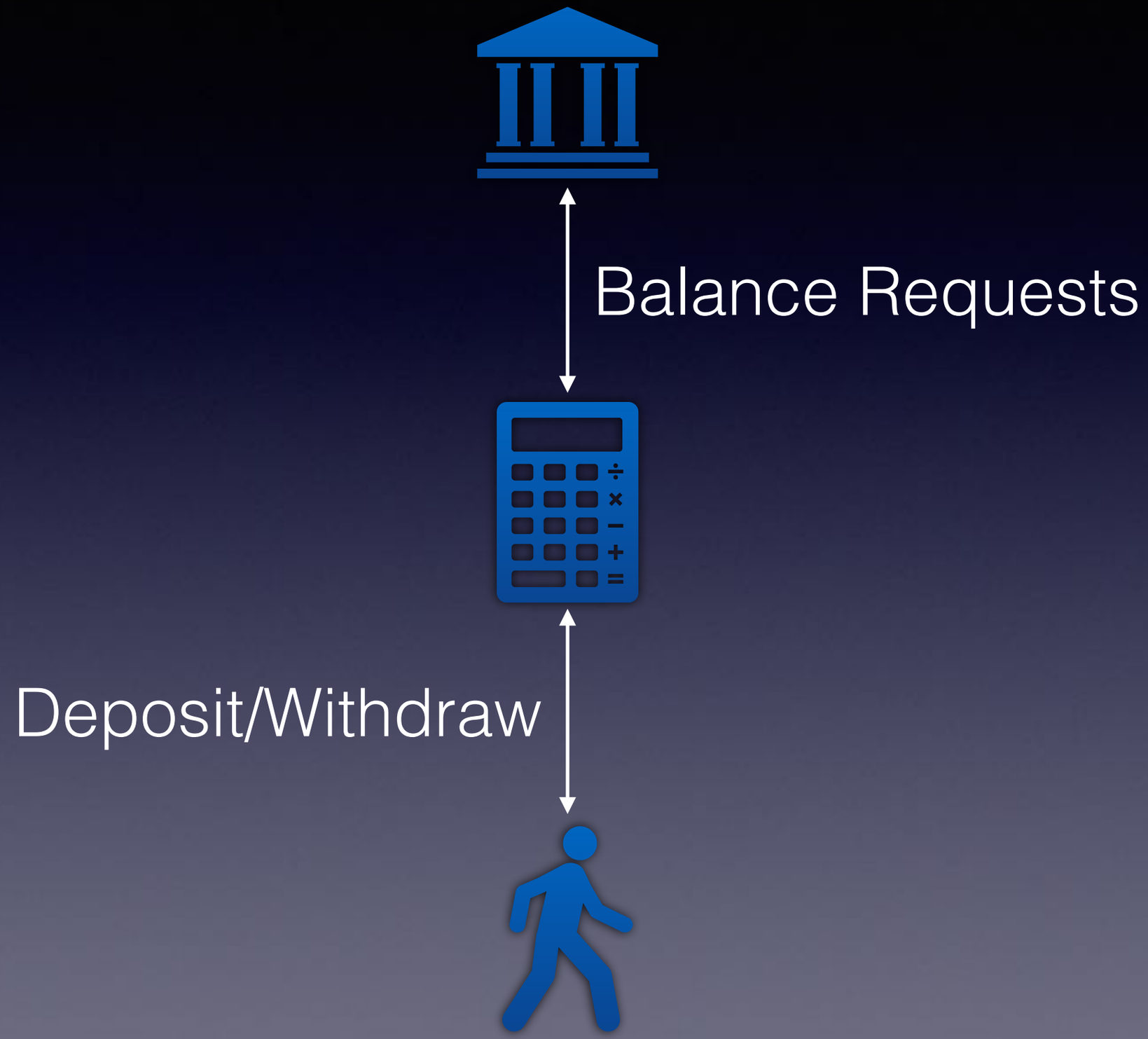
Protocol Demo

- In my demo I'll use Google's Protobuf
- It will serialize/deserialize messages for me
- Raw data may contain multiple messages
- So each message will begin with its length
- Shameless self-promotion!
  - See my Pyrobuf talk from PyCon AU 2016!

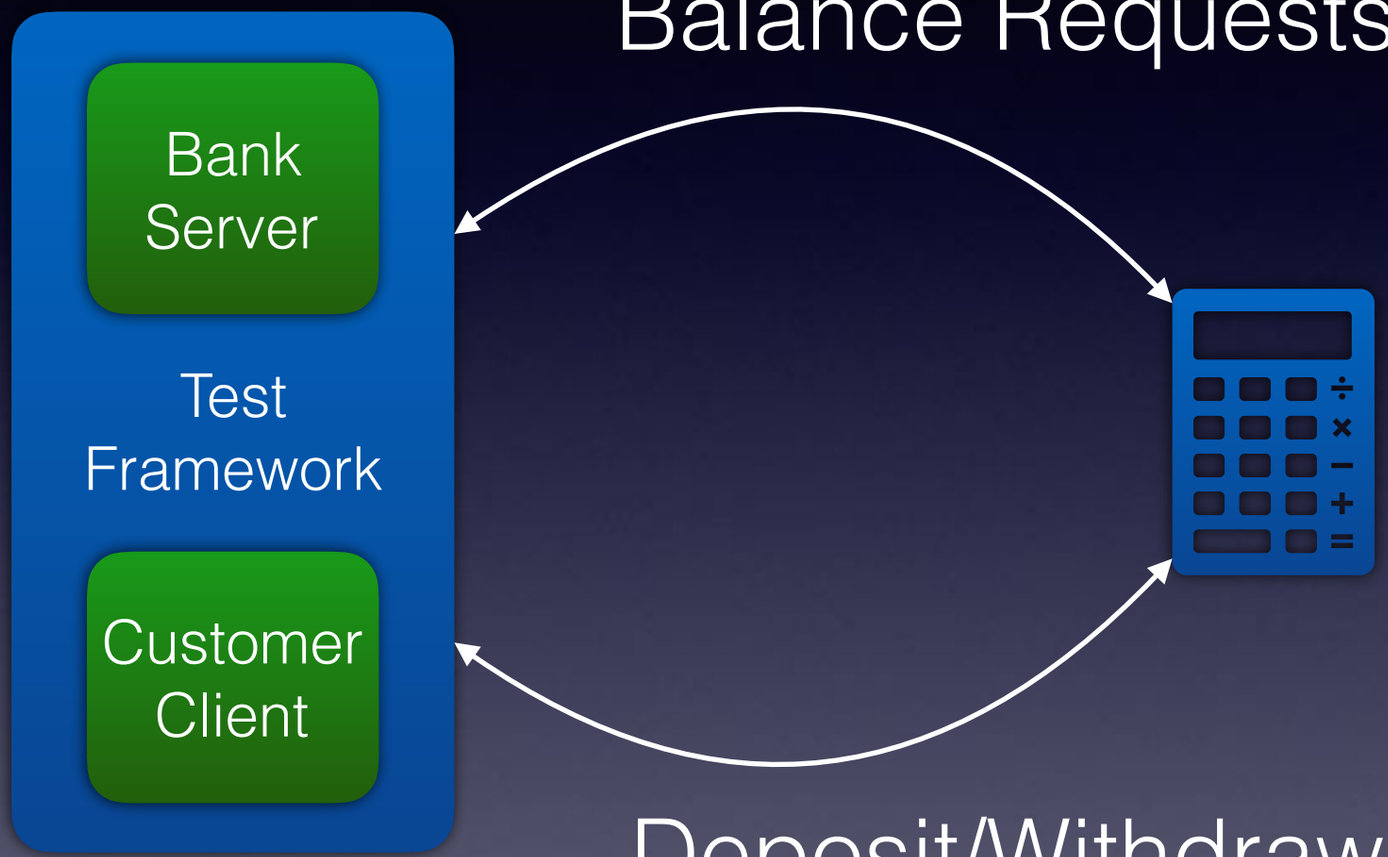
Live Demo

Putting it all Together

- At Optiver we use similar technology for testing
- Imagine a (very) simple ATM app…


Balance Requests
Deposit/Withdraw

Live Demo


Balance Requests
Bank Server
Test Framework
Customer Client
Deposit/Withdraw

Conclusion

- Asyncio is documented in the Python Standard Library
- See especially the “Develop with Asyncio” section

Questions?