Concurrency Part 2: Coroutines & AsyncIO

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 12

Last Class

Concurrency Part 1:

- Different kinds of concurrency:
  - Threads
  - “Tasks”
  - Processes
- Threads are awesome, they speed up code
  - Provide ‘Perceived’ parallelism, in reality it is Sequential
  - Shares address space – has access to common resources
  - Can run multiple threads at the same time
  - ThreadPoolExecutor allows us to run multiple threads at once.
- Race conditions are a thing
  - Access to shared resources need to be ‘Thread Safe’
  - Mutual Exclusion using Locks
- Locks are great but Deadlocks are bad.

Tasks – What are they?

It is possible to run 1 thread and have concurrency.

This avoids the complexity of race conditions and deadlocks.

So, what are tasks?

The technical definition:

A coroutine wrapped in an object.

The more descriptive definition:

A function that does something asynchronously. By Asynchronous, we mean code may not run in order and there may be a gap of time between statements (eg: http request).

Not to be confused by Tasks in other languages and frameworks (foreg: C# .NET Framework)

AsyncIO

AsyncIO is a builtin package that let's us create tasks and coroutines.

This package is made to run asynchronous code.

It's a very scary word amongst Python Developers.

Like so many programming constructs, they are simple to understand but difficult to apply.

No one really teaches this, but we will go through it.

You'll be able to impress other experienced python developers with your knowledge.

Before we get to AsyncIO…

We need to understand one more python construct

Coroutines!

These are the core of how AsyncIO works.

RECAP: Generators

Functions with a yield statement.

The yield statement returns values and “pauses” program execution.

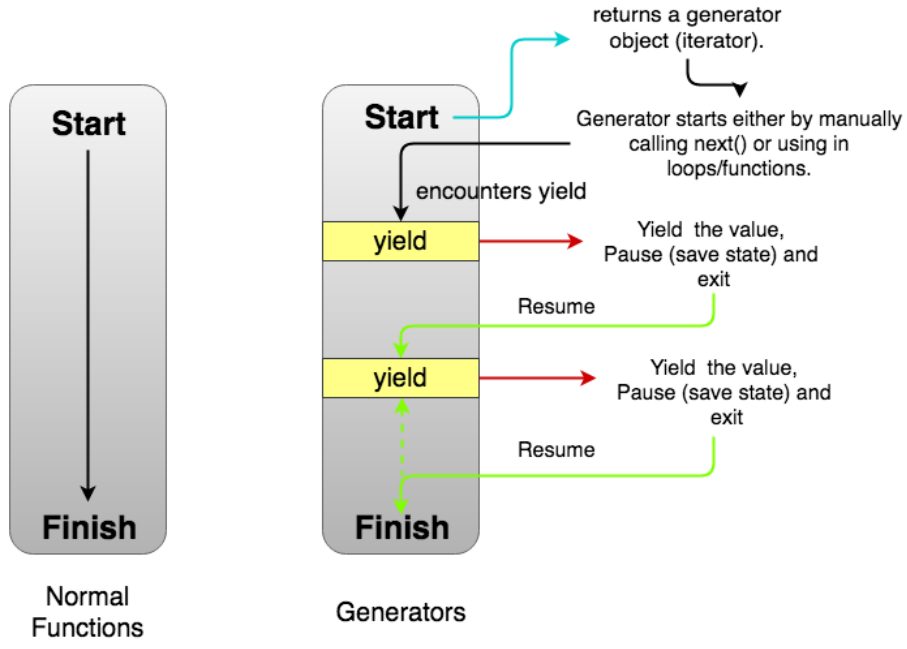
When the python interpreter sees a function with a yield statement it wraps the function in a generator object.

A generator object is an iterable, we can put it in a for loop

We can also write generator expressions using yield from and iterate over it.

Generators allow us to iterate over an iterables without wasting memory

That is, it doesn’t create the whole sequence at the same time in memory



RECAP: Generators

```
def my_generator():
    a = 0
    b = 1
    while True:
        yield a
        a, b = b, a + b

my_gen = my_generator()
for _ in range(5):
    print(next(my_gen))
```

Output:

0

1

1

2

3

Coroutines

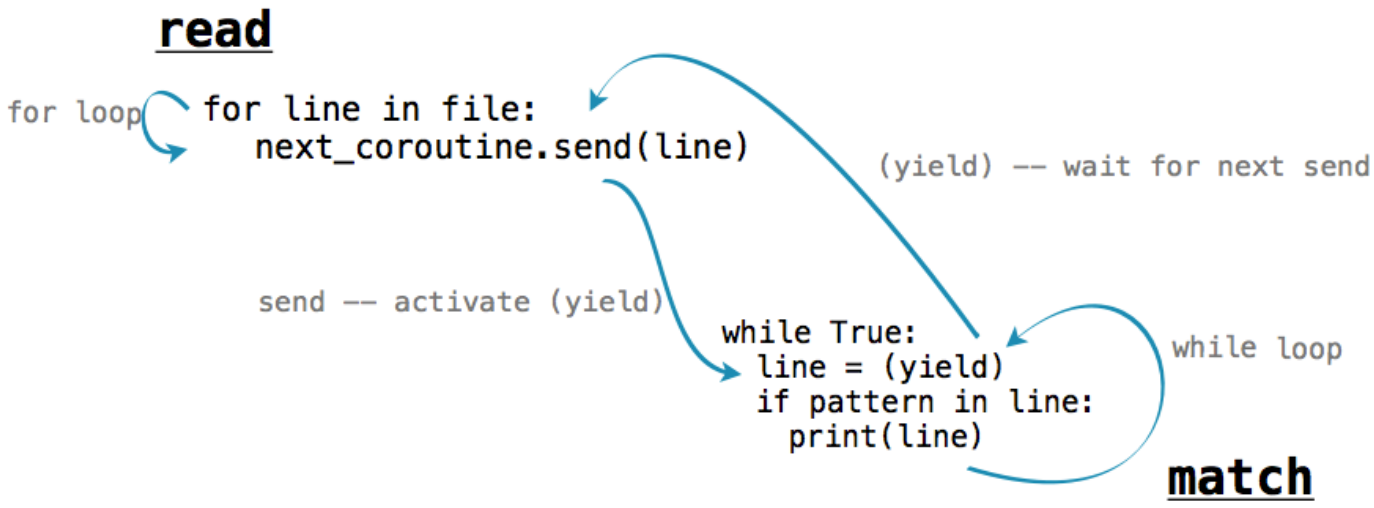
Generators **generate values**.

Coroutines **generate and consume values**.

That is coroutines are functions that use a yield statement, but can also take in new data in the middle of a programs execution.

This is a very powerful construct and the foundation of how AsyncIO works.

Realistically we won't be using these, but its important to understand how they work.



Let's look at an example:

coroutine_sample_code.py

Subroutines and Coroutines

Remember back when you learnt introductory programming?

You learnt that **functions** are also knows as **subroutines**.

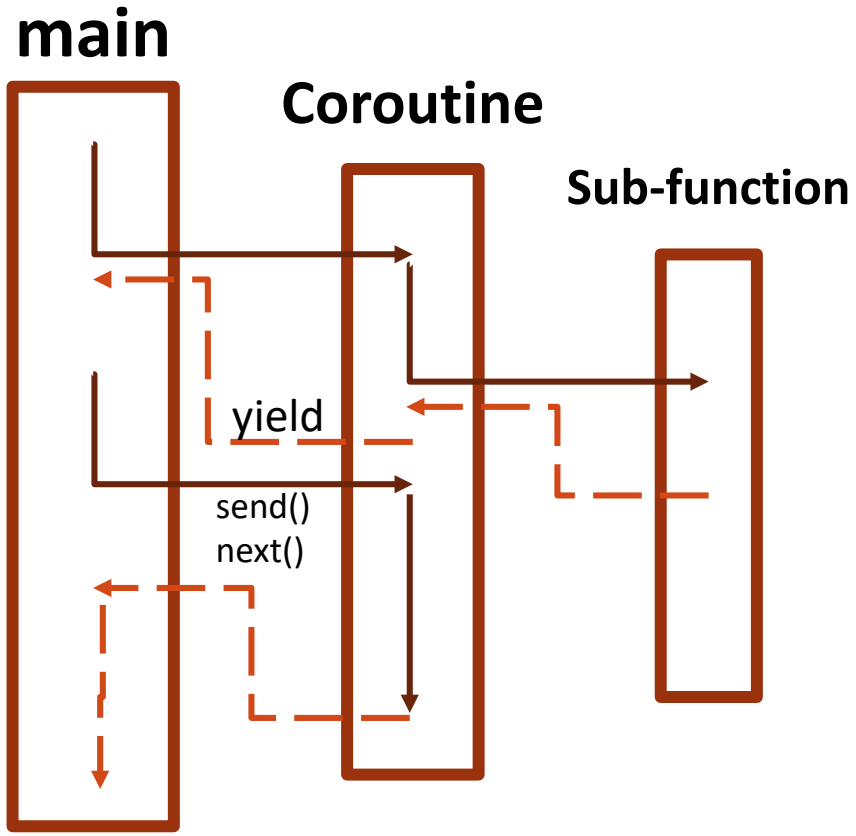
Well, Coroutines are a generalized form of subroutines.

Coroutines can return and accept values anywhere in the function.

Subroutines can only accept values via parameters at the beginning and return data at the end.

Coroutines, due to their flexibility, are great for situations where we need to suspend and resume execution.

- Exceptions
- Event loops
- Multitasking
- Etc.



AsyncIO

As the name implies, a built in package for writing Asynchronous I/O bound code.

These are usually tasks that we need to "**wait on**"

Web request
"**Wait on it**" to return with a http request.

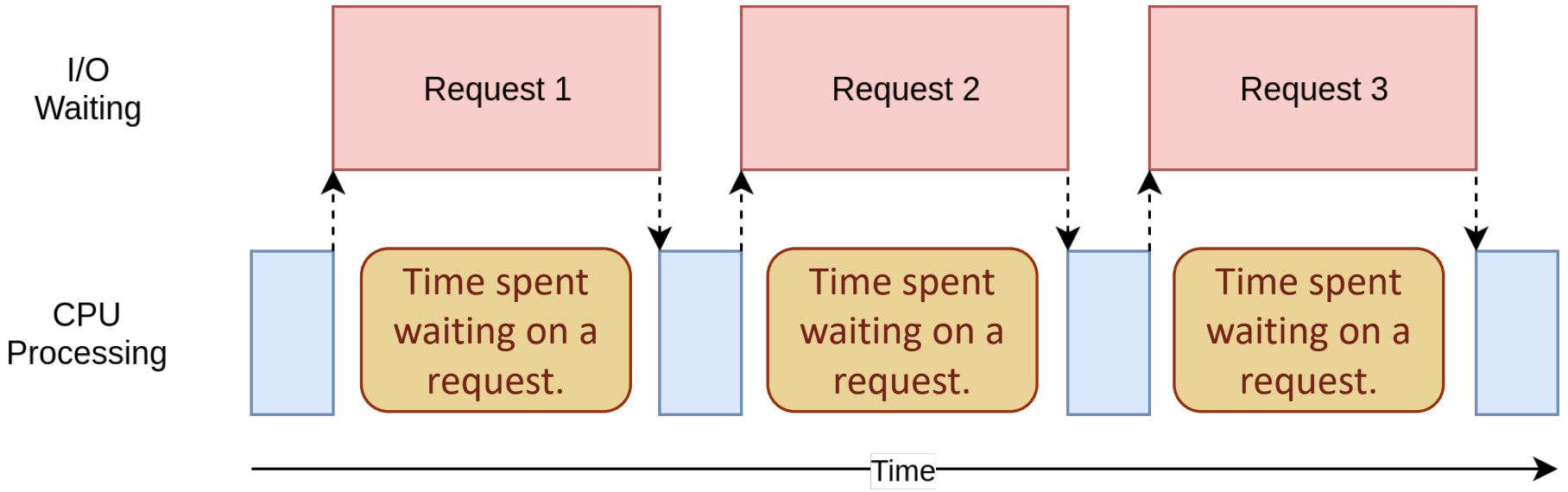
Database query
"**Wait on it**" to return data over the database connection

Filehandling
"**Wait on it**" to open and send/receive data from a file.

Requests

Essentially, these tasks are **Requests**.

Our program requests to perform an I/O task, we wait for it to complete and then we get a Response.



Blocking Calls/Functions

Whenever we write sample code to simulate concurrency, we use time.sleep()

We would never do this in real-world applications.

Waiting on a sleep call is mundane and unnecessary in most cases.

In our code, time.sleep() is used as a stand-in for any **blocking call**. These are instructions that are time-intensive and can stop or slow down our program.

Blocking calls provide an opportunity to switch to another function/task and use the time waiting productively.

AsyncIO – `await`

AsyncIO execute with the help of `await` and `async` keywords.

The `await` keyword is used to wait on blocking functions and calls.

That is, functions that are time-intensive and are usually blocked by a I/O task.
(although this could apply to a CPU bound problem as well)

When awaiting something, our program can switch to other functions and tasks.

```
async def main():
    result = await asyncio.gather(get_data_from_database(),
                                  get_data_from_database())
    print(f"The result: {result}")
```

AsyncIO - async

The keyword `async` is usually put before function calls.

We can also put these before for loops and with blocks.

Essentially the word '`async`' marks a block of code that can run asynchronously.

It's the syntax for telling the python interpreter that this code may wait on a request and it can use this opportunity to switch to another task.

```
async def main():
    result = await asyncio.gather(get_data_from_database(),
                                  get_data_from_database())
    print(f"The result: {result}")
```

AsyncIO

Let’s take a look at an example.

asyncio_sample_code.py

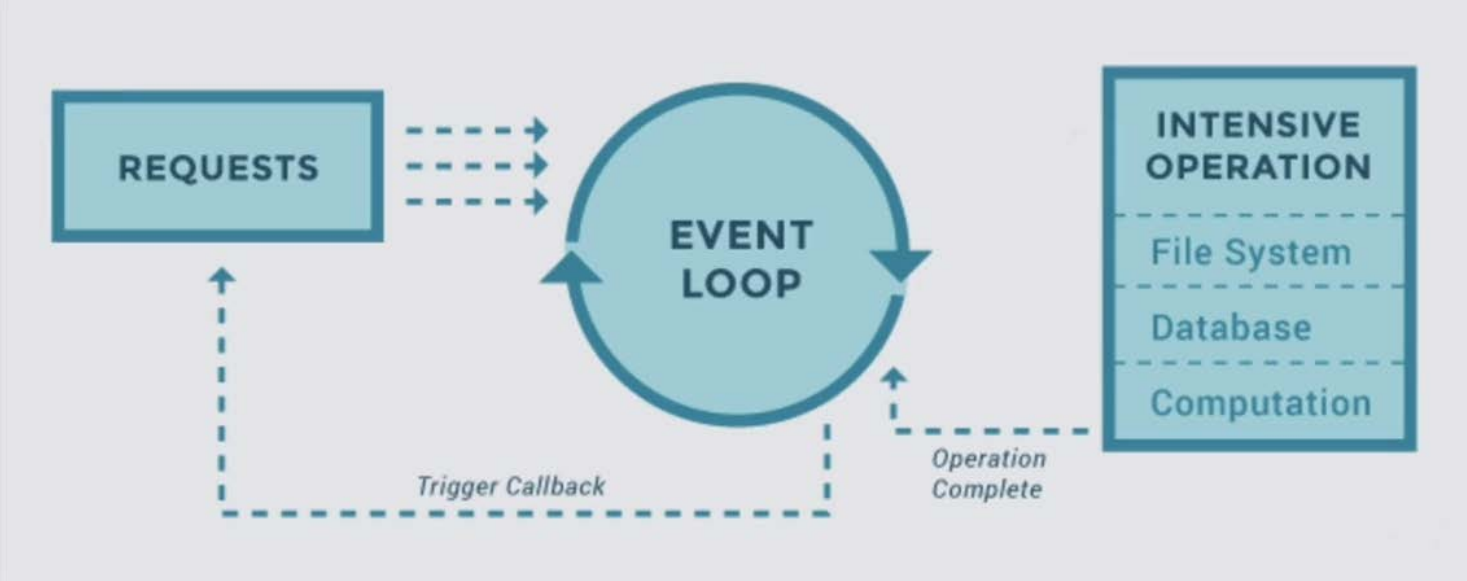
AsyncIO – Event Loop

Who's responsible for managing the task switching?

What voodoo is this that allows concurrency on a single thread?

And does the operating system play a role in this?

Introducing, the **Event Loop**.



AsyncIO – Event Loop

Every time we call `asyncio.run()` we are starting an event loop.

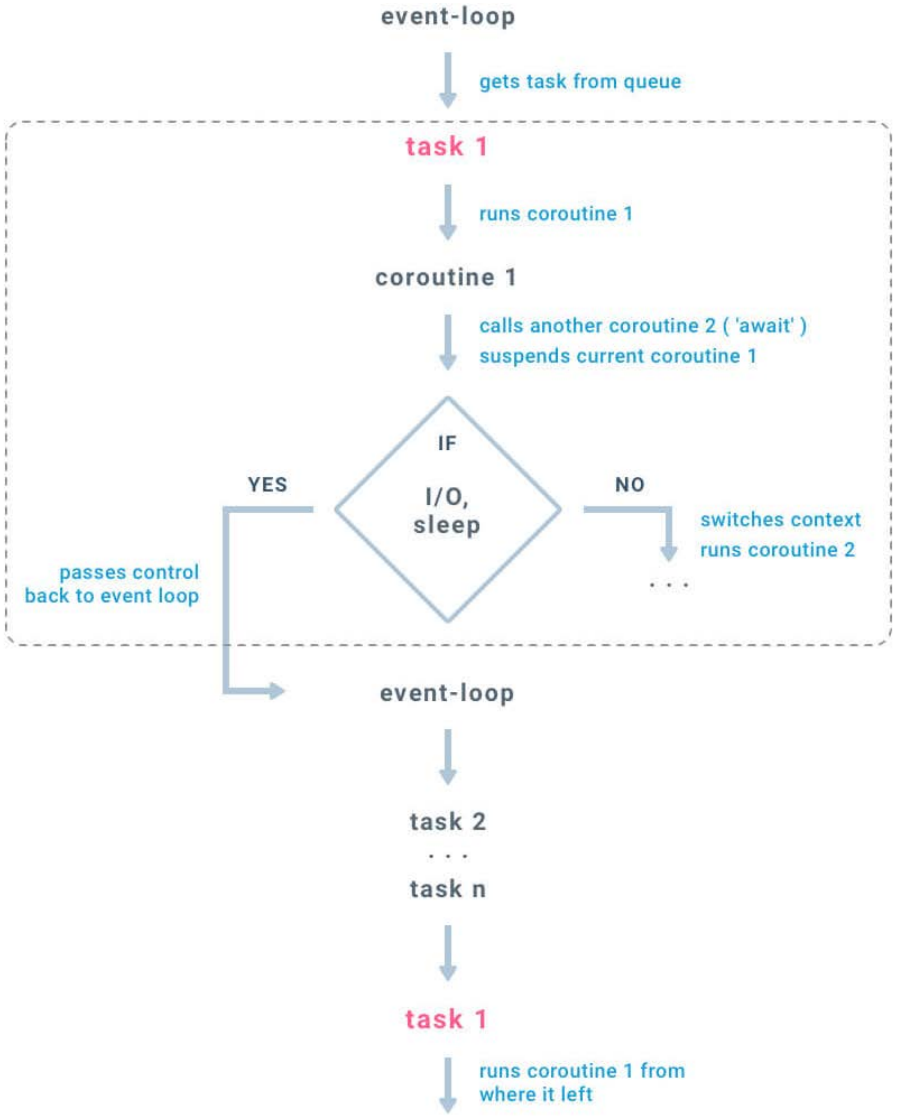
The Event loop is code written and executed in Python 2. It is responsible for keeping track of a number of requests.

Every time a request comes in to the event loop, the event loop will dispatch the request to whatever service it requests. This service is not in the Python environment and is usually an external entity.

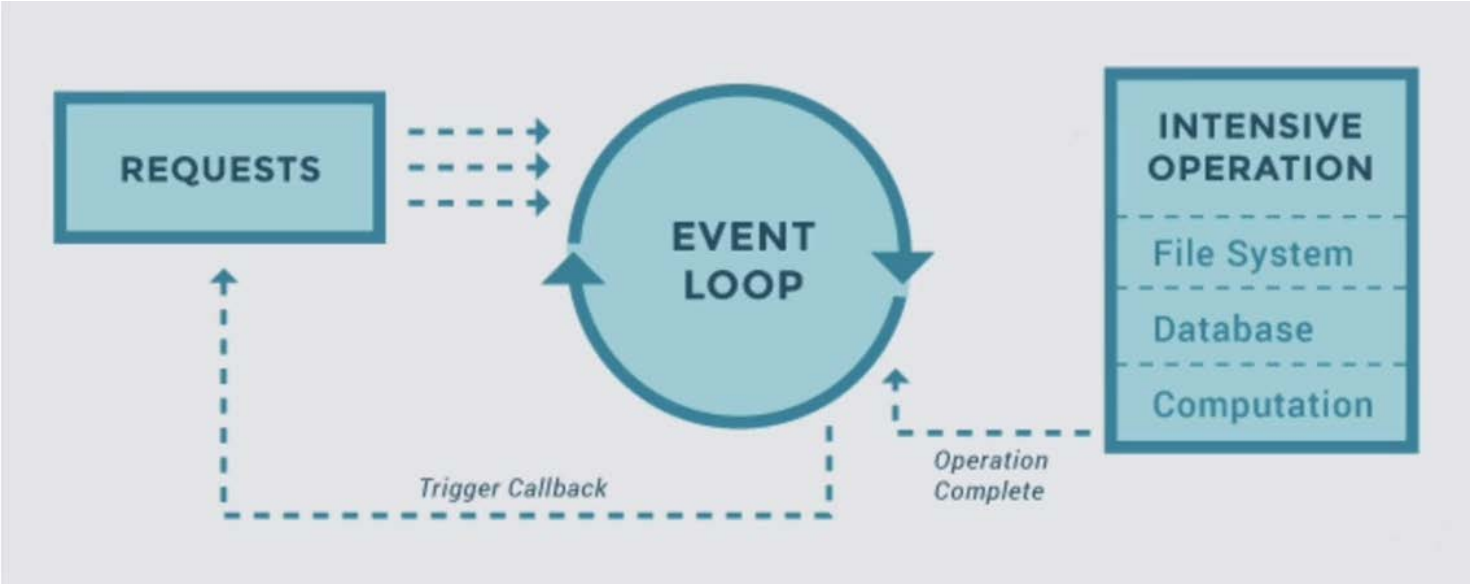
When this happens code is suspended and put on hold.

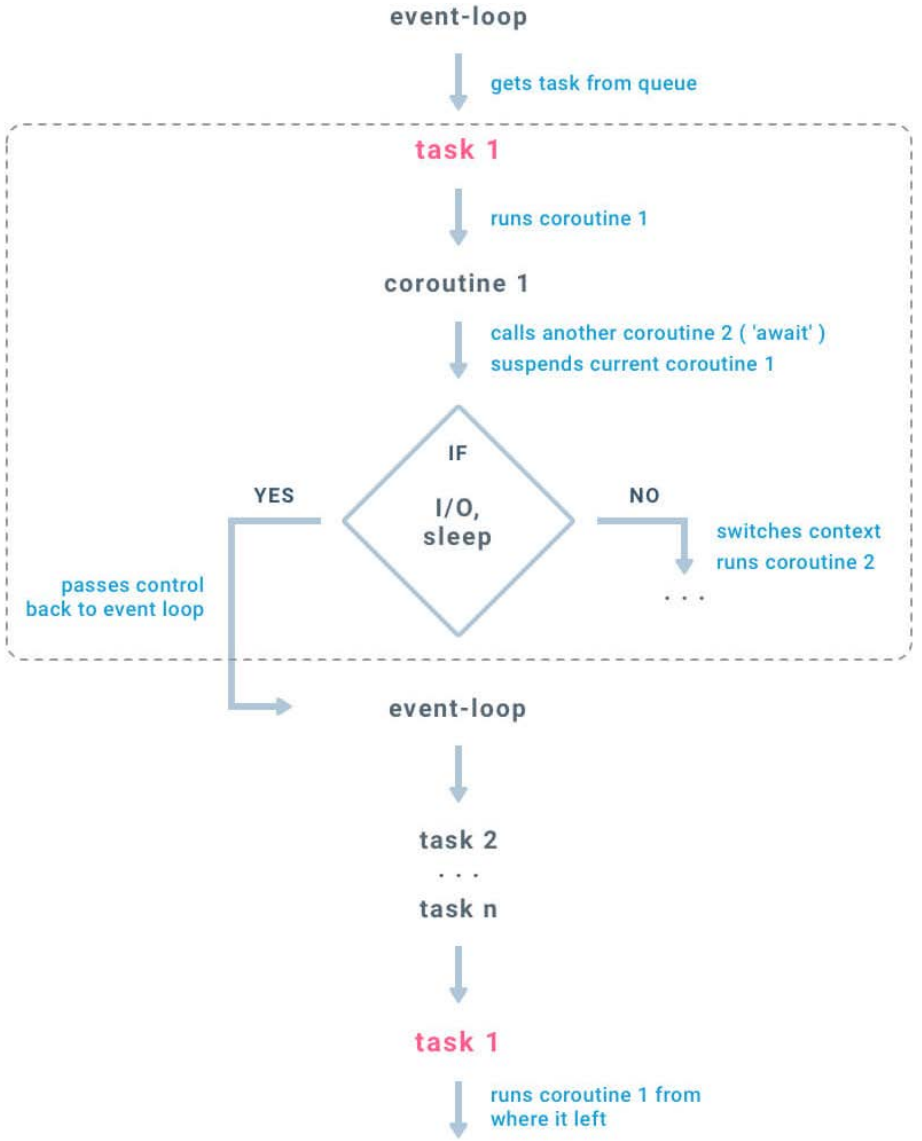
When the intensive operation as outlined by the request is complete, the operating system notifies the event loop that the task is complete.

The event loop then triggers a call back which returns the response and continues the suspended execution.



You really need to see these side by side





AsyncIO.Task

AsyncIO.Taskis an object that wraps around an async coroutine.

It allows us to pass, store and execute async coroutines.

The syntax is:

```
task = asyncio.create_task(async_coroutine())
```

We can then execute the task when the time is appropriate

```
# single task
asyncio.run(task)

# multiple tasks
asyncio.gather(*tasks)
asyncio.gather(task1, task2,…, task n)
```

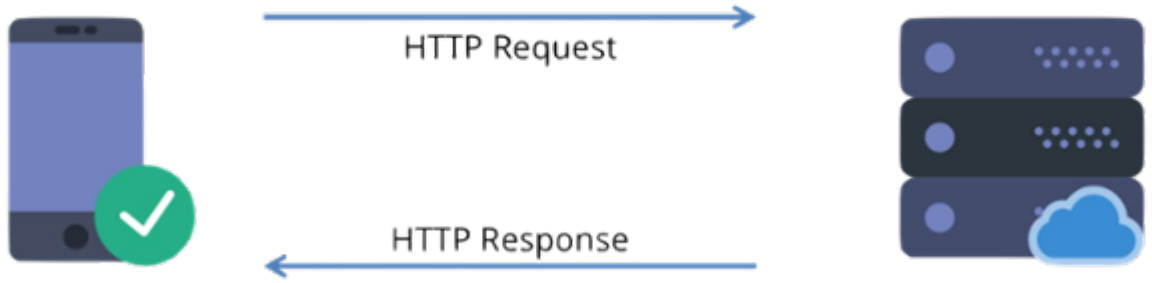
HTTP Requests - aiohttp

HTTP – Hyper Text Transfer Protocol.

HTTP is a protocol, that is a set of rules to communicate requests and responses over the internet.

There are 2 kinds of HTTP Requests:

- GET
- POST



aiohttp is a package that let's you write code for **asynchronous** clients and servers.

Note the word asynchronous.

This package has been built to be used asynchronously

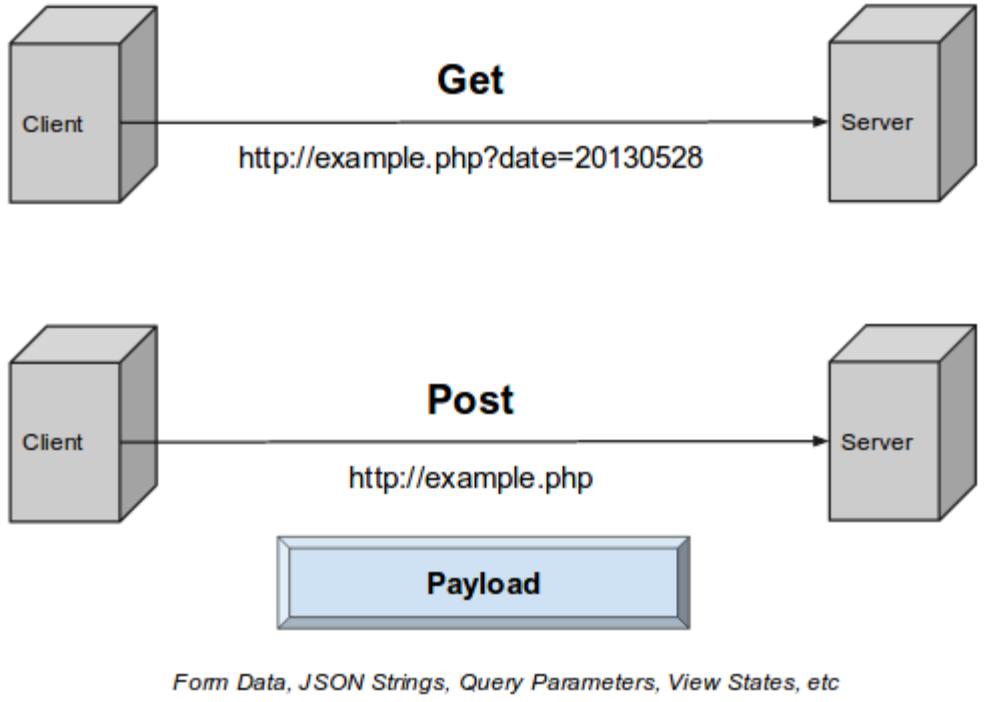
GET and POST Requests

GET requests are requests sent to a server to retrieve the state of a resource. In simple terms, it is a request to get data, usually in JSON format.

- May require parameters. These are visible in the URL
- Browser can cache results
- Easy to test and implement

POST requests are for writing data or submitting data to be processed. For example, logging in to a website with a username and password.

- Can send larger amounts of data in a more secure manner.
- Parameters are not visible in the URL.
- Browser will not cache results.



aiohttp and asyncio

aiohttp can allow us to send GET and POST requests using **`asyncio`** coroutines.

Let's see how we can do this.

asyncio_request_samplecode.py

Recap

And that's AsyncIO. This is the method of choice to implement concurrency in our code

- Python provides support for concurrency via threads, tasks and processes.
- Threads and tasks provide the illusion of concurrency.
- You can execute tasks with the help of Async IO
- Async IO is built on coroutines. Coroutines can yield and accept values anywhere in the function body.
- You await a blocking call. The Event Loop handles the flow of control for executing requests, suspending and waking up async coroutines
- You can store async coroutine instances as Tasks.

That’s it for today!

Lab 10 due Sunday March 29th