FINAL EXAM INFO

Final info

- Wednesday April 15th 11:30am – April 16th 8:00am
- All online exam. Closed book, no peer communication
- Honor system of invigilation
  - Work on your own
  - No discussing of how to solve problems
  - I will be looking very closely at code for similarities

- Covers everything from week 6 up to week 13
- If it’s in the slides and links in slides, it’s possibly on the exam

Final format – 2 parts

- Part 1: Online D2L, similar to the quizzes we've done
    - True/False
    - Multiple choice
    - Multiple options
    - Read code, choose the output
    - Read scenario, choose the correct code
- Strictly timed on D2L, currently ~30mins

Final format

Part 2: Practical coding

- Zip file containing multiple python file
- Each python file is one practical question
- Coding or draw UML diagram
- Coding is guided, part of code written for you

This part you have until April 16th 8:00am to submit to a final exam folder on D2L

STUDY STRATEGY

Strategy

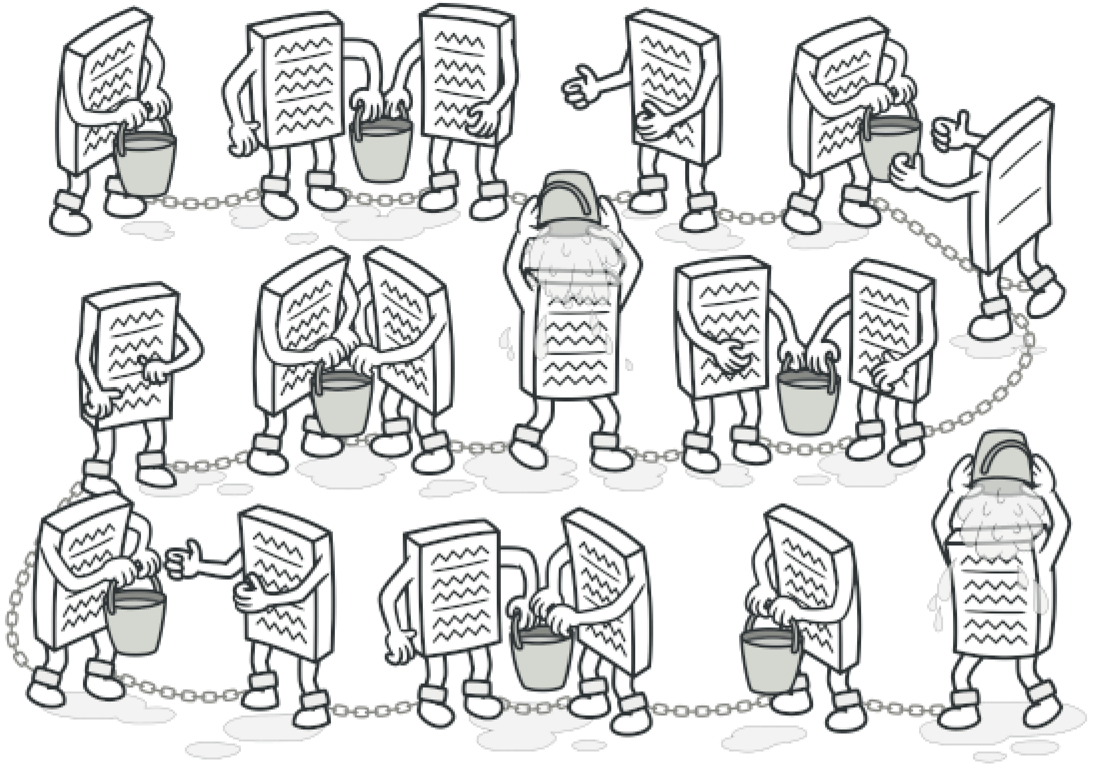
- Go over slides and links in slides
- Go over code samples I reference in slides sample.py
- Recall what we did in labs
- Understand all design patterns
  - What problems each is suitable for
  - How they look like in code/UML
  - When to use one over another one

Strategy

- Something unclear in slides?
  - Search geeksforgeeks, tutorialspoint, programiz + topic
  - Ask me

- Think about topics we spent a lot of time on in lecture
  - If I went over something in depth, chances are I’m going to cover it

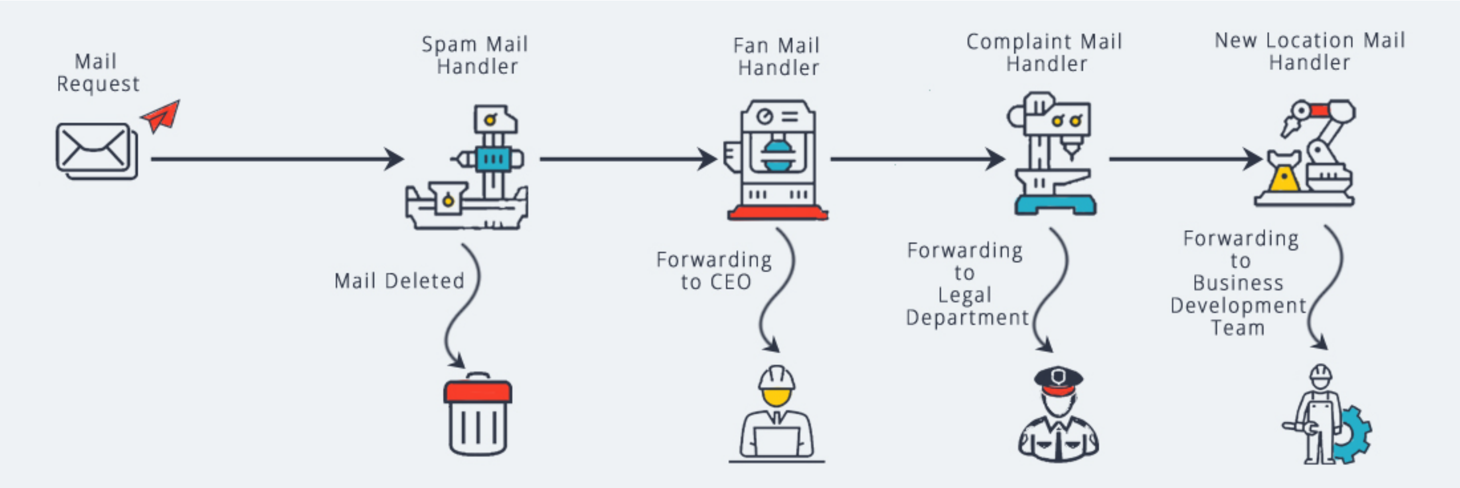
Chain of Responsibility



WHEN YOU WANT TO HAVE A LOT OF RESPONSIBILITIES BUT STILL BE FLEXIBLE

Chain of Responsibility

- When an object or a set of objects needs to undergo different “steps” of processing.
- These could be checks, validation, formatting, security, setup, etc.
- Set up a series of **Handlers**. Each handler is unique but **implements the same interface** and does something to a **Request**.
- Each Handler **has a reference to another handler** and it may, depending on it’s code, pass on the request to another handler.



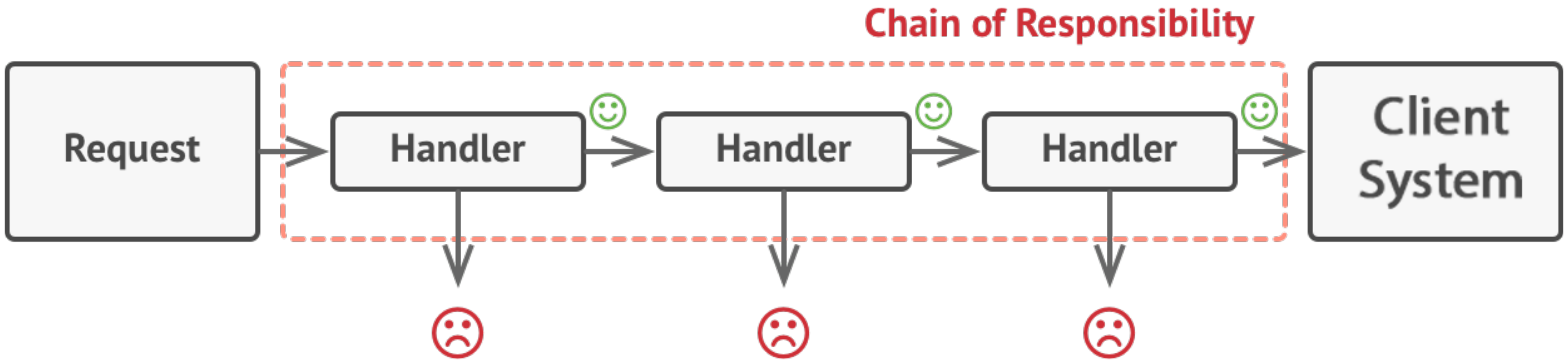
Chain of Responsibility

The Chain of Responsibility pattern separates these different processing steps into different classes. Each class is a **Handler**

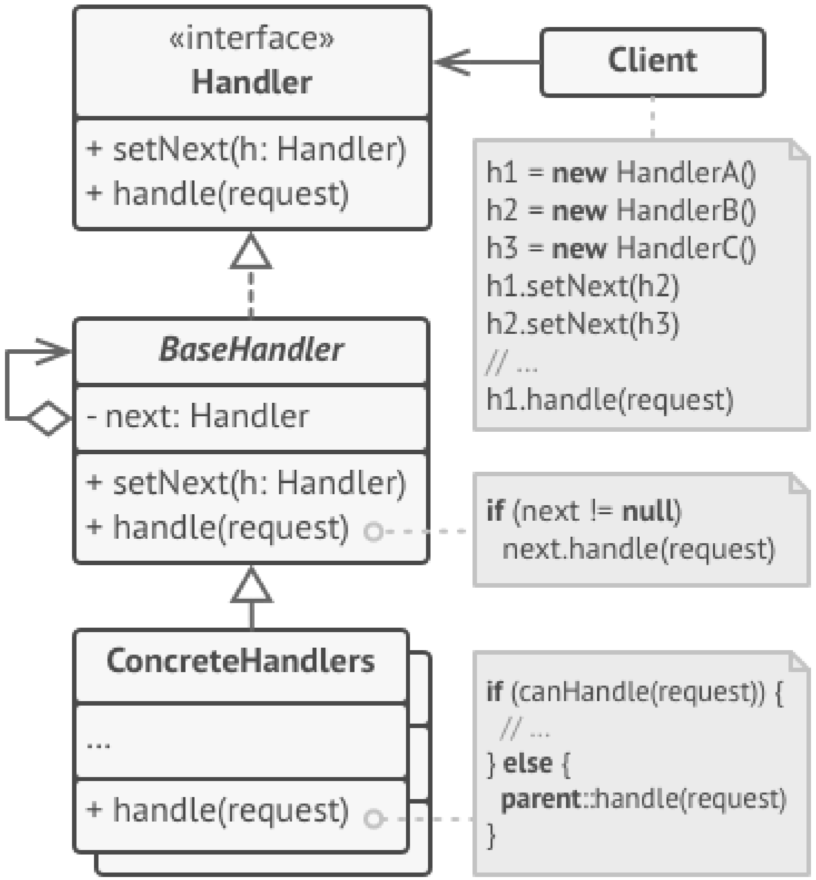
All these classes share the same interface, that is how one handler can pass on the request to another handler.

It kind of is like a linked list of responsibilities where each handler forms a node. We can arrange a different list for different scenarios.

The execution may stop midway and exit the chain if the Handler deems it necessary



Chain of Responsibility



Requirements:

- Each Handler implements the **same interface.**
- Base Handler is an **Optional** parent class that can hold some duplicate code (such as **setNext(h: Handler)**)
- Each handler implements a handle(request) method which is where they carry out their specific code.

Scenario

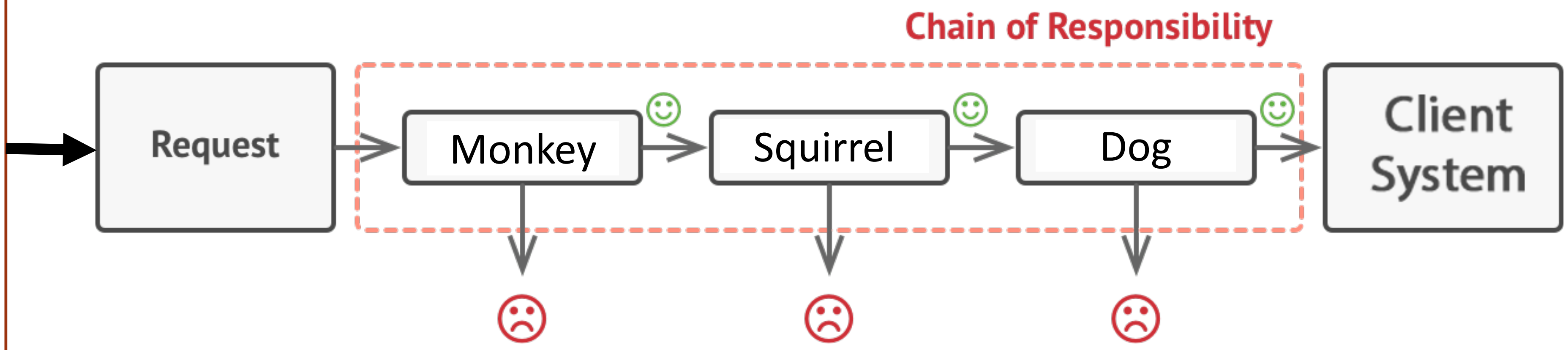
A monkey, squirrel and dog are lined up

A zookeeper passes some food to an animal

If the animal wants to eat it, they will eat and zookeeper gives the next food

If the animal doesn't want to eat, they will pass to the next animal

Client System
1. Sends a food item as a request
2. Animals determine if they want to eat it



chain_of_responsibility.py

Chain of Responsibility: Why and When do we use it

- If your program is expected to process different kinds of requests in various ways.
- When you need to do something in a particular order.
- If the sequence and ordering of request-processing is not known before hand and needs to be determined at run-time. We can control the order of request handling.
- Single Responsibility Principle. Each handler does one thing. We have decoupled **classes that invoke operations** (E.g. EnrolmentSystem) from **classes that perform operations** (the handlers).'
- Open/Closed Principle. We can introduce new handlers without modifying existing handlers or client code.

Chain of Responsibility – Disadvantages

- More classes to maintain.
- Some requests may end up unhandled. This may happen if we don't set up the ordering of handlers properly.

THREADS

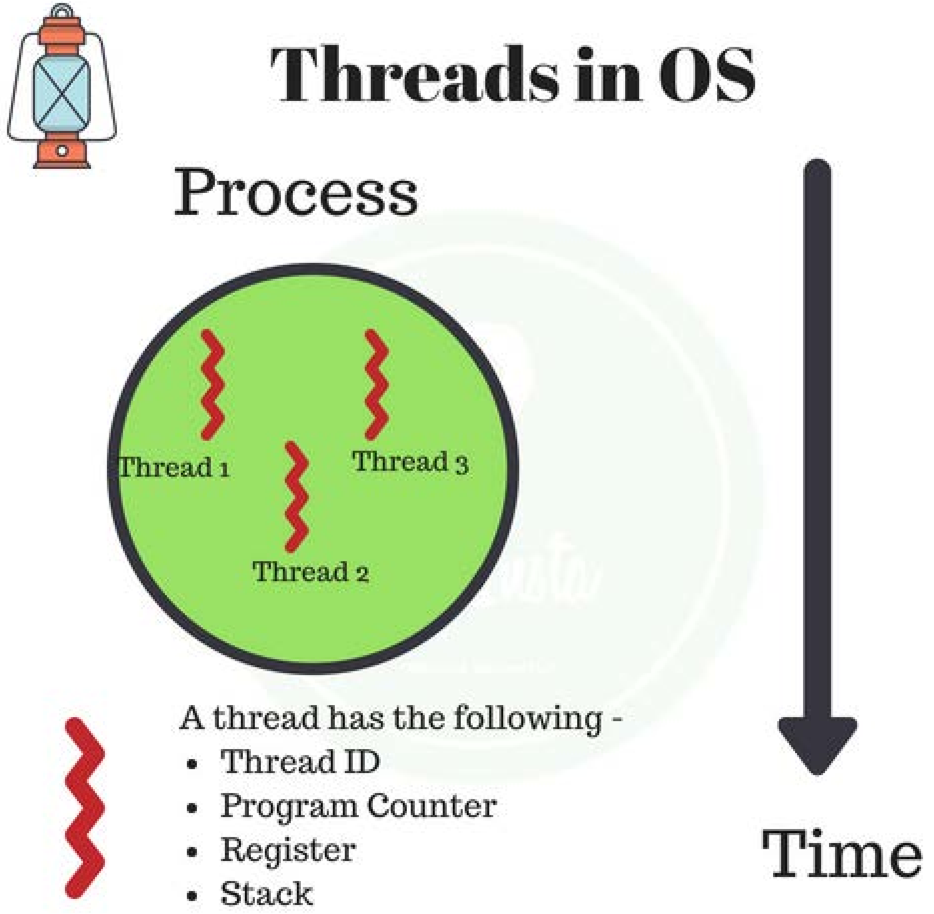
What is a Thread?

A thread is a **sequence of instructions** within a process that can be **executed independently**.

A process is a program that has been loaded into memory along with all the resources it needs to operate

A **process can run multiple threads**, each of which executes a stream of instructions concurrently and independently.

Threads operate within the **same address space** and as such, share the same resources and data.



Pre-emptive Multitasking

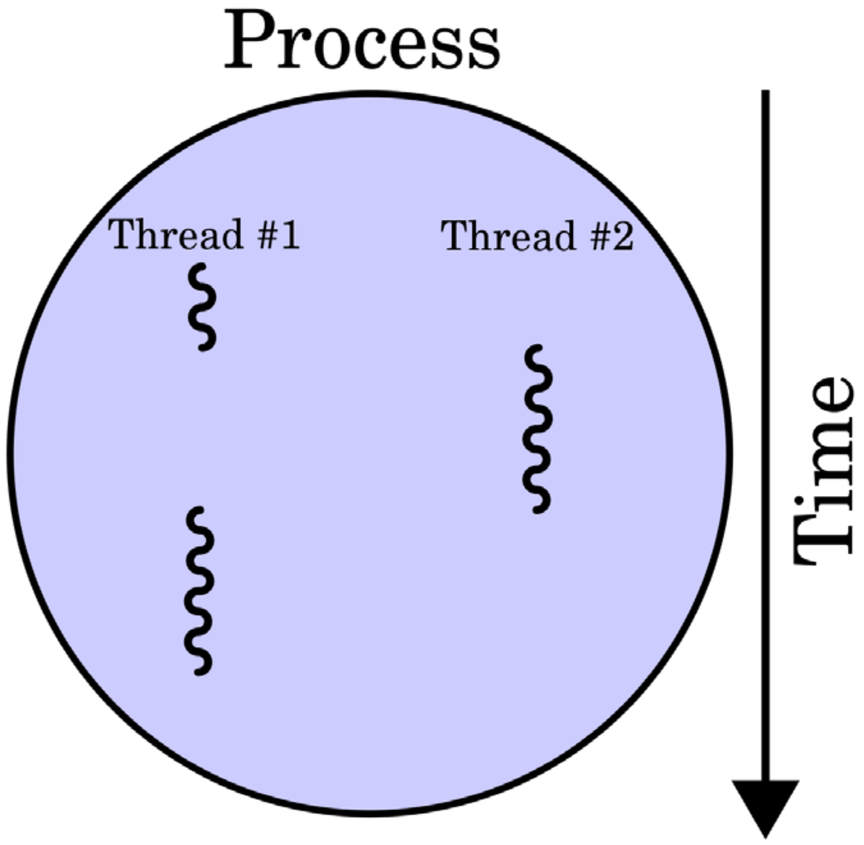
All threads that belong to a process run on the single processor (or CPU Core).

That is, only a single thread can be active at any given time.

So threads are being processes sequentially and not parallel

The Operating System (OS) is responsible for interrupting a thread and switching to a different thread to give the illusion of concurrency. This can happen in the middle of an instruction.

In other words, the OS can pre-empt your thread to make a switch. This is known as **Pre-emptive multitasking.**



What does threading let me do in code?

A thread allows us to execute code concurrently

- Ie: multiple slow IO request

I will often simulate these slow requests in sample code with time.sleep()

In real code time.sleep() is replaced with something useful, like make http requests

How do I write a thread?

There are three ways to create threads:

- Create an object of type Thread and assign a target function to it.
- Inherit from Thread and create your own thread class
- Create a Thread Pool Executor (Used to execute multiple threads in a simplified manner)

Method 1

Create an object of type Thread and assign a target function to it

- `x = threading.Thread(target=thread_function)`

Start the thread – executes code in the thread function

- `x.start()`

Optionally wait for the thread to finish before continuing rest of code

- `x.join()`

```
import threading
import time

def thread_function():
    time.sleep(1)
    print("finished")
```

Method 1

The following code creates three threads, starts and waits for them to finish

```
import threading
import time

def thread_function():
    time.sleep(1)
    print("finished")
```

```
if __name__ == "__main__":
    threads = []
    start = time.time()
    for _ in range(0,3):
        x = threading.Thread(target=thread_function)
        x.start()
        threads.append(x)

    for thread in threads:
        thread.join()
```

Method 2

Inherit from Thread and create your own thread class

- MyThreadClass inherits from threading.Thread
- The code we want to run in the thread is in the overridden run(self) function

The code below creates three MyThreadClass thread objects

- When thread.start() is called, the run(self) function is called in MyThreadClass
- thread.join() waits for all classes to finish before continuing in the main function

```
class MyThreadClass(threading.Thread):
    def __init__(self, io_time):
        super().__init__()
        self.io_time = io_time

    def run(self):
        time.sleep(self.io_time)
```

```
#main
threads = [MyThreadClass(1) for _ in range(3)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
```

Method 3

Create a Thread Pool Executor (Used to execute multiple threads in a simplified manner)

- Thread pool executor allows us to specify a re-usable pool of threads to run some code
- Below we’re creating 3 threads in the executor
- Map the function thread_function to each of the threads
    - thread_function accepts a parameter
    - Thread 1 is mapped to thread_function(0)
    - Thread 2 is mapped to thread_function(1)
    - Thread 3 is mapped to thread_function(2)
- All threads are automatically started, and waited to finish

```
def thread_function(name):
    time.sleep(2)

#main
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
        executor.map(thread_function, range(3))
```

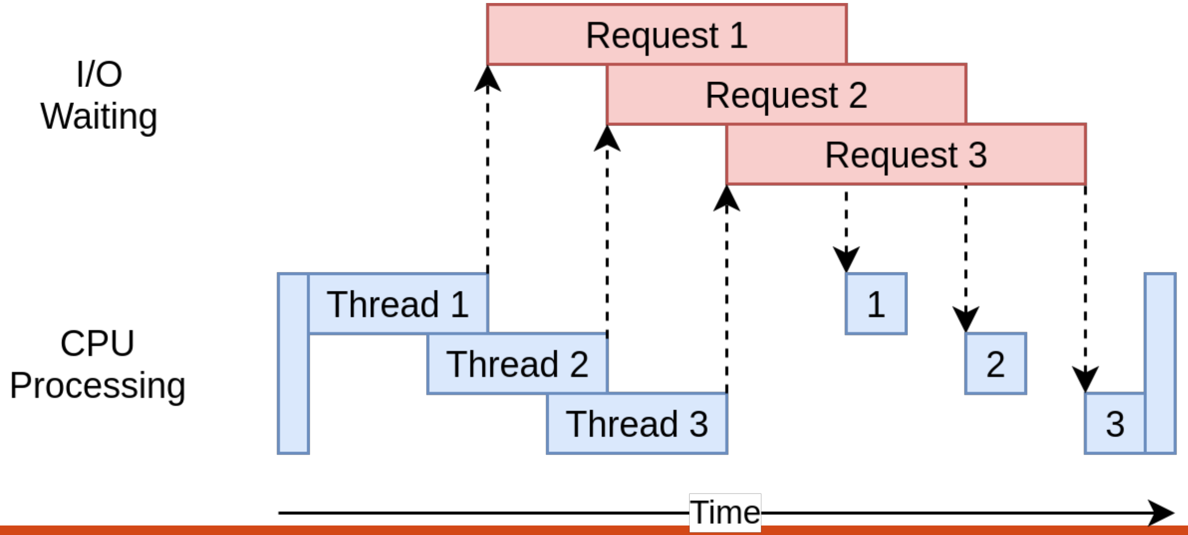
Threads - Advantages

Code runs faster!

Great for I/O Bound situations.

Most languages have support for threading in some form or the other.

They share the same memory space so it's easy to share data.



Threads - Disadvantages

Not so good for CPU Bound situations

Race conditions

Deadlocks

More code to write and can be hard to debug.

ASYNCIO EVENT LOOP

Async IO Event loop

What is it?

The event loop is the core of every asyncio application. Event loops run asynchronous tasks and callbacks, perform network IO operations, and run subprocesses.

Why do we use it?

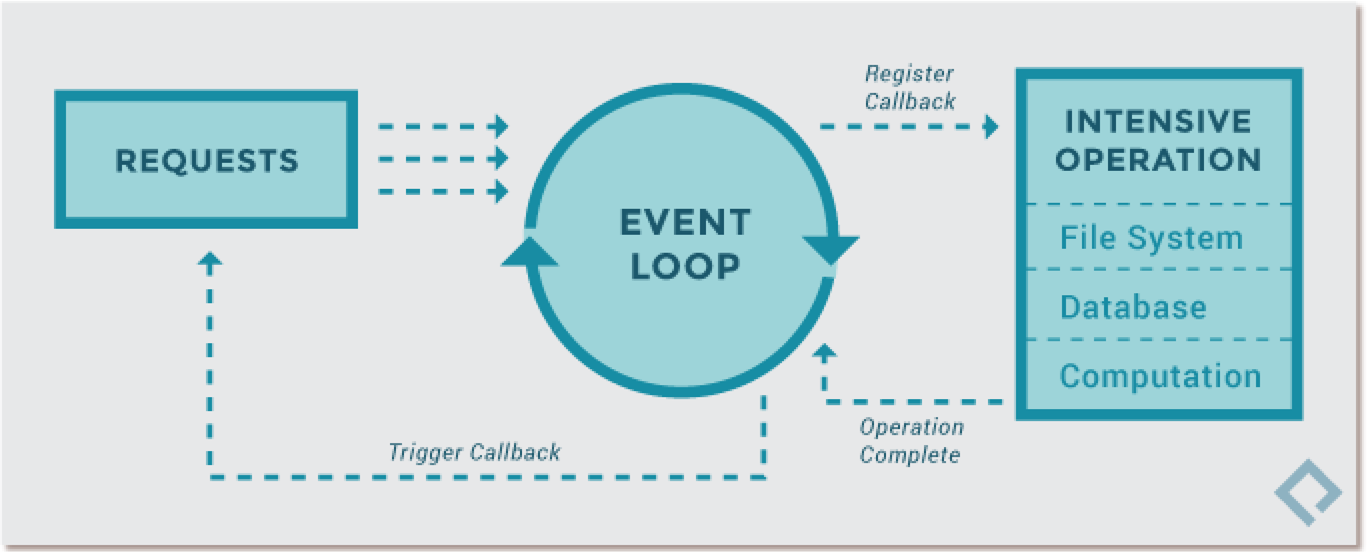
When we want to write code that executes concurrently, but have the code manage handing off processing time vs the OS managing processing time

How do we use it?

Let's look at an example

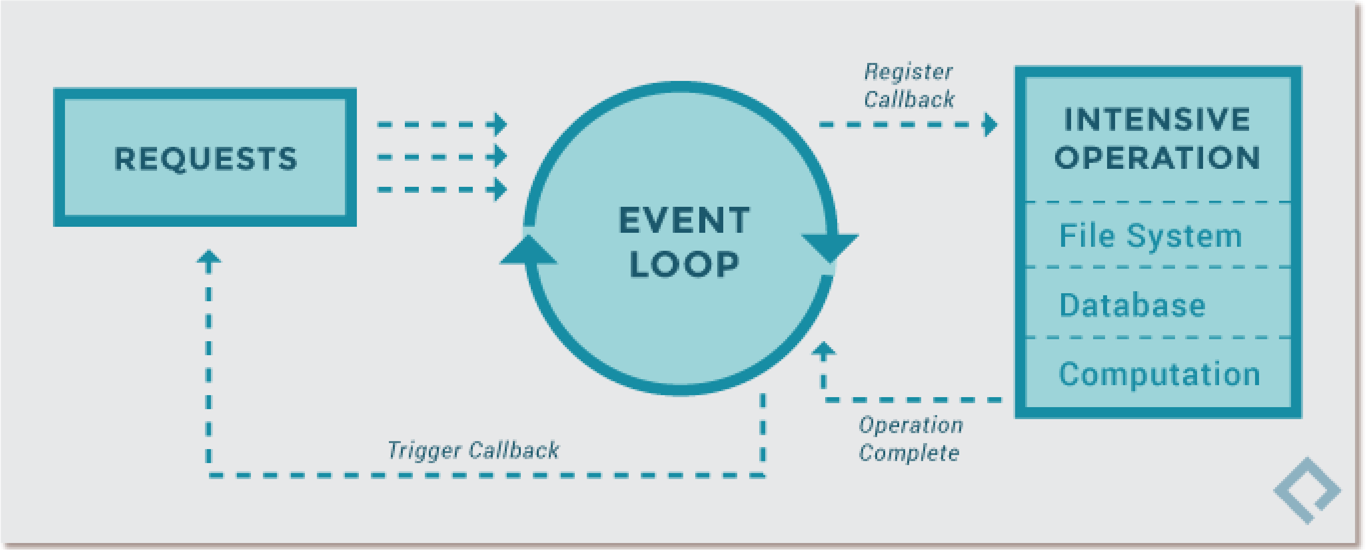
Async IO Event loop

Let's write some code to simulate accessing a database and filesystem at the same time using AsyncIO



Async IO Event loop

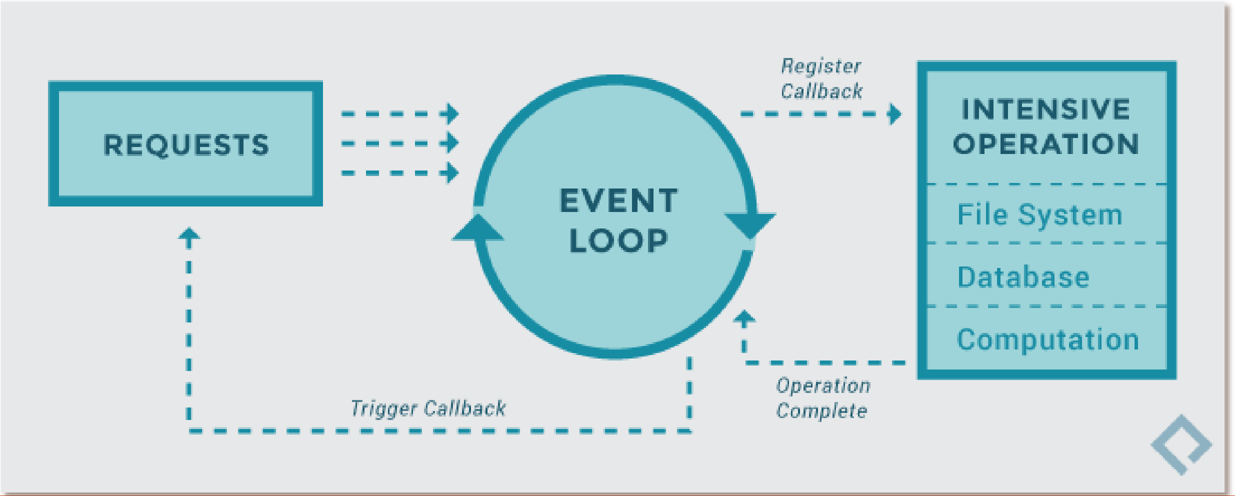
```
import asyncio

async def do_something_1():
    print("accessing database")
    await asyncio.sleep(1)

async def do_something_2():
    print("accessing file system")
    await asyncio.sleep(1)

async def main():
    await asyncio.gather(do_something_1(), do_something_2())

if __name__ == '__main__':
    asyncio.run(main())
```



Async IO Event loop

```python
if __name__ == '__main__':
    asyncio.run(main())
```

`asyncio.run(main())` – starts the event loop and runs the coroutine main function



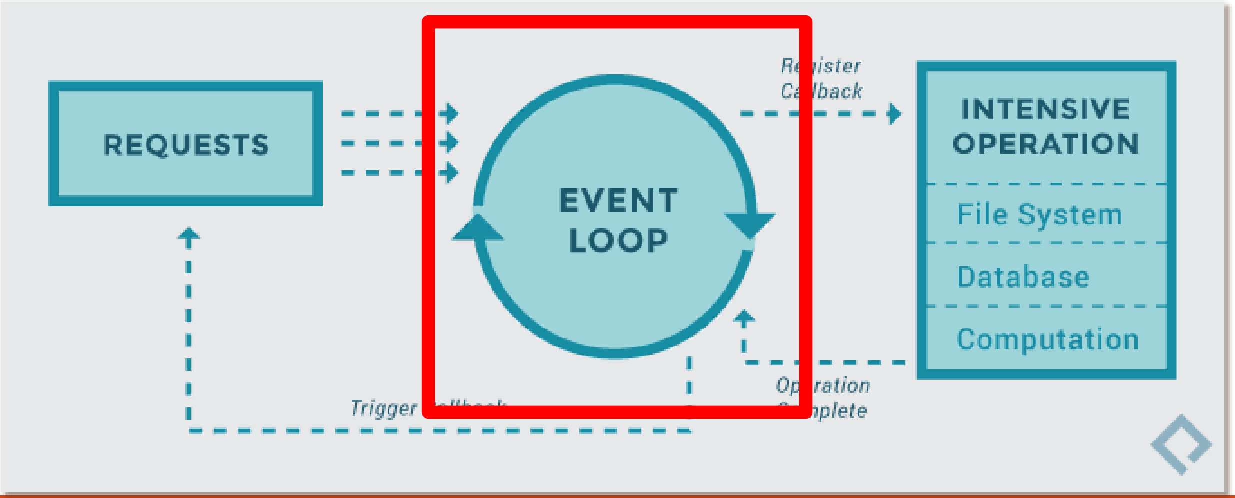
Async IO Event loop

```python
async def main():
    await asyncio.gather(do_something_1(), do_something_2())
```

Before executing `asyncio.gather`, what's happening in the background is the Event Loop is running

Because we ran `asyncio.run` on the main function

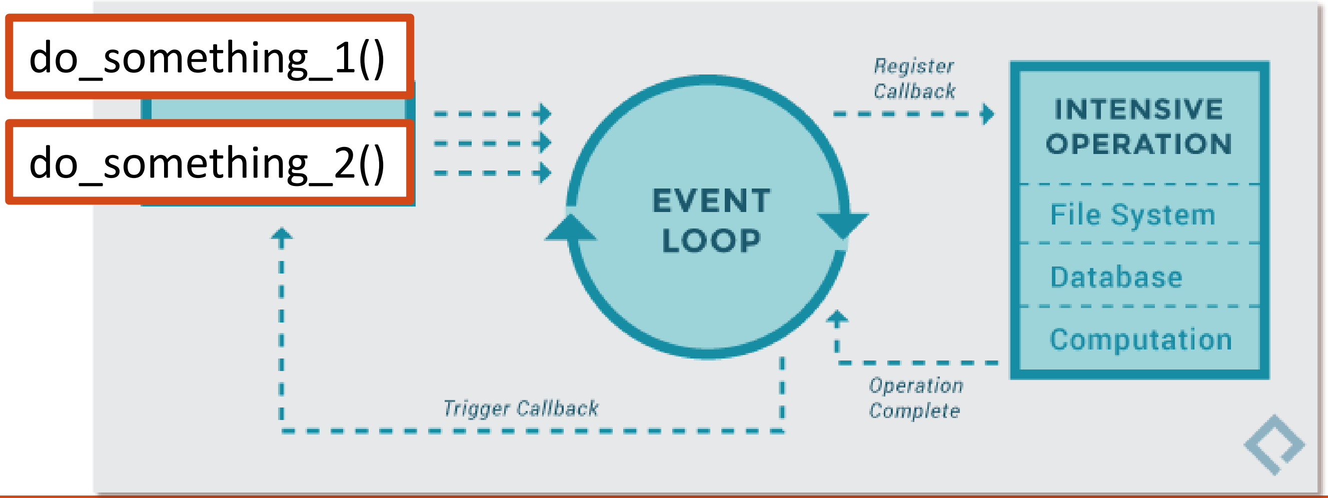
- while we're in main and its sub-functions, the event loop is constantly running, waiting for requests to arrive



Async IO Event loop

```
async def main():
    await asyncio.gather(do_something_1(), do_something_2())
```

- When calling `asyncio.gather`, we want to run `do_something_1()` and `do_something_2()` concurrently
- In the diagram below, do_something_1() and `do_something_2()` are requests, waiting for the event loop to run them

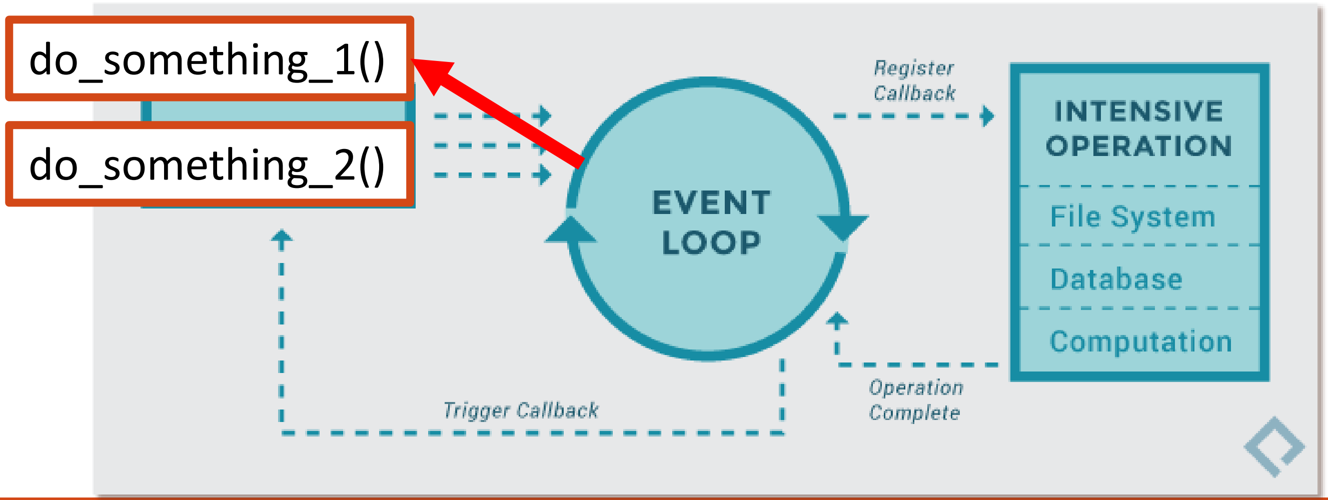


Async IO Event loop

```
async def main():
    await asyncio.gather(do_something_1(), do_something_2())
```

- The event loop gets `do_something_1()` and executes its code
- But accessing the database is going to take some time (`asyncio.sleep(1)`)

```
async def do_something_1():
    print("accessing database")
    await asyncio.sleep(1)
```

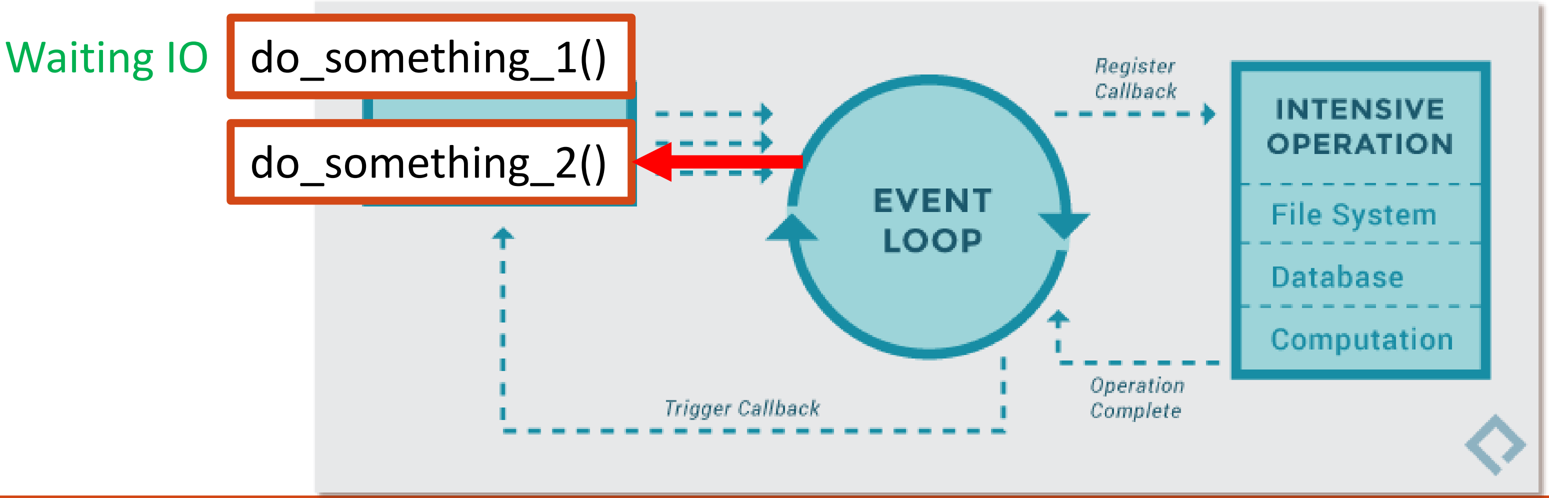


Async IO Event loop

```python
async def main():
    await asyncio.gather(do_something_1(), do_something_2())
```

- The event loop gets `do_something_1()` and executes its code
- But accessing the database is going to take some time (`asyncio.sleep(1)`)
- While `do_something_1()` is waiting for its IO operation, the event loop changes to a different waiting request

```python
async def do_something_1():
    print("accessing database")
    await asyncio.sleep(1)
```

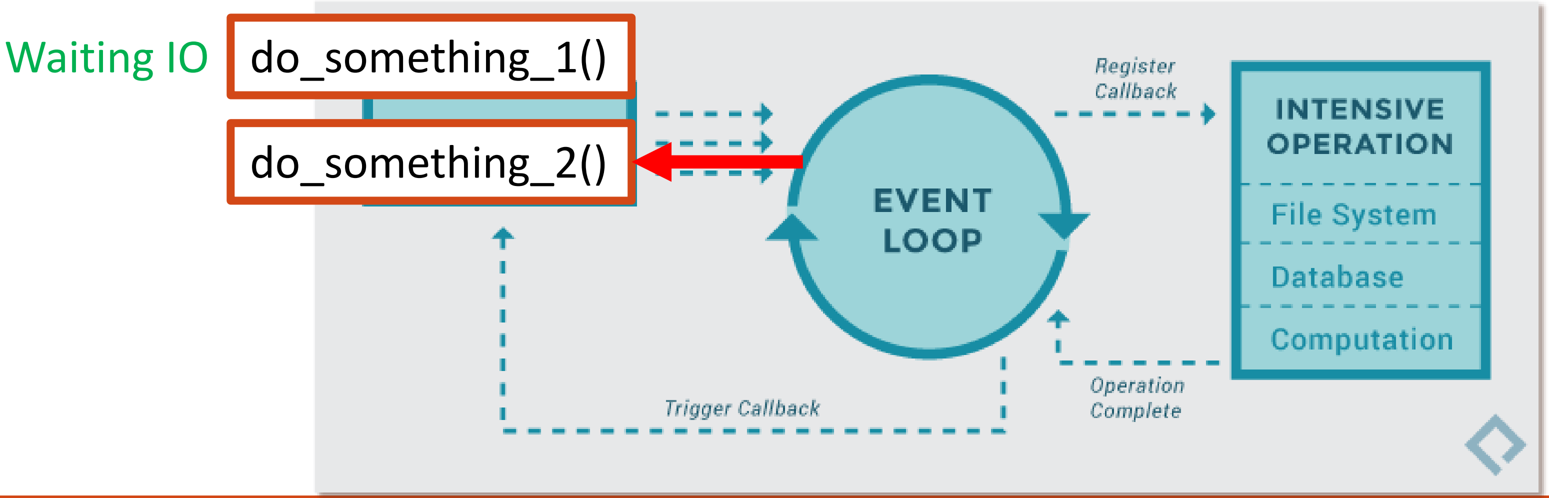


Async IO Event loop

```
async def main():
    await asyncio.gather(do_something_1(), do_something_2())
```

- The event loop gets `do_something_2()` and executes its code
- But accessing the file system is going to take some time (`asyncio.sleep(1)`)

```
async def do_something_2():
    print("accessing file system")
    await asyncio.sleep(1)
```

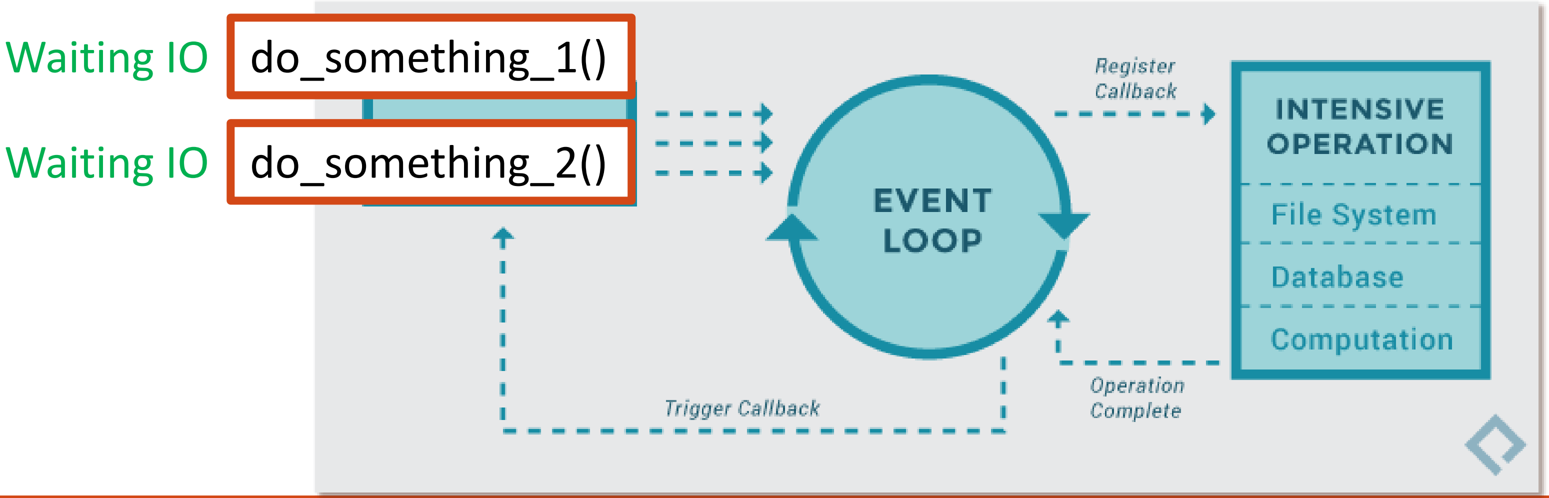


Async IO Event loop

```
async def main():
    await asyncio.gather(do_something_1(), do_something_2())
```

- The event loop gets `do_something_2()` and executes its code
- But accessing the file system is going to take some time (`asyncio.sleep(1)`)
- While `do_something_2()` is waiting for its IO operation, the event loop changes to a different waiting request

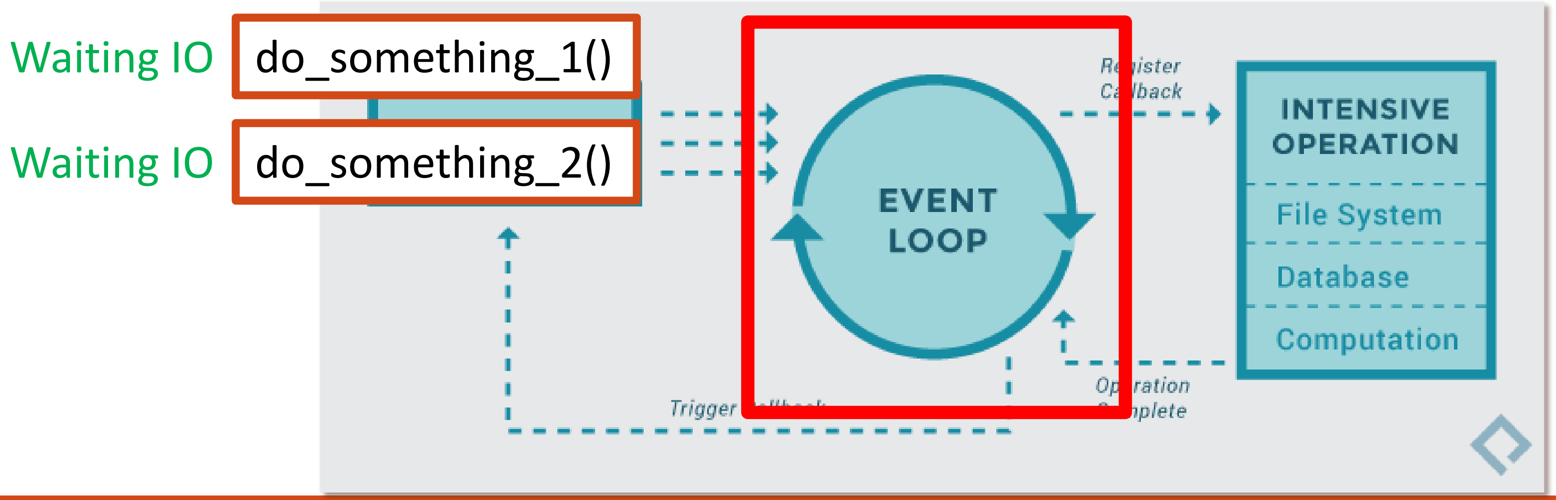
```
async def do_something_2():
    print("accessing file system")
    await asyncio.sleep(1)
```



Async IO Event loop

```python
async def main():
    await asyncio.gather(do_something_1(), do_something_2())
```

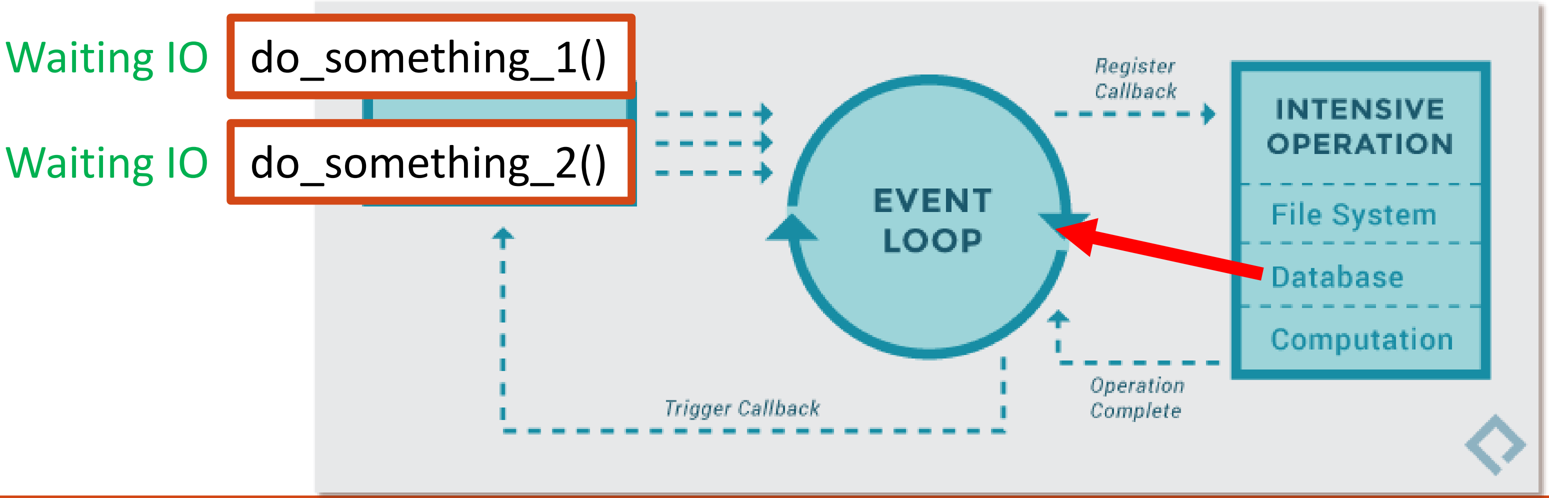
- Both functions are busy waiting, so the event loop keeps looping, waiting for new requests, or when old requests need to be woken up



Async IO Event loop

```
async def main():
    await asyncio.gather(do_something_1(), do_something_2())
```

- Eventually the database responds and needs to callback do_something_1()

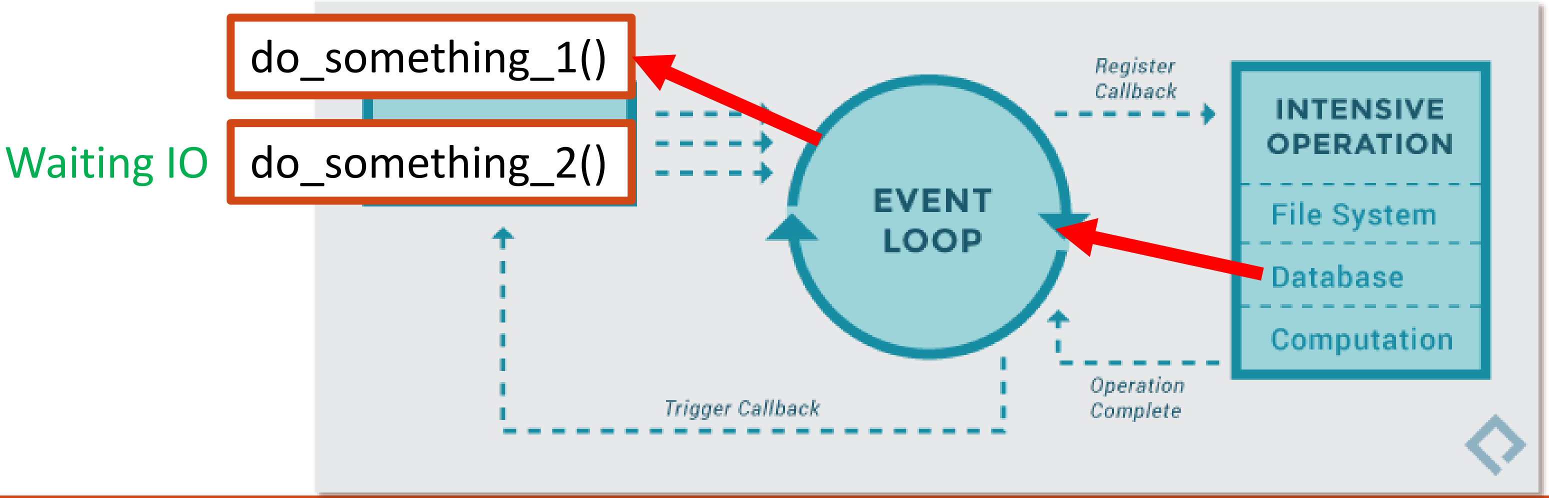


Async IO Event loop

```python
async def main():
    await asyncio.gather(do_something_1(), do_something_2())
```

- Eventually the database responds and needs to callback do_something_1()
- The event loop gets do_something_1() and executes the rest of its code
- There is no more code after the IO request, so the function finishes

```python
async def do_something_1():
    print("accessing database")
    await asyncio.sleep(1)
```

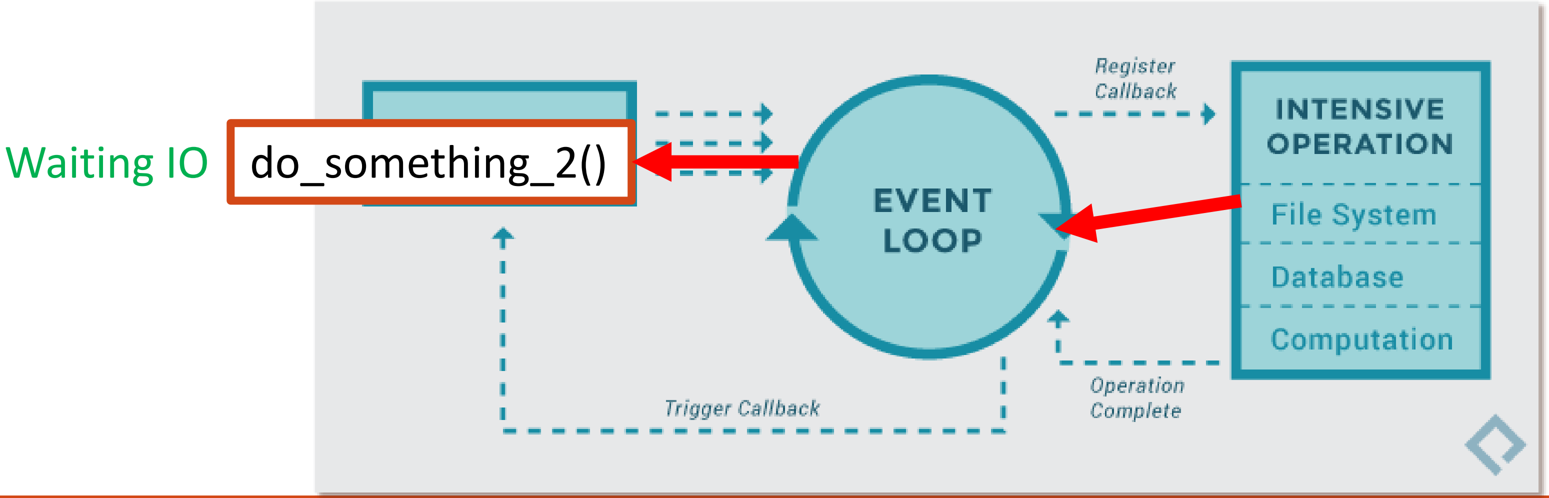


Async IO Event loop

```
async def main():
    await asyncio.gather(do_something_1(), do_something_2())
```

- Eventually the file system responds and needs to callback do_something_2()
- The event loop gets do_something_2() and executes the rest of its code
- There is no more code after the IO request, so the function finishes

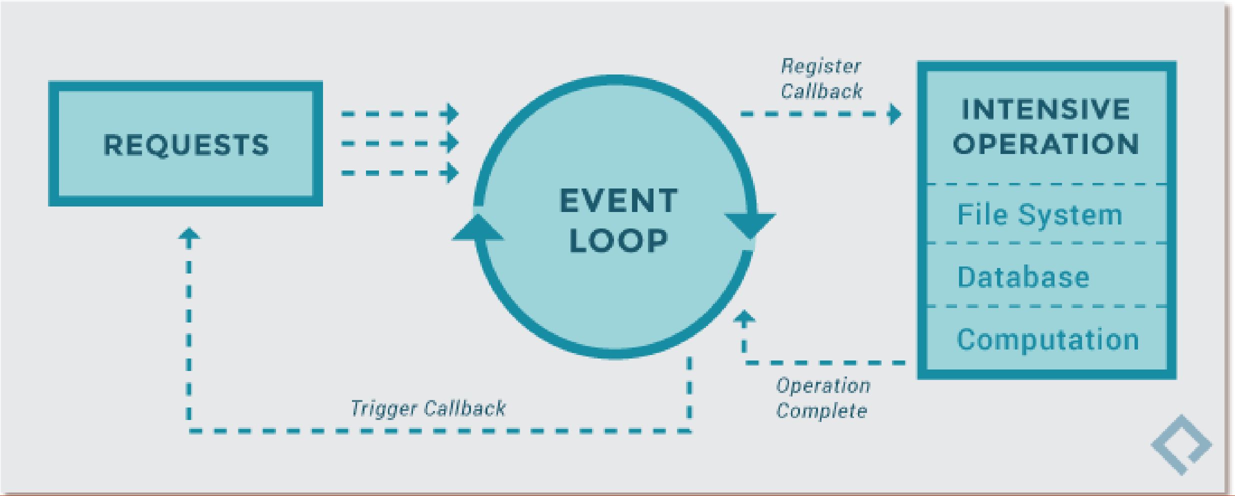
```
async def do_something_2():
    print("accessing file system")
    await asyncio.sleep(1)
```



Async IO Event loop

```python
if __name__ == '__main__':
    asyncio.run(main())
```

- We've returned from main and completed asyncio.run(main())
- At this point the event loop is finished



That’s it for today!

Review continues next class

Submit questions to: https://forms.gle/RhVNyoTm2iQbAsPD6

Can submit multiple questions, but be specific