- Asynchrony in Python
- Coroutines
- The asyncio module
- Simple example of asynchrony

Asynchrony in Python

What is asynchrony?
- The ability to perform multiple tasks concurrently

Scenarios where asynchrony is important:
- Processing a large dataset in parallel
- Handling multiple network connections simultaneously
- Performing algorithmic processing in the background
- Etc.

You can schedule concurrent tasks on a single thread
- The event loop manages task execution on a thread

The event loop can optimize I/O
- If a function is waiting on I/O...
- The event loop pauses the function and runs another one instead...
- When the first function completes I/O, it is resumed

The event loop can also optimize CPU-intensive functions
- The functions must explicitly "yield", so as not to hog the thread

Note: Python also supports genuine multithreading

Ex1

Coroutines

A coroutine is a special kind of generator function
- It can cede control during its processing (e.g. for I/O)
- The event loop then tries to give another coroutine some time
- The event loop can resume the original coroutine when it's ready

The preferred way to define a coroutine in modern Python is to prefix a function with the async keyword

```
async def someFunc(someArgs) :
    # Some long-running code that might yield control
    #   e.g. code that does slow I/O
    #   e.g. code that CPU-intensive processing
```

The asyncio Module

The asyncio module provides various methods that allow you to schedule and manage asynchrony
- Some of the common methods are listed here...

asyncio.sleep(seconds)
- Sleep for a specified delay (in seconds)

asyncio.run(aCoroutine)
- Creates a new event loop, and runs the coroutine

asyncio.create_task()
- Schedule a coroutine to be executed "soon" on the event loop

Simple Example of Asynchrony

Ex2

- asyncio.sleep() is a coroutine
- The await keyword yields control back to the event loop, which tries to schedule other coroutines in the meantime
- You can only use the await keyword in coroutines, i.e. functions marked as async
- You can't just 'invoke' coroutines, you must schedule via asyncio

- Simple example of creating a task
- Creating and awaiting multiple tasks
- Awaiting multiple tasks to complete

The asyncio.create_task() function creates a task
- The task is schedules for execution "soon" on the event loop
- The task is represented by a Task object

The Task class has methods that allow you to manage the running of the task, such as:
- done() - has the task completed yet?
- cancel() - stop the task now
- result() - get the result of the task (it must have finished!)

Ex3

Creating Multiple Tasks

You can create multiple tasks
- All the tasks run concurrently
- You can await for each task to complete individually

Ex4

Awaiting Multiple Tasks to Complete

The previous example awaited individual tasks to complete
- If you prefer, you can await multiple tasks to complete
- Use asyncio.gather(), which suspends until all tasks are done

Ex5

- Awaiting the result of a task
- Polling a task to see if it's done
- Cancelling a task

Awaiting the Result of a Task

A coroutine can return a value
- The calling code would like to retrieve the value when complete

Here's one way for the calling code to do this:
- Create a task, to schedule the coroutine for execution
- Await completion of the task
- The await expression gives the result of the completed coroutine

Ex6

If it's more convenient, you can combine these two statements into a single statement

Ex7

Polling a Task to See if it's Done

Sometimes you might want to poll a task to see if it's done
- Call done() on the task, to see if it's finished
- If it hasn't finished, do something else for a bit, then check again
- When it really has finished, call result() on the task

Ex8

Cancelling a Task

Sometimes you might want to cancel a task mid-flight
- Call cancel() on the task

Ex9