Asynchronous Lab

What is Asynchronous Programming

- Asynchronous programming is a programming paradigm that **does not block.**
- Instead, **requests** and function calls are issued and executed **somehow** in the **background** at some **future** time
- This frees the caller to perform other activities and **handle** the results of issued calls at a **later time** when **results** are **available** or when the caller is interested

Asynchronous Task

- **Asynchronous Function Call** Request that a function is called at some time and in some manner, allowing the caller to resume and perform other activities
- **Future**: A handle on an asynchronous function call allowing the status of the call to be checked and results to be retrieved.
- **Asynchronous Task**
  - The combination of the **asynchronous function call** and **future** together is often referred to as an asynchronous task. This is because it is more elaborate than a function call, such as allowing the request to be canceled and more

Asynchronous Programming

- The use of asynchronous techniques, such as issuing **asynchronous tasks** or **function calls**.
- Asynchronous programming is primarily used with **non-blocking I/O**, such as reading and writing from socket connections with other processes or other systems.
- **Non-blocking I/O**: Performing I/O operations via asynchronous requests and responses, **rather** than **waiting** for operations to complete.
- **Asynchronous I/O**: A shorthand that refers to combining asynchronous programming with non-blocking I/O.

Asynchronous Programming in Python

- Asynchronous programming in Python refers to making requests and not blocking to wait for them to complete.
- **asyncio** is short for asynchronous I/O. It is a Python library that allows us to run code using an asynchronous programming model. This lets us handle multiple I/O operations at once, while still allowing our application to remain responsive.

What is Asyncio

- Python 3.4 introduced the **asyncio** library, and Python 3.5 produced the **async** and **await** keywords to use it palatably.
- These new additions allow so-called **asynchronous programming**

Changes to Python to add Support for Coroutines

- More specifically, it was changed to support coroutines as first-class concepts. In turn, coroutines are the unit of concurrency used in asyncio programs.
- A coroutine is a function that can be suspended and resumed
  - **coroutine**: Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different point
  - A coroutine can be defined via the **“async def”** expression. It can take arguments and return a value, just like a function

```
# define a coroutine
async def custom_coro():
    # ...
```

Calling a coroutine function will create a coroutine object, this is a new class. It does not execute the coroutine function.

```
...
# create a coroutine object
coro = custom_coro()
```

A coroutine can execute another coroutine via the await expression.

This suspends the caller and schedules the target for execution.

```
...
# suspend and schedule the target
await custom_coro()
```

An asynchronous iterator is an iterator that yields awaitables.

The asyncio Module

- The “asyncio” module **provides functions** and **objects** for developing **coroutine**-based programs using the asynchronous programming paradigm
- Central to the asyncio module is the **event loop.**
-

When to Use Asyncio

- Use asyncio in order to adopt **coroutines** in your program.
- Use asyncio in order to use the **asynchronous programming** paradigm.
- Use asyncio in order to use **non-blocking I/O**

- Threads and processes achieve multitasking via the operating system that chooses which threads and processes should run, when, and for how long. The operating switches between threads and processes rapidly, suspending those that are not running and resuming those granted time to run. This is called **preemptive multitasking.**

- Coroutines in Python provide an alternative type of multitasking called **cooperating multitasking**

Non-Blocking I/O

- **Hard disk drives:** Reading, writing, appending, renaming, deleting, etc. files.
- **Peripherals**: mouse, keyboard, screen, printer, serial, camera, etc.
- **Internet**: Downloading and uploading files, getting a webpage, querying RSS, etc.
- **Database**: Select, update, delete, etc. SQL queries.
- **Email**: Send mail, receive mail, query inbox, etc.

Other reasons

- Perhaps you need to use a **third-party API** and the code examples use asyncio.
- Perhaps you need to integrate an **existing open-source solution** that uses asyncio.
- Perhaps you stumble across some code snippets that do what you need, yet they use asyncio.

Coroutines in Python

- A coroutine is a function that can be **suspended** and **resumed**.
- A **subroutine** can be executed, starting at one point and finishing at another point. Whereas, a **coroutine** can be executed then suspended, and resumed many times before finally terminating.

Coroutine vs Task

- A coroutine can be wrapped in an **asyncio.Task** object and executed independently
- The **Task** object provides a handle on the asynchronously execute coroutine
- **Task**: A wrapped coroutine that can be executed independently
- A **Task** cannot exist on its own, it must wrap a coroutine.
- Therefore a **Task** is a coroutine, but a coroutine is not a task

Coroutine vs Thread

- A coroutine is **defined** as a **function**.
- **Thread**: Managed by the operating system, represented by a **Python object**.
  - This means that coroutines are typically **faster** to **create** and start executing and take up less memory.
  - Conversely, **threads** are **slower** than coroutines to create and start and take up more memory.

Coroutine vs Process

- A coroutine is more **lightweight** than a **process.**
- In fact, a **thread** is more lightweight than a **process.**
- A **process** is a computer program. It may have one or **many threads.**
- A **Python process** is in fact a separate instance of the Python interpreter.
  - Process: Managed by the operating system, represented by a Python object

Define, Create and Run Coroutines

```
1 # define a coroutine
2 async def custom_coro():
3     # ...
```

A coroutine defined with the "**async def**" expression is referred to as a "*coroutine function*".

> "*coroutine function: A function which returns a coroutine object. A coroutine function may be defined with the async def statement, and may contain await, async for, and async with keywords.*
>
> — **PYTHON GLOSSARY**

A coroutine can then use coroutine-specific expressions within it, such as **await**, **async for**, and **async with**.

> *Execution of Python coroutines can be suspended and resumed at many points (see coroutine). await expressions, async for and async with can only be used in the body of a coroutine function.*
>
> — **COROUTINE FUNCTION DEFINITION**

For example:

```
# define a coroutine
async def custom_coro():
    # await another coroutine
    await asyncio.sleep(1)
```

How to Create a Coroutine

```
...
# create a coroutine
coro = custom_coro()
```

This does not execute the coroutine.

It returns a **"coroutine"** object.

> *You can think of a coroutine function as a factory for coroutine objects; more directly, remember that calling a coroutine function does not cause any user-written code to execute, but rather just builds and returns a coroutine object.*
>
> **— PAGE 516, PYTHON IN A NUTSHELL, 2017.**

How to Run a Coroutine From Python

- Coroutines can be **defined** and **created**, but they can only be executed **within** an **event loop**
- The **event loop** that executes coroutines, manages the **cooperative multitasking** between coroutines.
- The typical way to start a coroutine event loop is via the **asyncio.run()** function.

```python
# example of runing a coroutine
import asyncio

# define a coroutine
async def custom_coro():
    # wait for a tak to be done
    # await another coroutine
    await asyncio.sleep(1)

# main coroutine
async def main():
    # execute my custom coroutine
    await custom_coro()

# start the coroutine programs
asyncio.run(main())
```

What is the Event Loop

- The event loop is an **environment** for executing coroutines in a **single thread**
- The event loop, as its name suggests, is a **loop**. It **manages** a list of tasks (**coroutines**) and attempts to **progress** each in **sequence** in each iteration of the loop, as well as perform other tasks like executing callbacks and handling I/O

Create and Run Asyncio Tasks

- You can **create Task** objects from **coroutines** in **asyncio programs**
- Tasks provide a handle on independently **scheduled** and **running coroutines** and allow the **task to be queried**, **canceled**, and **results** and exceptions to be retrieved later
- The asyncio event loop **manages tasks.** As such, all coroutines become and are managed as tasks within the event loop

What is an Asyncio Task

- A Task is an object that schedules and independently runs an asyncio coroutine
- A task is created from a coroutine. It requires a coroutine object, wraps the coroutine, schedules it for execution, and provides ways to interact with it
-

Create Task with High-Level API

- The asyncio.create_task() function takes a coroutine instance and an optional name for the task and returns an asyncio.Task instance
  - Wrap the coroutine in a Task instance.
  - Schedule the task for execution in the current event loop.
  - Return a Task instance

```
...
# create a coroutine
coro = task_coroutine()
# create a task from a coroutine
task = asyncio.create_task(coro)
```

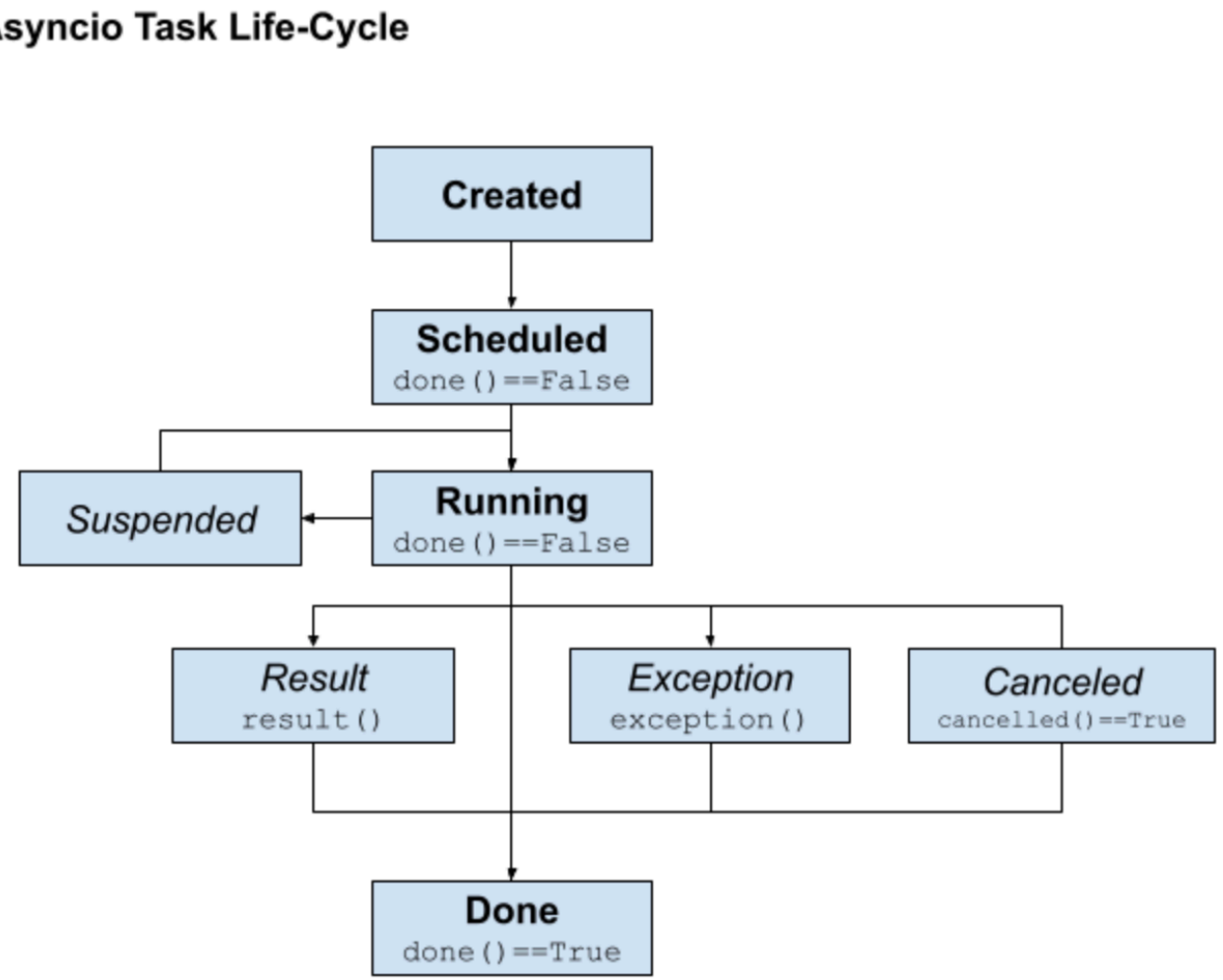
```
...
# create a task from a coroutine
task = asyncio.create_task(task_coroutine())
```

When Dose a Task Run?

- the task will not execute until the event loop has an opportunity to run.
-

```
...
# create a task from a coroutine
task = asyncio.create_task(task_coroutine())
# await the task, allowing it to run
await task
```

Asyncio Task Life-Cycle



Check if a Task is Done

We can check if a task is done via the **done()** method.

The method returns **True** if the task is done, or **False** otherwise.

For example:

```
...
# check if a task is done
if task.done():
    # ...
```

Check if a Task is Canceled

We can check if a task is canceled via the **cancelled()** method.

The method returns **True** if the task was canceled, or **False** otherwise.

For example:

```
...
# check if a task was canceled
if task.cancelled():
    # ...
```

A task is canceled if the **cancel()** method was called on the task and completed successfully, e..g **cancel()** returned **True**.

How to Get the Current Task

```
# example of getting the current task from the main coroutine
import asyncio

# define a main coroutine
async def main():
    # report a message
    print('main coroutine started')
    # get the current task
    task = asyncio.current_task()
    # report its details
    print(task)

# start the asyncio program
asyncio.run(main())
```

How to Get All Tasks

```python
# example of starting many tasks and getting access to all tasks
import asyncio
import time

# coroutine for a task
async def task_coroutine(value):
    # report a message
    print(f'{time.ctime()} task {value} is running')
    # block for a moment
    await asyncio.sleep(1)

# define a main coroutine
async def main():
    # report a message
    print(f'{time.ctime()} main coroutine started')
    # start many tasks
    started_tasks = [asyncio.create_task(task_coroutine(i)) for i in range(10)]
    # allow some of the tasks time to start
    await asyncio.sleep(0.1)
    # get all tasks
    tasks = asyncio.all_tasks()
    # report all tasks
    for task in tasks:
        print(f'{time.ctime()} > {task.get_name()}, {task.get_coro()}')
    # wait for all tasks to complete
    for task in started_tasks:
        await task

# start the asyncio program
asyncio.run(main())
```

Run Many Coroutines Concurrently

- A benefit of asyncio is that we can run many coroutines concurrently.
- These coroutines can be created in a group and stored, then executed all together at the same time.

What is Asyncio gather()

- The asyncio.gather() module function allows the caller to group multiple awaitables together.
- Once grouped, the awaitables can be executed concurrently, awaited, and canceled.

```python
# example of gather for many coroutines in a list
import asyncio
import time

# coroutine use for a tak
async def task_coro(value):
    # report a message
    print(f'{time.ctime()} task {value} executing')
    # sleep for a moment
    await asyncio.sleep(1)

# coroutine used for the entry point
async def main():
    # report a message
    print(f'{time.ctime()} main starting.')
    # create many coroutines
    coros = [task_coro(i) for i in range(10)]
    # run the tasks
    await asyncio.gather(*coros)
    # report a message
    print(f'{time.ctime()} main done')

# start the asyncio program
asyncio.run(main())
```

```
Mon Jul 10 23:00:08 2023 main starting.
Mon Jul 10 23:00:08 2023 task 0 executing
Mon Jul 10 23:00:08 2023 task 1 executing
Mon Jul 10 23:00:08 2023 task 2 executing
Mon Jul 10 23:00:08 2023 task 3 executing
Mon Jul 10 23:00:08 2023 task 4 executing
Mon Jul 10 23:00:08 2023 task 5 executing
Mon Jul 10 23:00:08 2023 task 6 executing
Mon Jul 10 23:00:08 2023 task 7 executing
Mon Jul 10 23:00:08 2023 task 8 executing
Mon Jul 10 23:00:08 2023 task 9 executing
Mon Jul 10 23:00:09 2023 main done
```

Wait for A Collection of Tasks

- The asyncio.wait() function takes a collection of awaitables, typically Task objects.

  This could be a **list**, **dict,** or **set** of task objects that we have created, such as via calls to the **asyncio.create_task()** function in a list comprehension.

  For example:

```
...
# create many tasks
tasks = [asyncio.create_task(task_coro(i)) for i in range(10)]
```

  The **asyncio.wait()** will not return until some condition on the collection of tasks is met.

Waiting for All Tasks

```python
# example of waiting for all tasks to complete
import random
import asyncio
import time

# coroutine to execute in a new task
async def task_coro(arg):
    # generate a random value between 0 and 1
    value = random.random()
    # block for a moment
    await asyncio.sleep(value)
    # report the value
    print(f'{time.ctime()} >task {arg} done with {value}')

# main coroutine
async def main():
    # create many tasks
    tasks = [asyncio.create_task(task_coro(i)) for i in range(10)]
    # wait for all tasks to complete # ALL_COMPLETED, FIRST_COMPLETED, FIRST_EXCEPTION
    done, pending = await asyncio.wait(tasks, return_when=asyncio.ALL_COMPLETED)
    # report result
    print(f'{time.ctime()} All done')

# start the asyncio program
asyncio.run(main())
```

```
Mon Jul 10 23:04:03 2023 >task 1 done with 0.06272708357096457
Mon Jul 10 23:04:04 2023 >task 3 done with 0.5542213402968057
Mon Jul 10 23:04:04 2023 >task 9 done with 0.6969553002921544
Mon Jul 10 23:04:04 2023 >task 0 done with 0.7130917658727823
Mon Jul 10 23:04:04 2023 >task 7 done with 0.7169015805286069
Mon Jul 10 23:04:04 2023 >task 4 done with 0.7485573996352604
Mon Jul 10 23:04:04 2023 >task 5 done with 0.7802128788125767
Mon Jul 10 23:04:04 2023 >task 6 done with 0.8736479282730453
Mon Jul 10 23:04:04 2023 >task 2 done with 0.8881288694371754
Mon Jul 10 23:04:04 2023 >task 8 done with 0.8909549946543802
Mon Jul 10 23:04:04 2023 All done
```

Wait for a Coroutine with a Time Limit

- The asyncio.wait_for() function takes an awaitable and a timeout.

```python
# example of waiting for a coroutine with a timeout
from random import random
import asyncio
import time

# coroutine to execute in a new task
async def task_coro(arg):
    # generate a random value between 0 and 1
    value = 1 + random()
    # report message
    print(f'{time.ctime()} >task got {value}')
    # block for a moment
    await asyncio.sleep(value)
    # report all tome
    print(f'{time.ctime()} >task done')

# main coroutine
async def main():
    # create a task
    task = task_coro(1)
    #execute and wait for the task without a timeout
    try:
        await asyncio.wait_for(task, timeout=0.2)
    except asyncio.TimeoutError:
        print(f'{time.ctime()} Gave up waiting, task canceled')

# start the asyncio program
asyncio.run(main())
```

```
Mon Jul 10 23:04:53 2023 >task got 1.9568179927895635
Mon Jul 10 23:04:54 2023 Gave up waiting, task cancele
```

Shield Tasks from Cancellation

- Asyncio tasks can be canceled by calling their cancel() method.
- We can protect a task from being canceled by wrapping it in a call to asyncio.shield()
- Importantly, the request for cancellation made on the Future object is not propagated to the inner task.
- If the task that is being shielded is canceled, the cancellation request will be propagated up to the shield, which will also be canceled.

```python
# example of using asyncio shield to protect a task from cancellation
import time
import asyncio

# define a simple asynchronous
async def simple_task(number):
    # block for a moment
    await asyncio.sleep(1)
    # return the argument
    return number

# cancel the given task after a moment
async def cancel_task(task):
    # block for a moment
    await asyncio.sleep(0.2)
    # cancel the task
    was_canceled = task.cancel()
    print(f'{time.ctime()} canceled: {was_canceled}')

# define a simple corotine
async def main():
    # create the coroutine
    coro = simple_task(1)
    # create a task
    task = asyncio.create_task(coro)
    # create the shielded task
    shielded = asyncio.shield(task)
    # create the task to cancel the previous task
    asyncio.create_task(cancel_task(shielded))
    # handle cancellation
    try:
        # await the shielded task
        result = await shielded
        print(f'{time.ctime()} >got: {result}')
    except asyncio.CancelledError:
        print(f'{time.ctime()} shielded was cancel')
    # wait a moment
    await asyncio.sleep(1)
    # report the details of the tasks
    print(f'{time.ctime()} shielded: {shielded}')
    print(f'{time.ctime()} task: {task}')

# start the loop
asyncio.run(main())

```

```
Mon Jul 10 23:09:28 2023 canceled: True
Mon Jul 10 23:09:28 2023 shielded was cancel
Mon Jul 10 23:09:29 2023 shielded: <Future cancelled>
Mon Jul 10 23:09:29 2023 task: <Task finished name='Task-2' coro=<simple_task() done, defined
```

Run a Blocking Task in Asyncio

- A blocking task is a task that stops the current thread from progressing.
- If a blocking task is executed in an asyncio program it stops the entire event loop, preventing any other coroutines from progressing.
- We can run blocking calls asynchronously in an asyncio program via the asyncio.to_thread() and loop.run_in_executor() functions.

```python
# example of running a blocking io-bound task in asyncio
import asyncio
import time

# a blocking io-bound task
def blocking_task():
    # report a message
    print(f'{time.ctime()} Task starting...')
    # block for a while
    time.sleep(2)
    # report a message
    print(f'{time.ctime()} Task done')

# main coroutine
async def main():
    # report a message
    print(f'{time.ctime()} Main running the blocking task')
    # create a coroutine for the blocking task
    coro = asyncio.to_thread(blocking_task)
    # schedule the task
    task = asyncio.create_task(coro)
    # report a message
    print(f'{time.ctime()} Main doing other things')
    # allow the scheduled task to start
    await asyncio.sleep(0)
    # await the task
    await task

# run the asyncio program
asyncio.run(main())
```

```
Mon Jul 10 23:11:11 2023 Main running the blocking task
Mon Jul 10 23:11:11 2023 Main doing other things
Mon Jul 10 23:11:11 2023 Task starting...
Mon Jul 10 23:11:13 2023 Task done
```

Asynchronous Interators

- **Iteration** is a basic operation in Python
- We can iterate **lists**, **strings**, and all manner of other **structures.**
- Asyncio allows us to develop **asynchronous iterators.**

Iterators

- An iterator is a Python object that implements a specific interface.
- Specifically, the **__iter__()** method that returns an instance of the iterator and the **__next__()** method that steps the iterator one cycle and returns a value.

> *iterator: An object representing a stream of data. Repeated calls to the iterator's __next__() method (or passing it to the built-in function next()) return successive items in the stream. When no more data are available a StopIteration exception is raised instead.*
>
> — **PYTHON GLOSSARY**

Asynchronous Iterators

- An asynchronous iterator is a Python object that implements a specific interface.
- An asynchronous iterator must implement the **__aiter__()** and **__anext__()** methods.
  - The __aiter__() method must return an **instance** of the iterator.
  - The __anext__() method must return an **awaitable** that steps the iterator.

Define an Asychonous Iterator

- We can define an asynchronous iterator by defining a class that implements the **__aiter__()** and **__anext__()** methods.
- These methods are defined on a Python object as per normal.
- Importantly, because the **__anext__()** function must return an **awaitable**, it must be defined using the **“async def”** expression.

Create Asynchronous Iterator

To use an asynchronous iterator we must create the iterator.

This involves creating the Python object as per normal.

```
...
# create the iterator
it = AsyncIterator()
```

This returns an "*asynchronous iterable*", which is an instance of an "*asynchronous iterator*".

Step an Asynchronous Iterator

One step of the iterator can be traversed using the **anext()** built-in function, just like a classical iterator using the **next()** function.

The result is an awaitable that is awaited.

```
...
# get an awaitable for one step of the iterator
awaitable = anext(it)
# execute the one step of the iterator and get the result
result = await awaitable
```

```
...
# step the async iterator
result = await anext(it)
```

Traverse an Asynchronous Iterator

The asynchronous iterator can also be traversed in a loop using the "**async for**" expression that will await each iteration of the loop automatically.

```
...
# traverse an asynchronous iterator
async for result in AsyncIterator():
    print(result)
```

```python
# example of an asynchronous iterator with async for loop
import asyncio
import time

# define an asynchronous iterator
class AsyncIterator():
    # constructor, define some state
    def __init__(self):
        self.counter = 0

    # create an instance of the iterator
    def __aiter__(self):
        return self

    # return the next awaitable
    async def __anext__(self):
        # check for no furture items
        if self.counter >= 10:
            raise StopAsyncIteration
        # increment the counter
        self.counter += 1
        # simulate work
        await asyncio.sleep(1)
        # return the counter value
        return self.counter

# main coroutine
async def main():
    # loop over async iterator with async for loop
    async for item in AsyncIterator():
        print(f'{time.ctime()} {item}')
# execute the asyncio program
asyncio.run(main())
```

```
Tue Jul 11 14:07:33 2023 1
Tue Jul 11 14:07:34 2023 2
Tue Jul 11 14:07:35 2023 3
Tue Jul 11 14:07:36 2023 4
Tue Jul 11 14:07:37 2023 5
Tue Jul 11 14:07:38 2023 6
Tue Jul 11 14:07:39 2023 7
Tue Jul 11 14:07:40 2023 8
Tue Jul 11 14:07:41 2023 9
Tue Jul 11 14:07:42 2023 10
```

Asynchronous Context Managers

- A **context manager** is a Python construct that provides a **try-finally like** environment with a consistent interface and handy syntax, e.g. via the “**with**” expression.
- It is **commonly used** with **resources**, ensuring the resource is **always closed** or **released after we are finished** with it, regardless of whether the usage of the resources was successful or failed with an exception.

Context Manager

- A context manager is a Python object that implements the __enter__() and __exit__() methods.

```
...
# open a context manager
with ContextManager() as manager:
    # ...
# closed automatically
```

```
...
# create the object
manager = ContextManager()
try:
    manager.__enter__()
    # ...
finally:
    manager.__exit__()
```

Asynchronous Context Manager

- The **__aenter__** and **__aexit__** methods are defined as coroutines and are awaited by the caller.
- This is achieved using the “**async with**” expression.

```
...
# create and use an asynchronous context manager
async with AsyncContextManager() as manager:
    # ...
```

```
...
# create or enter the async context manager
manager = await AsyncContextManager()
try:
    # ...
finally:
    # close or exit the context manager
    await manager.close()
```

How to use Asynchronous Context Manager

- We can define an asynchronous context manager as a Python object that implements the **__aenter__()** and **__aexit__()** methods.

```
# define an asynchronous context manager
class AsyncContextManager:
    # enter the async context manager
    async def __aenter__(self):
        # report a message
        print('>entering the context manager')
        # block for a moment
        await asyncio.sleep(0.5)

    # exit the async context manager
    async def __aexit__(self, exc_type, exc, tb):
        # report a message
        print('>exiting the context manager')
        # block for a moment
        await asyncio.sleep(0.5)
```

```
...
# use an asynchronous context manager
async with AsyncContextManager() as manager:
    # ...
```

```python
# example of an asynchronous context manager via async with
import asyncio
import time

import asyncio

# define an asynchronous context manager
class AsyncContextManager:
    # enter the async context manager
    async def __aenter__(self):
        # report a message
        print(f'{time.ctime()} >entering the context manager')
        # block for a moment
        await asyncio.sleep(0.5)

    # exit the async context manager
    async def __aexit__(self, exc_type, exc, tb):
        # report a message
        print(f'{time.ctime()} >exiting the context manager')
        # block for a moment
        await asyncio.sleep(0.5)

# define a simple coroutine
async def custom_coroutine():
    # create and use the asynchronous context manager
    async with AsyncContextManager() as manager:
        # report the result
        print(f'{time.ctime()} within the manager')

# start the asyncio program
asyncio.run(custom_coroutine())
```

```
Tue Jul 11 14:31:39 2023 >entering the context manager
Tue Jul 11 14:31:39 2023 within the manager
Tue Jul 11 14:31:39 2023 >exiting the context manager
```

Non-Blocking Streams

- A major benefit of asyncio is the ability to use non-blocking streams.
-

> *Streams are high-level async/await-ready primitives to work with network connections. Streams allow sending and receiving data without using callbacks or low-level protocols and transports.*
>
> — **ASYNCIO STREAMS**

Stream implemented

- HTTP or HTTPS for interacting with web servers
- SMTP for interacting with email servers
- FTP for interacting with file servers.

Checking Website Status

- We can query the HTTP status of websites using asyncio by opening a stream and writing and reading HTTP requests and responses.
- We can then use asyncio to query the status of many websites concurrently, and even report the results dynamically.

How to Check HTTP Status with Asyncio

- Open a connection
- Write a request
- Read a response
- Close the connection

Open HTTP Connection

This can be used to open an HTTP connection on port 80.

For example:

```
...
# open a socket connection
reader, writer = await asyncio.open_connection('www.google.com', 80)
```

We can also open an SSL connection using the **ssl=True** argument. This can be used to open an HTTPS connection on port 443.

For example:

```
...
# open a socket connection
reader, writer = await asyncio.open_connection('www.google.com', 443)
```

Write HTTP Request

- Once open, we can write a query to the StreamWriter to make an HTTP request.

For example, an HTTP version 1.1 request is in plain text. We can request the file path '/', which may look as follows:

```
GET / HTTP/1.1
Host: www.google.com
```

Importantly, there must be a carriage return and a line feed (\r\n) at the end of each line, and an empty line at the end.

As Python strings this may look as follows:

```
'GET / HTTP/1.1\r\n'
'Host: www.google.com\r\n'
'\r\n'
```

Write HTTP Request

This string must be encoded as bytes before being written to the **StreamWriter**.

This can be achieved using the **encode()** method on the string itself.

The default '**utf-8**' encoding may be sufficient.

For example:

```
...
# encode string as bytes
byte_data = string.encode()
```

The bytes can then be written to the socket via the **StreamWriter** via the **write()** method.

For example:

```
...
# write query to socket
writer.write(byte_data)
```

After writing the request, it is a good idea to wait for the byte data to be sent and for the socket to be ready.

This can be achieved by the **drain()** method.

This is a coroutine that must be awaited.

For example:

```
...
# wait for the socket to be ready.
await writer.drain()
```

Read HTTP Response

- Once the HTTP request has been made, we can read the response.
- This can be achieved via the **StreamReader** for the socket.
- The response can be read using the **read**() method which will read a chunk of bytes, or the **readline()** method which will read one line of bytes.

```
...
# read one line of response
line_bytes = await reader.readline()
```

```
...
# decode bytes into a string
line_data = line_bytes.decode()
```

Close HTTP Connection

- We can close the socket connection by closing the StreamWriter.

```
...
# close the connection
writer.close()
```

Checking HTTP Status Sequentially

```python
# check the status of many webpages
import asyncio
import time
from urllib.parse import urlsplit

# get the HTTP/S status of a webpage
async def get_status(url):
    # split the url into components
    url_parsed = urlsplit(url)
    print(f'{time.ctime()} fetch {url}')
    # open the connection
    if url_parsed.scheme == 'https':
        reader, writer = await asyncio.open_connection(url_parsed.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(url_parsed.hostname, 80)
    # send GET request
    query = f'GET {url_parsed.path} HTTP/1.1\r\nHost: {url_parsed.hostname}\r\n\r\n'
    # write query to socket
    writer.write(query.encode())
    # wait for the bytes to be written to the socket
    await writer.drain()
    # read the single line response
    response = await reader.readline()
    # close the connection
    writer.close()
    # decode and strip white space
    status = response.decode().strip()
    print(f'{time.ctime()} done {url}')
    # return the response
    return status

# main coroutine
async def main():
    # list of top 10 websites to check
    sites = ['https://www.google.com/',
        'https://www.youtube.com/',
        'https://www.facebook.com/',
        'https://twitter.com/',
        'https://www.instagram.com/',
        'https://www.baidu.com/',
        'https://www.wikipedia.org/',
        'https://yandex.ru/',
        'https://yahoo.com/',
        'https://www.whatsapp.com/'
        ]
    # check the status of all websites
    for url in sites:
        # get the status for the url
        status = await get_status(url)
        # report the url and its status
        print(f'{time.ctime()} {url:30}:\t{status}')

# run the asyncio program
asyncio.run(main())
```

```
Tue Jul 11 14:53:15 2023 fetch https://www.google.com/
Tue Jul 11 14:53:15 2023 done https://www.google.com/
Tue Jul 11 14:53:15 2023 https://www.google.com/       :       HTTP/1.1 200 OK
Tue Jul 11 14:53:15 2023 fetch https://www.youtube.com/
Tue Jul 11 14:53:16 2023 done https://www.youtube.com/
Tue Jul 11 14:53:16 2023 https://www.youtube.com/      :       HTTP/1.1 200 OK
Tue Jul 11 14:53:16 2023 fetch https://www.facebook.com/
Tue Jul 11 14:53:16 2023 done https://www.facebook.com/
Tue Jul 11 14:53:16 2023 https://www.facebook.com/     :       HTTP/1.1 302 Found
Tue Jul 11 14:53:16 2023 fetch https://twitter.com/
Tue Jul 11 14:53:17 2023 done https://twitter.com/
Tue Jul 11 14:53:17 2023 https://twitter.com/          :       HTTP/1.1 302 Found
Tue Jul 11 14:53:17 2023 fetch https://www.instagram.com/
Tue Jul 11 14:53:17 2023 done https://www.instagram.com/
Tue Jul 11 14:53:17 2023 https://www.instagram.com/    :       HTTP/1.1 302 Found
Tue Jul 11 14:53:17 2023 fetch https://www.baidu.com/
Tue Jul 11 14:53:17 2023 done https://www.baidu.com/
Tue Jul 11 14:53:17 2023 https://www.baidu.com/        :       HTTP/1.1 200 OK
Tue Jul 11 14:53:17 2023 fetch https://www.wikipedia.org/
Tue Jul 11 14:53:18 2023 done https://www.wikipedia.org/
Tue Jul 11 14:53:18 2023 https://www.wikipedia.org/    :       HTTP/1.1 200 OK
Tue Jul 11 14:53:18 2023 fetch https://yandex.ru/
Tue Jul 11 14:53:18 2023 done https://yandex.ru/
Tue Jul 11 14:53:18 2023 https://yandex.ru/            :       HTTP/1.1 302 Moved temporarily
Tue Jul 11 14:53:18 2023 fetch https://yahoo.com/
Tue Jul 11 14:53:19 2023 done https://yahoo.com/
Tue Jul 11 14:53:19 2023 https://yahoo.com/            :       HTTP/1.1 301 Moved Permanently
Tue Jul 11 14:53:19 2023 fetch https://www.whatsapp.com/
Tue Jul 11 14:53:20 2023 done https://www.whatsapp.com/
Tue Jul 11 14:53:20 2023 https://www.whatsapp.com/     :       HTTP/1.1 302 Found
```

Checking Website Status Concurrently

```python
# check the status of many webpages
import asyncio
import time
from urllib.parse import urlsplit

# get the HTTP/S status of a webpage
async def get_status(url):
    # split the url into components
    url_parsed = urlsplit(url)
    print(f'{time.ctime()} fetch {url}')
    # open the connection
    if url_parsed.scheme == 'https':
        reader, writer = await asyncio.open_connection(url_parsed.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(url_parsed.hostname, 80)
    # send GET request
    query = f'GET {url_parsed.path} HTTP/1.1\r\nHost: {url_parsed.hostname}\r\n\r\n'
    # write query to socket
    writer.write(query.encode())
    # wait for the bytes to be written to the socket
    await writer.drain()
    # read the single line response
    response = await reader.readline()
    # close the connection
    writer.close()
    # decode and strip white space
    status = response.decode().strip()
    print(f'{time.ctime()} done {url}')
    # return the response
    return status

# main coroutine
async def main():
    # list of top 10 websites to check
    sites = ['https://www.google.com/',
        'https://www.youtube.com/',
        'https://www.facebook.com/',
        'https://twitter.com/',
        'https://www.instagram.com/',
        'https://www.baidu.com/',
        'https://www.wikipedia.org/',
        'https://yandex.ru/',
        'https://yahoo.com/',
        'https://www.whatsapp.com/'
        ]
    # create all coroutine requests
    coros = [get_status(url) for url in sites]
    # execute all coroutines and wait
    results = await asyncio.gather(*coros)
    # process all results
    for url, status in zip(sites, results):
        # report status
        print(f'{time.ctime()} {url:30}:\t{status}')

# run the asyncio program
asyncio.run(main())
```

```
Tue Jul 11 14:54:40 2023 fetch https://www.google.com/
Tue Jul 11 14:54:40 2023 fetch https://www.youtube.com/
Tue Jul 11 14:54:40 2023 fetch https://www.facebook.com/
Tue Jul 11 14:54:40 2023 fetch https://twitter.com/
Tue Jul 11 14:54:40 2023 fetch https://www.instagram.com/
Tue Jul 11 14:54:40 2023 fetch https://www.baidu.com/
Tue Jul 11 14:54:40 2023 fetch https://www.wikipedia.org/
Tue Jul 11 14:54:40 2023 fetch https://yandex.ru/
Tue Jul 11 14:54:40 2023 fetch https://yahoo.com/
Tue Jul 11 14:54:40 2023 fetch https://www.whatsapp.com/
Tue Jul 11 14:54:40 2023 done https://www.wikipedia.org/
Tue Jul 11 14:54:40 2023 done https://www.youtube.com/
Tue Jul 11 14:54:40 2023 done https://twitter.com/
Tue Jul 11 14:54:40 2023 done https://www.whatsapp.com/
Tue Jul 11 14:54:41 2023 done https://www.instagram.com/
Tue Jul 11 14:54:41 2023 done https://www.baidu.com/
Tue Jul 11 14:54:41 2023 done https://www.facebook.com/
Tue Jul 11 14:54:41 2023 done https://www.google.com/
Tue Jul 11 14:54:41 2023 done https://yandex.ru/
Tue Jul 11 14:54:41 2023 done https://yahoo.com/
Tue Jul 11 14:54:41 2023 https://www.google.com/       :       HTTP/1.1 200 OK
Tue Jul 11 14:54:41 2023 https://www.youtube.com/     :       HTTP/1.1 200 OK
Tue Jul 11 14:54:41 2023 https://www.facebook.com/    :       HTTP/1.1 302 Found
Tue Jul 11 14:54:41 2023 https://twitter.com/         :       HTTP/1.1 302 Found
Tue Jul 11 14:54:41 2023 https://www.instagram.com/   :       HTTP/1.1 302 Found
Tue Jul 11 14:54:41 2023 https://www.baidu.com/       :       HTTP/1.1 200 OK
Tue Jul 11 14:54:41 2023 https://www.wikipedia.org/   :       HTTP/1.1 200 OK
Tue Jul 11 14:54:41 2023 https://yandex.ru/           :       HTTP/1.1 302 Moved temporarily
Tue Jul 11 14:54:41 2023 https://yahoo.com/           :       HTTP/1.1 301 Moved Permanently
Tue Jul 11 14:54:41 2023 https://www.whatsapp.com/    :       HTTP/1.1 302 Found
```