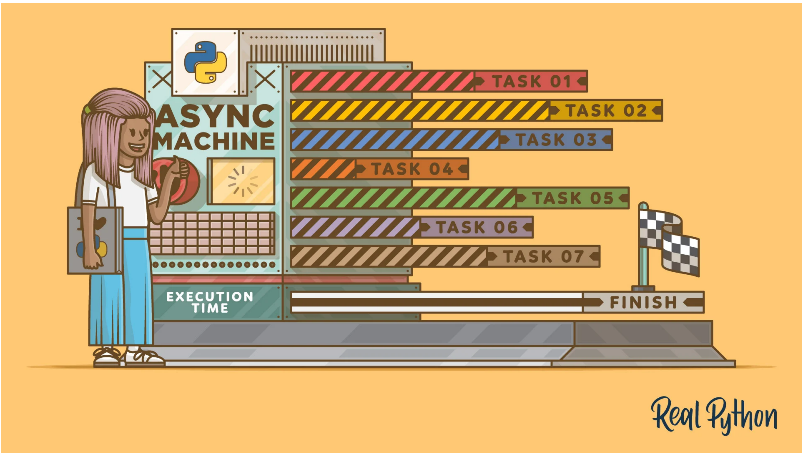
Asynchronous Programming in Python



An Introduction to Asynchronous Programming in Python with AsyncIO

Here's what we'll cover :

- What is Asynchronous Programming
- AsyncIO Library
- Components of Async IO
- Difference between Threading and Asynchronous

What is Asynchronous programming ?

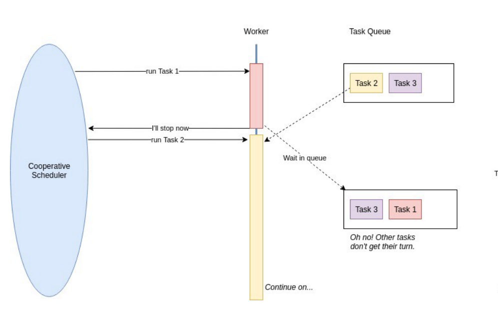
- Asynchronous programming is a programming paradigm that enables better concurrency, that is multiple threads running concurrently
- In Python, asyncio module provides this capability
- Multiple tasks can run concurrently on a single thread, which is scheduled on a single CPU core
- It uses cooperative multitasking means developers can specify in their code when a task voluntarily gives up the CPU so that the event loop can schedule another task.
- Asynchronous routines "pause" while waiting on their ultimate result and let other routines run in the meantime.
- It's not about using multiple cores, it's about using a single core more efficiently

AsyncIO(Asynchronous input-output) :

- AsyncIO is a library which helps to run code concurrently using single thread or event loop
- It is basically using async/await API for asynchronous programming
- asyncio is designed to allow you to structure your code so that when one piece of linear single-threaded code (called a "coroutine") is waiting for something to happen another can take over and use the CPU.

Components of Async IO Programming :

- Event Loop : It manages, distributes the execution and flow of control between various tasks.
- Coroutine : Special type of Python generator which returns the control back to the event loop on encountering the await keyword
- Tasks : Tasks are used to schedule coroutines concurrently
- Futures : It is a low-level awaitable object that is supposed to have a result in the future.



Coroutines :

- Coroutines are mainly generalization forms of subroutines.
- An async function uses the await keyword to denote a coroutine.
- When using the await keyword, coroutines release the flow of control back to the event loop.
- To run a coroutine, we need to schedule it on the event loop
- After scheduling, coroutines are wrapped in Tasks as a Future object.

Subroutines vs. Coroutines :

- Function is also known as a "method" or "procedure" or "sub-process" or "subroutine" denote bits of code that can be called by other code
- Subroutine Calling : In this model of calling each time a function is called execution moves to the start of that function, then continues until it reaches the end of that function
- Coroutine Calling : Suspend its execution and transfer control to other coroutine and can resume again execution from the point it left off.

Caller Function Caller Coroutine

Creatinga A Coroutine :

- The syntax async def introduces either a native coroutine or an asynchronous generator.
- The expressions async with and async for are also valid.

In [2]:
```python
#Syntax Example 1
async def example_coroutine_function(a, b):
    #A function that you introduce with async def is a coroutine
    # Asynchronous code goes here
    ...
    await ...
    return ...      # It may use await, return, or yield, but all of these are optional.

def example_function(a, b):
    # Synchronous code goes here
    ...
    return ...
```

In [3]:
```python
#Syntax Example 2
async def g():
    # Pause here and come back to g() when f() is ready
    r = await f() #To call a coroutine function, you must await it to get its results.
    return r
```

In [4]:
```python
import asyncio
import random

async def myCoroutine():
    process_time = random.randint(2,5)
    await asyncio.sleep(process_time)
    print('This Task Has Been Completed')

myCoroutine()
```

Out[4]:
```
<coroutine object myCoroutine at 0x000002D148CC7340>
```

Note : Object Returned By A Coroutine Function is coroutine object

In [2]:
```python
#Simple Example Demonstrating Use Of "Async / Await" Keywords
import asyncio
from datetime import datetime
import time
#Using these four functions for further demonstration
async def addition(a,b):
    #await sleep() function of asyncio module that waits for a specified number of seconds
    await asyncio.sleep(3)
    print("Addition Result       : ", a + b)

async def multiplication(a,b):
    await asyncio.sleep(1)
    print("Multiplication Result : ", a * b)

async def division(a,b):
    await asyncio.sleep(5)
    print("Division Result       : ", a / b)

async def subtraction(a,b):
    await asyncio.sleep(7)
    print("Subtraction Result    : ", a - b)
```

In [3]:
```python
#Running Asynchrous Without Actuall Implementation
async def main():

    await division(10,20)

    await subtraction(10,20)

    await addition(10,20)

    await multiplication(10,20)


print("Start Time : ", datetime.now())
start = time.time()

#asyncio.run(coroutine, debug=False) - This method takes as input a coroutine and runs it
await main()

print("\nEnd   Time : ", datetime.now())
print("\nTotal Time Taken : {} Seconds".format(time.time() - start))
```

```
Start Time :  2022-09-21 10:52:25.895606
Division Result       :  0.5
Subtraction Result    :  -10
Addition Result       :  30
Multiplication Result :  200

End   Time :  2022-09-21 10:52:41.947935

Total Time Taken : 16.052329063415527 Seconds
```

Caution : There is a slight difference on how Jupyter uses the loop compared to IPython.

Tasks :

- Task is a subclass of futures and it is used to run coroutines concurrently within an event loop.
- Tasks are used to schedule coroutines concurrently.
- When submitting a coroutine to an event loop for processing, we can get a Task object, which provides a way to control the coroutine's behavior from outside the event loop.
- We can create a task using create_task (an inbuilt function of asyncio library)

Create Tasks for Parallel Execution using "create_task ( )" Method

`create_task(coroutine, name=None) - This method takes as input coroutine and returns Task instance eligible to execute in parallel.`

`It also accepts name parameters which can be used to assign a name to a task.`

In [5]:
```python
async def main():

    div_task = asyncio.create_task(division(10,20))

    subtract_task = asyncio.create_task(subtraction(10,20))

    add_task = asyncio.create_task(addition(10,20))

    mul_task = asyncio.create_task(multiplication(10,20))

    await div_task
    await subtract_task
    await add_task
    await mul_task

print("Start Time : ", datetime.now(), "\n")
start = time.time()

await main()

print("\nEnd   Time : ", datetime.now())
print("\nTotal Time Taken : {} Seconds".format(time.time() - start))
```

```
Start Time :  2022-09-21 10:57:06.802844

Multiplication Result :  200
Addition Result       :  30
Division Result       :  0.5
Subtraction Result    :  -10

End   Time :  2022-09-21 10:57:13.810928

Total Time Taken : 7.01826810836792 Seconds
```

Execute Multiple Coroutines using "gather ( )" Method

`gather(awaitables, return_exceptions=False) - This function accepts a list of awaitables as input and returns their results once all awaitables have completed running`

In [6]:
```python
#Instead print using return in previous functions
async def addition(a,b):
    await asyncio.sleep(3)
    return "Addition", a + b

async def multiplication(a,b):
    await asyncio.sleep(1)
    return "Multiplication", a * b

async def division(a,b):
    await asyncio.sleep(5)
    return "Division", a / b

async def subtraction(a,b):
    await asyncio.sleep(7)
    return "Subtract", a - b
```

In [7]:
```python
async def main():

    corrs_result  = await asyncio.gather(division(10,20),
                          subtraction(10,20),
                          addition(10,20),
                          multiplication(10,20))

    for task, result in corrs_result:
        print("{} : {}".format(task, result))


print("Start Time : ", datetime.now(), "\n")
start = time.time()

await main()

print("\nEnd   Time : ", datetime.now())
print("\nTotal Time Taken : {} Seconds".format(time.time() - start))
```

```
Start Time :  2022-09-21 10:58:31.622804

Division : 0.5
Subtract : -10
Addition : 30
Multiplication : 200

End   Time :  2022-09-21 10:58:38.638756

Total Time Taken : 7.015951633453369 Seconds
```

Retrieve Current Task and All Tasks

`current_task() - This method returns Task instance for the task in which it's called.`

`all_tasks() - This method returns list of tasks which are not finished yet.`

In [11]:
```python
#Getting Current Task
async def addition(a,b):
    curr_task = asyncio.current_task() ## Retrieve current task.
    print("{} Started.".format(curr_task.get_name()))
    await asyncio.sleep(3)
    return a + b

async def multiplication(a,b):
    curr_task = asyncio.current_task()
    print("{} Started.".format(curr_task.get_name()))
    await asyncio.sleep(1)
    return a * b

async def division(a,b):
    curr_task = asyncio.current_task()
    print("{} Started.".format(curr_task.get_name()))
    await asyncio.sleep(5)
    return a / b

async def subtraction(a,b):
    curr_task = asyncio.current_task()
    print("{} Started.".format(curr_task.get_name()))
    await asyncio.sleep(7)
    return a - b
```

In [8]:
```python
#Getting ALl Tasks
async def main():

    div_task = asyncio.create_task(division(10,20), name="Division")

    subtract_task = asyncio.create_task(subtraction(10,20), name="Subtraction")

    add_task = asyncio.create_task(addition(10,20), name="Addition")

    mul_task = asyncio.create_task(multiplication(10,20), name="Multiplication")
    #The Task-13 points to the main coroutine.
    total_tasks = asyncio.all_tasks()
    print("Total Tasks ({}) : {}\n".format(len(total_tasks), [task.get_name() for task in total_tasks]))

    await div_task
    await subtract_task
    await add_task
    await mul_task
    #1 task will be pending which is the main coroutine.
    total_tasks = asyncio.all_tasks()
    print("\nTotal Tasks ({}) : {}\n".format(len(total_tasks), [task.get_name() for task in total_tasks]))

    print("\nAddition Result       : ", add_task.result())
    print("Multiplication Result : ", mul_task.result())
    print("Division Result       : ", div_task.result())
    print("Subtraction Result    : ", subtract_task.result())

print("Start Time : ", datetime.now(), "\n")
start = time.time()

await main()

print("\nEnd   Time : ", datetime.now())
print("\nTotal Time Taken : {} Seconds".format(time.time() - start))
```

```
Start Time :  2022-09-21 10:59:45.453277

Total Tasks (5) : ['Addition', 'Multiplication', 'Task-13', 'Division', 'Subtraction']


Total Tasks (1) : ['Task-13']


Addition Result       :  ('Addition', 30)
Multiplication Result :  ('Multiplication', 200)
Division Result       :  ('Division', 0.5)
Subtraction Result    :  ('Subtract', -10)

End   Time :  2022-09-21 10:59:52.474235

Total Time Taken : 7.020957946777344 Seconds
```

Event Loops :

- It as the central executor in asyncio,This mechanism runs coroutines until they complete.
- You can imagine it as while(True) loop that monitors coroutine, taking feedback on what's idle, and looking around for things that can be executed in the meantime.
- It can wake up an idle coroutine when whatever that coroutine is waiting on becomes available.
- Only one event loop can run at a time in Python.
- We can create a event loop using get_event_loop (an inbuilt function of asyncio library)

In [12]:
```python
async def main():
    div_task = asyncio.create_task(division(10,20))
    subtract_task = asyncio.create_task(subtraction(10,20))
    add_task = asyncio.create_task(addition(10,20))
    mul_task = asyncio.create_task(multiplication(10,20))
    #make a task/coroutine wait for other task/tasks to complete using wait
    await asyncio.wait([div_task, subtract_task,add_task,mul_task])

#to run the above function we'll use Event Loops these are low level functions to run async functions
loop = asyncio.get_event_loop()
await main()
#loop.run_until_complete(main())
# loop.close()

# We can also use High Level functions Like: asyncio.run(function_asyc())
```

```
Task-29 Started.
Task-30 Started.
Task-31 Started.
Task-32 Started.
```

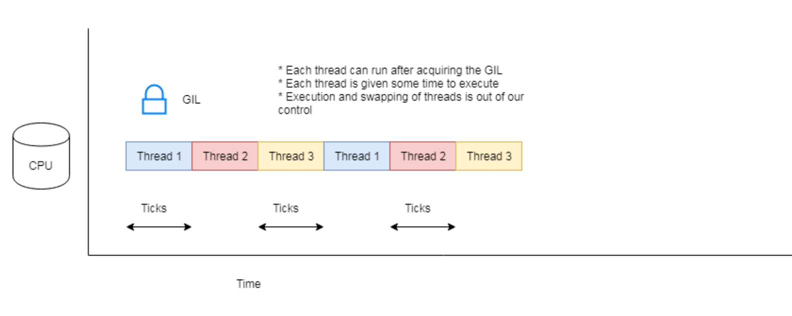
Futures in Asyncio :

- It is a low-level awaitable object that is supposed to have a result in the future.
- When a future object is awaited it means that the coroutine will wait until the Future is resolved in some other place.
- This API exists to enable callback-based code to be used with async/await

In [13]:
```python
import asyncio
from asyncio import Future

#bar coroutine will mark the future as done
async def bar(future):
    print("bar will sleep for 2 seconds")
    await asyncio.sleep(2)
    print("bar resolving the future")
    future.set_result("future is resolved") #Mark the Future as done and set its result.

#foo coroutine will await the future till it is not marked as done
async def foo(future):
    print("foo will await the future")
    future_result = await future
    print("Future result: " + future_result)
    print("Is future is done:", future.done()) #done()- Return True if the Future is done.
    return future.result() #result()- Return the result of the Future.

async def main():
    future = Future()
    #future object is passed to both foo and bar coroutines
    results = await asyncio.gather(foo(future), bar(future))
    print(results) #2nd element is None as bar is not returning anything

if __name__ == "__main__":
    await main() #run() introduced in python >=3.7
    print("main exiting")
```

```
foo will await the future
bar will sleep for 2 seconds
bar resolving the future
Future result: future is resolved
Is future is done: True
['future is resolved', None]
main exiting
```

How is asyncio different from threading ?



`Both threading and asyncio are suited for IO bound code.Major difference is that,`

- In threading the execution and swapping of threads is not something that we can control [Preemptive]
- Where as Asynchronous gives more control to the developer over the execution of tasks [Cooperative]
- In Threading Since threads uses same memory, sharing of objects between them is little tricky
- Where as in Asynchronous it is easier to manage objects between tasks and not worry about race conditions
- In Threading It's hard to understand how threads run and their order and so is difficult to spot any bugs.
- Where as in Asynchronous Code is more readable and light weight.

Implementation :

`For this benchmark, we will be fetching data from a sample URL  with different frequencies, like once, ten times, 50 times, 100 times, 500 times, respectively.`

In [15]:
```python
#Ignoring Warnings
import warnings
warnings.filterwarnings("ignore")
```

In [16]:
```python
#lists for benchmarking
freq = [1,10,50,100,500]
```

Code of Multithreading:

In [17]:
```python
#Code of Multithreading:
import requests
import time
from concurrent.futures import ThreadPoolExecutor

thread_time = []

def fetch_url_data(pg_url):
    try:
        resp = requests.get(pg_url)
    except Exception as e:
        print(f"Error occured during fetch data from url{pg_url}")
    else:
        return resp.content


def get_all_url_data(url_list):
    with ThreadPoolExecutor() as executor:
        resp = executor.map(fetch_url_data, url_list)
    return resp


if __name__=='__main__':
    url = "https://www.udemy.com"
    for ntimes in freq:
        start_time = time.time()
        responses = get_all_url_data([url] * ntimes)
        process_time = time.time() - start_time
        thread_time.append(process_time)
        print(f'Fetch total {ntimes} urls and process takes {process_time} seconds')
```

```
Fetch total 1 urls and process takes 0.9331731796264648 seconds
Fetch total 10 urls and process takes 1.7237188816070557 seconds
Fetch total 50 urls and process takes 5.962061405181885 seconds
Fetch total 100 urls and process takes 10.16848349571228 seconds
Fetch total 500 urls and process takes 47.21777296066284 seconds
```

Async IO Programming Example:

In [32]:
```python
#aiohttp - Asynchronous HTTP Client/Server for asyncio and Python.
#pip install aiohttp
```

In [18]:
```python
import asyncio
import time
from aiohttp import ClientSession, ClientResponseError

asyinco_time = []

async def fetch_url_data(session, url):
    try:
        async with session.get(url, timeout=60) as response:
            resp = await response.read()
    except Exception as e:
        print(e)
    else:
        return resp
    return


async def fetch_async(loop, r):
    url = "https://www.udemy.com"
    tasks = []
    async with ClientSession() as session:
        for i in range(r):
            task = asyncio.ensure_future(fetch_url_data(session, url))
            tasks.append(task)
        responses = await asyncio.gather(*tasks)
    return responses


if __name__ == '__main__':
    for ntimes in freq:
        start_time = time.time()
        loop = asyncio.get_event_loop()
        future = asyncio.ensure_future(fetch_async(loop, ntimes))
        await future #will run until it finish or get any error
        responses = future.result()
        process_time = time.time() - start_time
        asyinco_time.append(process_time)
        print(f'Fetch total {ntimes} urls and process takes {process_time} seconds')
```

```
Fetch total 1 urls and process takes 0.9526138305664062 seconds
Fetch total 10 urls and process takes 1.6786060333251953 seconds
Fetch total 50 urls and process takes 2.657850980758667 seconds
Fetch total 100 urls and process takes 2.1606643199920654 seconds
Fetch total 500 urls and process takes 7.41143274307251 seconds
```

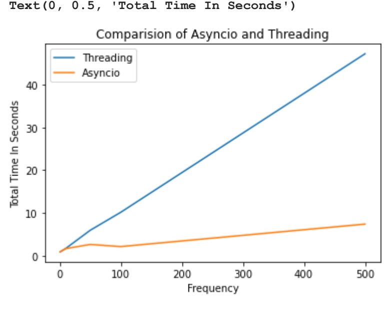
Benchmarking:

In [19]:
```python
import pandas as pd
comparision_list = pd.DataFrame(
    {'Frequency': freq,
     'Threading': thread_time,
     'Asyncio': asyinco_time
    })

print(comparision_list)
```

```
   Frequency  Threading   Asyncio
0          1   0.933173  0.952614
1         10   1.723719  1.678606
2         50   5.962061  2.657851
3        100  10.168483  2.160664
4        500  47.217773  7.411433
```

In [20]:
```python
plt = comparision_list.plot.line(x="Frequency", title="Comparision of Asyncio and Threading")
plt.set_ylabel("Total Time In Seconds")
```

Out[20]:
```
Text(0, 0.5, 'Total Time In Seconds')
```



Conclusion :

- As we saw, Async IO showed better performance with the efficient use of concurrency than multi-threading.
- Async IO can be beneficial in applications that can exploit concurrency.
- Though, based on what kind of applications we are dealing with, it is very sensible to choose Async IO over other implementations.

FAQ's on Asynchronous Programming :

- What is Asynchronous Programming?.
- Explain what async IO is in Python ?.
- What are the advantages of using async IO over multithreading or multiprocessing?
- How do you implement asynchronous event loops in Python ?
- What does await do in Python ?
- What's the difference between coroutines and generators in Python ?
- Can you give me some examples where you would use AsyncIO in your application code ?
- Can you explain what a future object is in Python ?
- What do you understand about context switching ?
- What is GIL ? Do you think it poses a problem for multi-threaded apps written in Python ?
- What is the maximum number of threads per process in Python ?