Chapter 16

Programming

16.1 Python Programming

16.1.1 Asynchronous Programming

When working with applications that involve heavy I/O operations, such as downloading stock market data, running database queries, these operations may be time intensive and degrade the performance of the application. If we run these operations sequentially, then the performance issues grow linearly in the number of such operations and the application does not scale. If we can exploit concurrency to make these requests together for simultaneous waiting, our application would be much less idle and benefit from less latency. In general, an application's work can be divided into I/O and CPU bound (bound means limiting factor) work. The I/O operations involve a computer's input and output devices, such as keyboard, hard drive and network cards. CPU operations involving the central processing unit, and consists of work such as mathematical computation. Concurrency refers to allowing more than a single task to be handled at the same time. One of the ways this is achieved in Python is through the *asyncio* module, which stands for asynchronous I/O. Asynchronous programming refers to the act of running tasks in the background, separate from the main application. The *asyncio* library utilizes the concurrency model known as a single-threaded event loop. We will explore what these mean later. Additionally, *asyncio* provides utility functions that allow us to work with multiple threads and multiple processes, giving us better control over these workflows in neat abstraction.

Concurrency means that at least two tasks are happening at the same time. This can be rather vague and needs to be clarified. Concurrency does not in fact imply that something is running in parallel. When we say that parallelism exists, it requires that the tasks are both concurrent and executing at the same time. We may be concurrently letting the cake cool in the oven while preparing the table - these two events can be conducted concurrently. When you hire someone to cook the fish while you prepare the jus, the events occur in parallel. Concurrency can be obtained on a single core CPU via task-switching using a mechanism known as preemptive multitasking. Parallelism can only happen on a machine with at least two cores. Parallelism implies concurrency but not the other way around. With multitasking, one may have multiple tasks happening concurrently - except only one of them is executing at one time. Multitasking falls under two types - preemptive multitasking and cooperative multitasking. In the former, the OS (operating system) decides how and when to switch between different tasks via time slicing. In the latter, the running process decides to give up CPU time via explicitly earmarked regions in the code. *asyncio* in particular employs the cooperating multitasking model to achieve concurrency.

As mentioned, it is single threaded (and therefore also single proccess) and runs a concurrent-but-not-parallel model. Of course, we can combine parallelism with *asyncio*, and we will see how to do that later. Cooperating multitasking has some benefits. One of that is that it is less resource intensive. When the OS switches between running threads or processes, a context switch is said to be involved, and the OS needs to save the state of the running process/thread in order to continue later. This is expensive. By explicitly earmarking the areas for giving up CPU control, the application also has efficiency gains, assuming it is done properly.

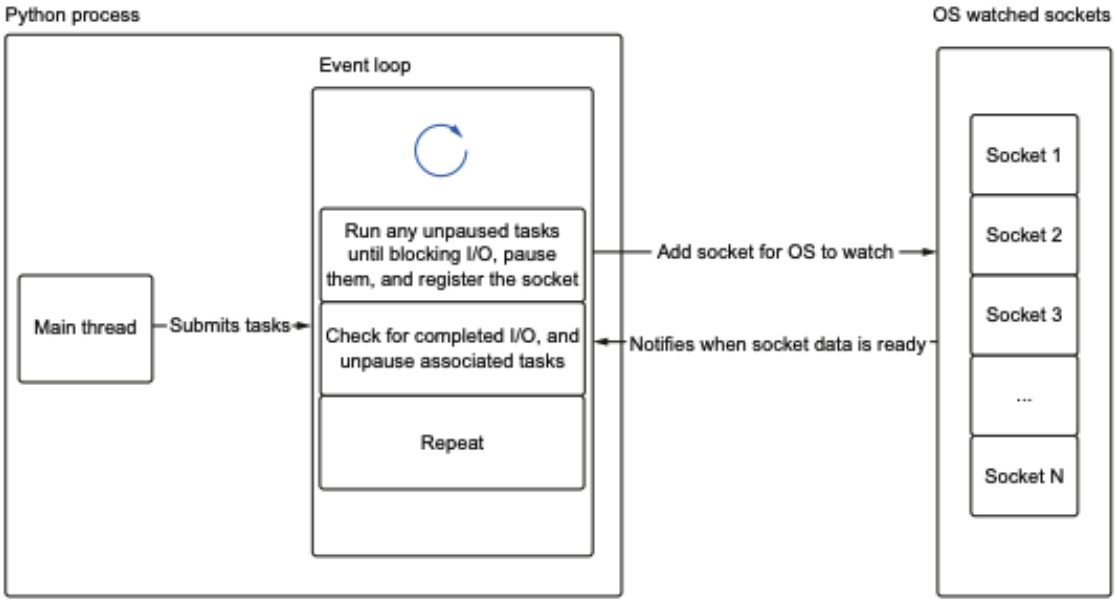
So this begs the question of the difference between a process and a thread. A process is an application that has its own memory space that other applications cannot access. Multiple processes can run on a single machine at the same time if there are multiple cores, otherwise they run concurrently via time slicing. The algorithms that determines when the OS triggers task-switching is OS-dependent, and will hence not be discussed in this section, which is intended to be mostly OS-independent. A thread on the other hand, does not have its own memory space. Instead, they share the memory with the same process that created it. A process is always minimally associated with a thread known as the main thread. The other threads created from the main thread is known as worker threads. Threads can run alongside one another on a multi-core CPU, and the OS can also switch between threads via time slicing. One can think of threads as lightweight processes, with a different memory profile. The entry into any Python application begins with a process and its main thread.

However it turns out that multi-threading for achieving concurency in Python is somewhat nuanced due to a feature known as the GIL (global interpretor lock). Any high-level language is converted into machine language by compilers or interpretors, or a mix of both, and has lower-level representations in bytecodes, assembly language and so on, which is a further abstraction on top of binary code, which specifies computer instruction by ones and zeros. We are purposefully abstracted away from this gritty detail, and so we will not go into its details. Briefly, the GIL prevents a Python process from executing more than one line of Python bytecode instruction at any time. While this may seem restrictive, the feature exists due to the memory management in CPython, the reference implementation of Python. Memory in CPython is managed by a mechanism known as reference counting, which keeps track the number of variables referencing the object. When a variable pointer is assigned to the object, the reference count is incremented - and then decremented when the variable no longer needs it. When the reference count hits zero, it is put in garbage collection to free up memory. CPython however, is not thread safe, which means that if multiple threads share a varaible and attempt to modify it, the result is non-deterministic, a state known as race condition. While threads can run concurrently on different cores, the interpretor lock ensures that only one of them runs Python code at a time to prevent race conditions. The other threads idle. While this can paint a doom picture for Python as a programming language, there are workarounds. Firstly, the GIL is released when the task occurs outside the Python runtime. An example is when I/O operations happen. I/O operations make lower level system calls in the OS that is outside of the Python runtime, and the GIL is released since we are not working directly with Python objects. The GIL is reacquired when the data is received and translated back into a Python object instance. In other languages such as Java, C, C++, the GIL is non-existent and threads running on different cores are parallel. *asyncio* exploits this fact that I/O releases GIL to give us concurrency with a single-thread using objects known as coroutines. A coroutine itself can be thought of as an even lighter-weight version of a thread. Of course, single-threads belong to the same process, so coroutines share the same memory space. To understand this single-threaded event loop model, we introduce sockets. A socket is a low-level abstraction for sending and receiving data over between applications over a network.

Sockets are by default blocking in nature, meaning that when we fire off a request for a data packet, it stops/blocks until we receive the data. However, they can be made to operate in a non-blocking mode, which means we fire and forget, until the OS notifies us that we have an inbox. These notification systems is interfaced by the OS and is OS-dependent. *asyncio* abstracts away these details to work with the different notification systems, allowing the developers to be mostly ignorant about the underlying mechanisms of data transfer. In *asyncio*, we we hit an I/O operation, we request the OS to make the I/O request, and then register in the notification system to remind us of any updates. Meanwhile, we are free to return to the Python runtime and execute other work. When the notification is received, Python receives the object and continues the code execution. The tasks that are waiting for I/O are tracked inside the Python runtime using a construct known as the event loop.

In *asyncio*, the event loop keeps a queue of tasks, which are wrappers around coroutines. When an event loop is instantiated, an empty queue of tasks is created. Each iteration of the event loop checks for tasks to run and runs them until an I/O operation is hit, which *asyncio* then makes the necessary system calls for the operation and for being notified of their progress. It then goes on to look for other tasks to execute. Each iteration also involves checking for any tasks that have already completed.

Figure 16.1: An example of a thread submitting tasks to the event loop.
Source: Figure 1.9 in Fowler [7]



Creating a coroutine is fairly straightforward, using the *async def* keyword and *await* keyword at the I/O code points. We will demonstrate this later in some substantial examples. When a coroutine is called in the manner *result = coroutine(args)*, it is not executed immediately, but a coroutine object is assigned to the variable *result*. To run a coroutine, it needs to be explicitly put on the event loop, and an iteration of the event loops need to be triggered. The different ways of doing so would be of primary interest going forward in this section.

One of the convenience functions is *asyncio.run*, we takes in a coroutine and runs it.

```
1 import asyncio
2
3 async def main():
```

```
    await asyncio.sleep(1)
    print("slept one(s)")
    return True

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

Here, *asyncio* creates a new event-loop, and runs the *main* coroutine until completion. *asyncio.sleep* is also coroutine, which we will use as a stand-in function to simulate an I/O operation such as a database request. After completion, some cleanup is done and the event loop is closed. The *asyncio.run* function is intended to be the main entry into an asynchronous application. Other coroutines should be launched inside the coroutine that is called inside this *run* function, which is in our case, *main*. The *await* keyword is followed by an object of the *awaitable* type, which is satisfied by coroutines. The coroutine itself is paused on the await sleep instruction, and is then woken up later to continue execution. While no concurrent work was done here, these are the basic building blocks of an asynchronous application.

In fact, suppose we write this:

```
import asyncio

async def main():
    await asyncio.sleep(1)
    print("slept one(s)")
    await asyncio.sleep(1)
    print("slept one(s)")
    return True

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

it will actually take roughly two seconds. As mentioned, the coroutine itself pauses until the line completes, which means we are sleeping one second twice, one after the other, instead of together. But we really want concurrency. In order to achieve this, we need to introduce *task*s, which are wrappers around coroutines to schedule the contained coroutine to run on the event loop as early as possible. This scheduling and execution is non-blocking, while the *await* keyword is blocking. We can do this with *asyncio.create_task* function. When a task is created, it is scheduled to run on the next iteration of the event loop. An iteration of the event loop can be triggered by the *await* statement. Usually, when a task is created, we would await it at some point. An interesting aside is made. If we write this:

```
async def verbose_sleep(s):
    await asyncio.sleep(s)
    print("slept for", s)

async def main():
    task1 = asyncio.create_task(verbose_sleep(5))
    task2 = asyncio.create_task(verbose_sleep(6))
    return True

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

the application would actually terminate immediately. Here the task objects are created but the event loop is not triggered and the function terminates. If instead we go with

```
async def verbose_sleep(s):
    await asyncio.sleep(s)
```

```
    print("slept for", s)

async def main():
    task1 = asyncio.create_task(verbose_sleep(5))
    task2 = asyncio.create_task(verbose_sleep(3))
    await task2
    return True

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

then we we will see 'slept for 3' - both tasks are scheduled and run on the event loop, but the application terminates after the second task is completed. There are still more seconds to sleep on the first task and it never gets to finish. Instead,

```
async def verbose_sleep(s):
    await asyncio.sleep(s)
    print("slept for", s)

async def main():
    task1 = asyncio.create_task(verbose_sleep(5))
    task2 = asyncio.create_task(verbose_sleep(6))
    await task2
    return True

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

this would see both 'slept for 5' and 'slept for 6'. Even though we never awaited the first task, it runs and completes before the second tasks completes. Of course, if we await both tasks, we will always see both print statements. Tasks are scheduled to run as soon as possible, which generally means when the first *await* statement is encountered after the task has been created. The sleeping here is done concurrently, and in the last example, should terminate after approximtely six seconds and not eleven. Of course, we may do some work without wasting time. Before we demonstrate that, let's create a utility decorator function that acts as a stopwatch timer for our application. We will discuss decorators in other sections -

```
def asyn_timefn(func):
    import time
    from functools import wraps
    @wraps(func)
    async def timediff(*args, **kwargs):
        a = time.time()
        result = await func(*args, **kwargs)
        print(f"@timefn: {func.__name__} took {time.time() - a} seconds")
        return result
    return timediff
```

Now suppose we want to compute the mean of ten million numbers of a uniform random sample.

```
async def verbose_sleep(s):
    await asyncio.sleep(s)
    print("slept for", s)

@asyn_timefn
async def main():
```

```
    task1 = asyncio.create_task(verbose_sleep(5))
    task2 = asyncio.create_task(verbose_sleep(6))
    import random
    import numpy as np
    work = np.mean([random.uniform(0,1) for _ in range(10000000)])
    await task1
    await task2
    return work

if __name__ == "__main__":
    work = asyncio.run(main())
    print(work)

'''
slept for 5
slept for 6
@timefn: main took 8.519409894943237 seconds
0.4998836811668402
'''
```

Interestingly it took quite abit more than six seconds to complete this - it turns out that the mathematical computation was not done concurrently. This is because the tasks were created and scheduled but no event loop iteration was triggered. We can change the order of line 11 and 12, but this would only mean we are doing the computation and task2 in parallel. We want to do the math while waiting for both task1 and task2. Here's the trick:

```
@asyn_timefn
async def main():
    task1 = asyncio.create_task(verbose_sleep(5))
    task2 = asyncio.create_task(verbose_sleep(6))
    import random
    import numpy as np
    await asyncio.sleep(0)
    work = np.mean([random.uniform(0,1) for _ in range(10000000)])
    await task1
    await task2
    return work
'''
slept for 5
slept for 6
@timefn: main took 6.001631021499634 seconds
0.49996590071870267
'''
```

The zero-sleep operation triggers the event loop, fires both sleep requests and does the mathematical work. There is almost no additional time spent on the statistical sampling work. Now we know how to work with tasks and coroutines, and some quirks. Let's discover some more of the *asyncio* workflow and control.

Network connections can be unstable and might hang indefinitely. We want to be able to cancel tasks. *Task* objects come with method *cancel*, and the running tasks raises a *CancelledError* when we await on the task.

```
async def main():
    task1 = asyncio.create_task(verbose_sleep(5))
    assert not task1.done()
```

```
    task1.cancel()
    try:
        await task1
    except asyncio.CancelledError:
        print("task was cancelled")
    return

if __name__ == "__main__":
    asyncio.run(main())

'''
task was cancelled
'''
```

It should be noted that the *CancelledError* is only thrown at await statements, meaning to say if the cancel is submitted while the task is midway through executing plain Python code, the code runs until the next await statement, if any. We might already have some time in mind to cancel the task, which we should then go for *asyncio.wait_for*, taking in a coroutine/task and number of seconds to wait for - after which a *TimeoutException* is thrown:

```
async def main():
    try:
        task1=asyncio.create_task(verbose_sleep(3))
        await asyncio.wait_for(task1,2)
        await task1
    except asyncio.TimeoutError:
        print("task was timed out with status cancel =", task1.cancelled())
    return

if __name__ == "__main__":
    asyncio.run(main())

'''
task was timed out with status cancel = True
'''
```

But suppose we want to let the task continue executing, instead of cancelling it after specified seconds we can *asyncio.shield* the function:

```
async def main():
    try:
        task1=asyncio.create_task(verbose_sleep(3))
        await asyncio.wait_for(asyncio.shield(task1),2)
        await task1
    except asyncio.TimeoutError:
        print("task was timed out with status cancel =", task1.cancelled())
        await task1
    return

if __name__ == "__main__":
    asyncio.run(main())
'''
task was timed out with status cancel = False
slept for 3
'''
```

Ok now let's understand what links tasks and coroutines. We know they are both awaitable, but what is an awaitable? To do that we need to introduce *future*s. Futures are Python objects that contain a single value you expect to get in the future. When you first create one, that value normally doesn't exist, and somewhere downstream, you have it, and you set the value of the future to that object. Futures can be awaited, and are either *done* or *undone*. The following example should be sufficiently clear:

```
async def main():
    future = asyncio.Future()
    print(future.done())
    async def _helper():
        await asyncio.sleep(3)
        future.set_result("DONE")

    task = asyncio.create_task(_helper())
    await asyncio.sleep(0)
    value = await future
    print(future.done(), "with result", value)
    return

if __name__ == "__main__":
    asyncio.run(main())
'''
False
True with result DONE
'''
```

The internal implementation of the *asyncio* API relies heavily on futures. A task can be thought of as combination of coroutine and future. When a task is created, an empty future is created - with the expectation that the future result is set to the output of the coroutine in question.

They are awaitables, which means to say they are object instances of classes that implement the *Awaitable* abstract base class. Abstract base classes are discussed in other sections, but briefly, it means they specify some behavior that parent classes implementing it should satisfy. It determines the contract specification. For instance, when we hire a burger cook, when we call *chef.cook()*, we know we want a burger, even though different cooks can prefer sous vide, grill or whatever. In our case the method to be implemented is the *__await__* method. The inheritance diagram looks something like this:

```
Awaitable
        Coroutine
        Future
            Task
```

When we look at the API documentation for *asyncio*, it is useful to know what is the object instance they require. The children is always an instance of the parent class, so for a function that takes in an awaitable, we really can pass in an instance of coroutine, future or task. A function that takes in instances of coroutine only would not work when a task is passed in, however.

Common mistakes in using the *asyncio* API are discussed. One is trying to use CPU-bound code in tasks or coroutines without multiprocessing. See that the following two listings that approximately the same time.

```
async def cpu_work():
    import numpy as np
    import random
```

```
    sample = [random.uniform(0,1) for _ in range(10000000)]
    sum = 0
    for num in sample:
        sum += num
    return sum

@asyn_timefn
async def main():
    task1 = asyncio.create_task(cpu_work())
    task2 = asyncio.create_task(cpu_work())

    await task1
    await task2
    return

if __name__ == "__main__":
    asyncio.run(main())
'''
@timefn: main took 4.500612020492554 seconds
'''
```

```
def cpu_work():
    import numpy as np
    import random
    sample = [random.uniform(0,1) for _ in range(10000000)]
    sum = 0
    for num in sample:
        sum += num
    return sum

@asyn_timefn
async def main():
    cpu_work()
    cpu_work()
    return
'''
@timefn: main took 4.320184946060181 seconds
'''
```

Actually the second program runs faster, because there is lesser overhead without coroutines, Although the difference is marginal, it definitely does not help the program. Firstly, the CPU work here does not have any await statements. So it would not give up execution for other I/O tasks behind it in the event loop to get a chance to submit the I/O request until it finishes execution. Secondly, because *asyncio* is single threaded, no CPU work occurs in parallel. Furthermore, even if it were multi-threaded, the GIL would prevent any parallelism. What we would want here is multi-processing, which would be discussed later. The second mistake would be to asynchronously call blocking APIs. When we know that we are making an I/O operation through an API, we might be tempted to wrap it in a coroutine. But the APIs themselves may be blocking the event loop itself, so we will not get performance benefits.

```
async def hangukquant():
    import requests
    res=requests.get("https://www.hangukquant.com")
    return

@asyn_timefn
```

```
async def main():
    tasks=[asyncio.create_task(hangukquant()) for _ in range(30)]
    await asyncio.sleep(0)
    [await task for task in tasks]
    return

if __name__ == "__main__":
    asyncio.run(main())
'''
@timefn: main took 6.972156286239624 seconds
'''
```

That the network request is an I/O operation is accurate. However, the internal implementation of the *requests* module block on the *get* method, so no asynchronous work is done here. We can either use multithreading with blocking APIs such as *requests* or use dedicated asyncio-supported libraries such as *aiohttp*, which uses non-blocking sockets. Later, we will examine them.

Previously we used the *asyncio.run* method to handle the event loop for us. We can actually have more fine grained control over the event loop behavior, but we would need to be careful about cleaning up the resoruce and closing the loop. This would work:

```
async def main():
    await verbose_sleep(1)
    return

if __name__ == "__main__":
    loop = asyncio.new_event_loop()
    try:
        loop.run_until_complete(main())
    finally:
        loop.close()
```

We can also replace *new_event_loop* with *get_event_loop*, which gets the running loop if any, or creates a new one. If we do *get_running_loop* instead, we would get an error that there is no currently running event loop. We can also explicitly put non-async functions on the event loop to run on the next iteration with *call_soon*:

```
def print_hello():
    print("hello")

async def main():
    loop = asyncio.get_running_loop()
    loop.call_soon(print_hello)
    await verbose_sleep(1)
    return

if __name__ == "__main__":
    loop = asyncio.new_event_loop()
    try:
        loop.run_until_complete(main())
    finally:
        loop.close()
'''
hello
slept for 1
'''
```

We refer readers to documentation to nuances, if more control over the event loop is desired within the application. Other options, such as debug mode exists, i.e. *asyncio.run(main(), debug=True)*. When working with production quality applications, exception handling is very important. Note that when exception is thrown inside a running task, the task is considered done with the result as exception. No exception is thrown up the call stack, and there is no cleanup. To get the exception thrown to us, we need to *await* the task, otherwise the exception may never be retrieved, particularly if the task is not garbage collected.

We want to have control over the workflow of coroutines, and explore some useful APIs that are *asyncio* compatible. An example is the *aiohttp* library, which uses non blocking sockets to make web requests and returns coroutines for the requests. As mentioned, the *requests* library is blocking and would block the thread it runs in. Before we go into *aiohttp*, let us first understand the role of context managers in resource control. In particular, we are interested in asynchronous context managers - they implement two coroutines *__aenter__* and *__exit__*. Among other things, context managers help to manage resources and perform auxiliary functionalities such as resource clean up. A common example would be:

```
with open("test.txt") as file:
    lines= file.readlines()
```

which is a more Python-ic way of doing this:

```
file = open("test.txt")
try:
    lines= file.readlines()
finally:
    file.close()
```

With asynchronous context managers, the syntax is slightly different. For instance, we will use *async_with* instead of the *with* keyword. We can acquire an asynchronous lock like this:

```
lock = asyncio.Lock()
counter = 0
async def increment():
    global counter
    async with lock:
        counter += 1
```

We will not go into too intimate details with context managers - they will discussed in other sections. *aiohttp* uses async context managers for acquiring HTTP sessions and connections. Sessioning is a networking term for keeping connections open. Without going into the details, when HTTP requests are made over the network, protocols such as the TCP handshake and exchange of other information are expensive. Sessioning keeps the connections open and allow us to recycle them, and this is known as connection pooling, which reduces the resources required and improve the performance of *aiohttp*-based applications. With this non-blocking library, we may use sessioning and coroutines to speed up the same problem in Listing 16.1.1.

```
async def hangukquant(session):
    async with session.get("https://www.hangukquant.com") as res:
        return res.text

@asyn_timefn
async def main():
    import aiohttp
    async with aiohttp.ClientSession() as session:
        tasks=[asyncio.create_task(hangukquant(session)) for _ in range(30)]
```

```
        await asyncio.sleep(0)
        [await task for task in tasks]
    return

if __name__ == "__main__":
    asyncio.run(main())
'''
@timefn: main took 1.417881965637207 seconds
'''
```

Notice that our program runs significantly faster. The *ClientSession* creates a default maximum of hundred connections. This may be altered - we refer readers to *aiohttp* documentation. Previously, we could set timeouts with *asyncio.wait_for* method. Although this works with arbitrary coroutines, and hence also applicable here, *aiohttp* provides out of the box functionalities to do this cleanly. At the session level, we can instantiate the client session with a timeout object to get a timeout limit and replace the relevant lines of code with:

```
    session_timeout = aiohttp.ClientTimeout(total=1,connect=0.3)
    timed_session = aiohttp.ClientSession(timeout=session_timeout)
    async with timed_session:
        ...
```

which sets a time limit on the whole request and also for subroutines such as establishing connections. On the per-request level, we can also specify timeouts by passing a client timeout object to the get request:

```
    async with session.get(
        "https://www.hangukquant.com",
        timeout=aiohttp.ClientTimeout(total=0.5)
    ) as res:
        ...
```

If the requests are not successful by the time specified, we would get an *asyncio.TimeoutError* on await statement. The last code example works, but it is kind of clunky. First, we have to wrap coroutines in task objects, then trigger an iteration of the event loop with a pseudo-sleep method and then await each task. We can do this cleanly using the *asyncio.gather* method, which takes in a sequence of awaitables and runs them concurrently. If any of the awaitables passed in were coroutines, they are automatically wrapped in task objects so that they get run on the event loop concurrently. The gather method itself returns an awaitable, and even though the individual tasks may not complete in deterministic order, the results are returned in the order we pass them in. Refer to the following example:

```
async def hangukquant(session, request_no):
    async with session.get("https://www.hangukquant.com") as res:
        result=res.text
    return request_no

async def main():
    import aiohttp
    async with aiohttp.ClientSession() as session:
        results = await asyncio.gather(*[hangukquant(session,i) for i in range(30)])
        print(results)
    return

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

```
'''
[0, 1, 2, 3, ..., 28, 29]
'''
```

Much cleaner. What about exceptions? *gather* has an optional parameter *return_exceptions*, specifying two different behavior - it defaults to *False*, which causes *gather* to throw an exception when we await it if any of the coroutines throw an exception. It should be noted that in these scenarios, the other coroutines are not actually cancelled - if we handle the exception, they will be allowed to run until completion. Otherwise, they will run until they are cleaned up downstream or to completion, whichever is earlier. If we set *return_exceptions* to *True*, then the *gather* method will not throw any exception. Instead, the return value will be thrown exception inside the coroutine:

```
async def hangukquant(session, request_no):
    async with session.get("https://www.hangukquant.com") as res:
        import random
        if random.uniform(0,1) > 0.5:
            raise Exception("unlucky error")
        result=res.text
    return request_no

@asyn_timefn
async def main():
    import aiohttp
    async with aiohttp.ClientSession() as session:
        results = await asyncio.gather(*[hangukquant(session,i) for i in range(10)],
    return_exceptions=True)
        print(results)
    return

if __name__ == "__main__":
    asyncio.run(main())
'''
[0, Exception('unlucky error'), 2, Exception('unlucky error'), Exception('unlucky error
    '),
    Exception('unlucky error'), Exception('unlucky error'), Exception('unlucky error'),
    8, 9
]
'''
```

Note that if let the exception be thrown at *gather*, then we will only retrieve the first one when we await it. Maybe this is okay, but maybe we do want to specify a different behavior. Perhaps we would like to cancel our tasks if one of the tasks fail, such as when the tasks make API requests to the same server and we receive a 429 code as rate limits get triggered. Spamming the server with the remaining requests might get us blacklisted - so cancelling the remaining tasks would be desirable. If the tasks take disproportionate amount of time to complete, the *gather* method only completes after the longest task is done - we may prefer to work on the results of the requests that have already completed first. *asyncio* provides the *as_completed* method which takes in a list of awaitables and returns an iterator of futures. We can iterate over and await on each of the futures, with the result being the return value of the coroutine on a first-finish basis. Therefore, the ordering of results are non-deterministic. We demonstrate with the following:

```
async def work(num):
    import random
    work_cost = random.uniform(0,4)
```

```
    await asyncio.sleep(work_cost)
    return (num,work_cost)

@asyn_timefn
async def main():
    async_work = [work(i) for i in range(10)]
    for finished in asyncio.as_completed(async_work):
        res = await finished
        print(res)
    return

if __name__ == "__main__":
    asyncio.run(main())

'''
(6, 0.3673681494715675)
(9, 1.276757248018661)
(7, 1.3913958197279603)
(3, 2.3091369741247556)
(5, 2.3978259114992753)
(8, 2.4415577155094477)
(2, 2.5467792505773)
(1, 2.60398276135025)
(0, 3.159090865296541)
(4, 3.824869021632983)
@timefn: main took 3.829425096511841 seconds
'''
```

Since we have access to individual tasks, we also have better control over exception handling. As before, any task exception would be thrown at the await statement. There is an optional parameter *timeout* in the *asyncio.as_completed* function, which specifies the number of seconds to let the group of tasks run. Any task taking longer than the timeout would throw a *asyncio.TimeoutException* when awaited. As we get the results of those that finished earlier right away, any result retrieved may straight away be worked on without the results of the others. While this is an advantage over *gather*, we might want even more control. When timeout occurs, exception is thrown, but there is no cancellation. The tasks still run in the background. Perhaps, we would like to cancel them. *asyncio* offers another method *wait* that gives several options on when we would like to receive our results from the tasks. The method returns two sets - one in which the tasks are completed (either a result is computed or exception is thrown) and the other in which the tasks are still running. The optional *timeout* configuration on this method specifies the time requirements. Additionally, upon timeout, no exception is thrown, the coroutines are not cancelled and we would have to iterate over the pending task set to cancel them. Instead of throwing the exception, on timeout, all the tasks still running are just returned in the pending set for us to do our preferred handling. The options in the *wait* protocol is *ALL_COMPLETED*, *FIRST_EXCEPTION*, *FIRST_COMPLETED*. It should be pretty clear from their variable names what behavior they specify. We refer the details to the documentation, and instead provide an example. To be brief, the *ALL_COMPLETED* option would behave similar to *gather*, and the *FIRST_COMPLETED* option would behave similar to *as_completed*. Of course, if we receive the results of the tasks that completed earlier, we may go ahead and perform work on it without waiting for the other tasks to complete. *FIRST_EXCEPTION* option is interesting - we shall demonstrate a program that cancels the pending tasks on encountering an exception:

```
async def work(num):
```

```
    import random
    work_cost = random.uniform(0,1)
    await asyncio.sleep(work_cost)
    if random.uniform(0,1) < 0.30:
        raise Exception(f"Task {num} had an unlucky failure")
    return (num,work_cost)

@asyn_timefn
async def main():
    async_work = [asyncio.create_task(work(i)) for i in range(10)]
    done, pending = await asyncio.wait(async_work, return_when=asyncio.FIRST_EXCEPTION)
    for item in done:
        if item.exception() is None:
            print(item.result())
        else:
            print(item.exception())
    for item in pending:
        print("cancelling ", async_work.index(item))
        item.cancel()
    return

if __name__ == "__main__":
    asyncio.run(main())

'''
Task 7 had an unlucky failure
(4, 0.3565076327157949)
(2, 0.20337667536574278)
(9, 0.1277640138373286)
cancelling  5
cancelling  8
cancelling  3
cancelling  6
cancelling  0
cancelling  1
'''
```

Note that the *done* variable set is guranteed to contain minimally one element - the task that threw an exception is considered as done.