Python: AsyncIO

Introduction

Synchronous programming is when each statement is executed sequentially. Therefore, each statement has to wait for the previous one to finish executing:

```
# This executes first
func1()

# This waits to execute until AFTER func1() is finished
func2()
```

In Asynchronous programming is a technique that allows code to be executed independent of others' state. Therefore, a task is not required to be completely finished before executing other parts of code. This is useful when waiting for network operations or maintaining data flow from a websocket.

The idea of switching and assigning tasks is referred to as *concurrency*. Asynchronous programming refers to a single thread where the thread is handed-off to different jobs. There is a saying, "use async IO when you can; use threading when you must." While building durable multithreaded code in Python can sometimes be difficult and error-prone, async IO avoids some of these issues.

Coroutines & Tasks

Firstly, a 'coroutine' must be created. A coroutine is a function that can be suspended before reaching a `return` statement and indirectly pass control to another coroutine for some time. In Python, this is done by specifying `async` before the function, which creates a wrapper around the function to allow it to run asynchronously. However, this alters the function, so it no longer functions as a simple, vanilla function. Instead, the function now returns a coroutine object:

```
import asyncio

# This is not a typical function now, but a coroutine!
async def main():
    print("Hello World!")

# Run the coroutine
asyncio.run(main())
```

This basic script can now be run from the command line:

```
$ python my_file.py
Hello World!
```

The `await` keyword is placed before code to wait on its completion before continuing, which creates idle time. In this example, we tell the asynchronous function to wait for 5 seconds before continuing:

```
import asyncio

async def main():
    print("Hello World!")
    await asyncio.sleep(5)
    print("I just waited for 5 seconds!")

asyncio.run(main())
```

However, there is nothing very asynchronous happening in the example above. To run processes asynchronously, we must define tasks using `create_task()` and pass in the coroutine. In the example below, a task is created and waited to finish. Once it is finished, the code continues:

```
import asyncio
from asyncio import Task                          # Import `Task` for type-
hints

async def foo():
    print("This goes first!")
    await asyncio.sleep(2)

async def main():
    my_task: Task = asyncio.create_task(foo())  # Make `foo()` a task
    await my_task                              # Let the task finish
running and wait for completion...
    print("This goes second, after waiting!")

asyncio.run(main())
```

Tasks run asynchronously when there is idle time specified by `await`. In the example below, these two functions trade off when they each have idle time:

```
from asyncio import Task
import asyncio

async def main():
    my_task: Task = asyncio.create_task(foo())  # 1. Declare a task: run
this when there is idle time
    print("This goes first!")                   # 2. This runs fist
    await asyncio.sleep(2)                      # 3. Create idle time, let
the foo() task run
```

```
    print("This goes third!")                   # 6. This runs third during
the foo() idle time
    foo_return: str = await my_task             # 7. Don't cut foo() short;
let it finish
    print(foo_return)                           # 9. Finish by printing the
returned result of foo()

async def foo() -> str:
    print("This goes second!")                  # 4. This runs second during
the main() idle time
    await asyncio.sleep(2)                      # 5. Create idle time, let
main() continue
    return "This goes fourth!"                  # 8. Run this, as main()
awaits foo() to finish

asyncio.run(main())
```

Executing Multiple Coroutines

When calling multiple coroutines, they can be run individually or they can be grouped together. Consider the following code, which runs asynchronous functions individually:

```
import asyncio

async def brew_coffee() -> str:
    print("Start brewing coffee...")
    await asyncio.sleep(3)
    print("Done brewing coffee.")
    return "Coffee is ready!"

async def toast_bagel() -> str:
    print("Start toasting bagel...")
    await asyncio.sleep(3)
    print("Done toasting bagel.")
    return "Bagel is ready!"

async def main():
    coffee_result: str = brew_coffee()
    bagel_result: str = toast_bagel()
```

An alternative to this would be to create tasks and call. However, an additional method that is just as fast is to group the coroutines together.
The main function can be rewritten to group coroutine execution together using the `gather` method. The batched coroutines are then run with `await` and return values can be unpacked, if any. For example:

```python
...

async def main():
    # Group the coroutines together in a batch
    batch_coroutines = asyncio.gather(brew_coffee(), toast_bagel())

    # Run the coroutine group
    coffee_result, bagel_result = await batch_coroutines

asyncio.run(main())
```

Event Loops

An event loop is the process that monitors coroutines, takes feeback on what is idle, and searches for things that can be executed in the meantime. It is essentially the 'manager' of asynchronous functions. So far, we have seen the event loop being called with `asyncio.run(main())`. This process gets the event loop, runs tasks until they are complete, and then closes the event loop. Depending on the scenario, another way for managing an event loop is by instantiating it and running it in a *try* block:

```python
from asyncio import AbstractEventLoop   # Import `AbstractEventLoop` for type-hints
import asyncio

# Instantiate the loop
loop: AbstractEventLoop = asyncio.get_event_loop()

# Run the loop in a `try` bloc
try:
    loop.is_running()   # Returns `True`
    loop.run_until_complete(main())
finally:
    loop.close()
    loop.is_closed()    # Returns `True`
```

In this example, the `AbstractEventLoop` methods are used to check if the event loop is running or closed. However, typically `asyncio.run()` will suffice for running the event loop.