Python Event-Driven programming

In this tutorial, we will learn about Event-Driven programming and the Python module (Asyncio) that we can use to do Python Event-Driven programming.

Event-Driven programming

Eventually, the flow of a program depends upon the events, and programming which focuses on events is called Event-Driven programming. We were only dealing with either parallel or sequential models, but now we will discuss the asynchronous model. The programming model following the concept of Event-Driven programming is called the Asynchronous model. The working of Event-Driven programming depends upon the events happening in a program.

Other than this, it depends upon the program's event loops that always listen to a new incoming event in the program. Once an event loop starts in the program, then only the events will decide what will execute and in which order.

Look at the following flow chart of event loops to understand the working of events in event-driven programming:

Asyncio- Python Event-Driven programming module

The asyncio module was added into Python in version 3.4, and it is available on all the later versions of Python. Asyncio module provides a very good infrastructure for writing concurrent code as single-threaded using the coroutines in the program.

In the asyncio module of Python, following the different concepts are used while doing event-driven programming:

- The event loop
- Futures
- Coroutines
- @asyncio.coroutine decorator
- Tasks
- Transports
- and Protocols

Let's learn about all these different concepts used by how they work while doing event-driven programming

The event loop

The event loop is a functionality of the asyncio module used to handle all the events happening in a computational program. Event loops act like round the way while the execution of the whole program is happening, and it also keeps track of the executed and new incoming events. One of the major advantages of the asyncio module is that it allows only a single event loop per process.

In the asyncio module, we have some methods by which we can manage the event loop in the code. Following are such methods are provided in the asyncio module.

- **time() -** We can use this method to return the current time to the user, according to the internal clock present in the event loop.
- **loop = get_event_loop() -** By using this method, we will get the event loop in return according to the current context executing in the program.
- **call_soon(CallbackFunction, ArgumentGiven) -** With this function, we can arrange that a callback function will be called as soon as possible. The callback function given in the argument will be called when the control returns to the event loop and after call_soon() returns.
- **call_later(time_to_delayed, CallbackFunction, ArgumentGiven) -** Using this method, we can arrange that the callback function given in the method will be called after the time delay (in seconds) we provided.
- **new_event_loop() -** Using this method, we can create and return a new event loop project.
- **set_event_loop() -** With the help of this method, we can set the event loop for the current context executing from the program to the loop.
- **run_forever() -** By using the loop.run_forever() method in the program, we can run a loop until the stop() method is called.

Now, we will look at a Python program in the following example where we will use the event loop method, i.e., get_event_loop() method. By using this method, we will print the command we give inside the event.

Example: Look at the following Python program with event loops in it:

```
# Importing asyncio module in the program
import asyncio
# A default function with event loops method
def loopText(loop):
  # A text printing command
    print('Printing this text through the event loop')
    loop.stop() # Stopping the loop
loop = asyncio.get_event_loop() # Using get_event_loop() method to print the text
loop.call_soon(loopText, loop) # Using call_soon() method from event loops
loop.run_forever() # run_forever() on event loop
loop.close() # Closing the loop
```

Output

```
Printing this text through the event loop
```

Explanation:

We have first imported the asyncio module in the program to use the event loops methods.

After that, we have defined a default function where we gave 'loop' as a parameter and print command inside the function. We used the stop() method from event loops to stop the event.

After that, we used the get_event_loop() method on the loop parameter to print the text from the default function. Then, in the call_soon() event loo[illegible] parameter of a function as arguments. In last, we used

Futures

Future classes given in the asyncio module are compa
future class given in the asyncio module represents th
to be accomplished.

There are some major differences between the concurrent.futures.Future and asyncio.futures.

- The callback function that we have registered in the future class with the add_done_callback() method will always be called only through the call_soon() method from the event loop.
- The exception() and result() methods from the future class will not take a timeout or time given argument, and they will show an error in the output when the future isn't done yet on these functions.
- We cannot use asyncio.futures.Future class with the as_completed() or wait() functions present in the concurrent.futures package as it is not compatible with them.

Now, we will look at a Python program in the following example, where we will use the future class methods from the asyncio module and print the text in the output.

Example:

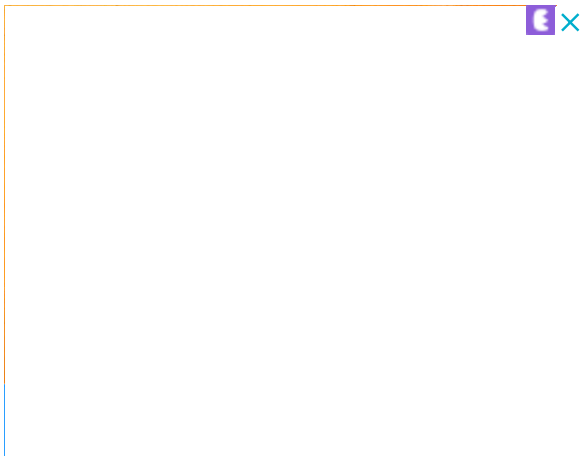

```
# Importing asyncio module in the program
```

```
import asyncio
# A default function from the async module using future parameter in it
async def myFunction(future):
  await asyncio.sleep(2) # Using sleep() function of asyncio module
  future.set_result('This text is printed using future class methods!') # Printing text from future param
# Using get_event_loop() method from event loop
loop = asyncio.get_event_loop()
future = asyncio.Future() # Using future() class method
# Calling default function from future class method
asyncio.ensure_future(myFunction(future))
# Using try & finally method future parameter of function
try:
  loop.run_until_complete(future)
  print(future.result()) # printing result from future class
finally:
    loop.close() # finally closing the loop
```

Output

```
This text is printed using future class methods!
```

Explanation:



We have first defined a default function, i.e., 'myFun
function, we used the sleep() method as a two-second

Then, we gave the text which we want to print in the result using future class methods. We used the get_event_loop method from the event loop in the program. Then, we used the future() class method on the future parameter given in the default function.

Now, we can print the text in the output as we have used the future in it. To print the text in the output, we have used the try and finish method, wherein the try method, we have called the printing command, and in the finish method, we closed the loop using the close() method.

Coroutines

The concept of coroutines inside the asyncio module is very similar to the concept of coroutine in the thread object from the threading module.

This concept of coroutines in the asyncio module is the generalization form of the subroutine concept.

We can even suspend a coroutine during the program's execution so that the suspended coroutine will wait for external processing given by the user. The suspended coroutine will return to where it had last suspended only after the external processing was done completely.

In asyncio module coroutines, we can use the following two ways that will help us to implement the coroutines in the program:

1. @asyncio.coroutine decorator
2. async def function()

Let's understand both ways by using their implementat

1. @asyncio.coroutine decorator

We can implement the coroutines inside the program by utilizing the generators with the asyncio module decorator, i.e., @asyncio.coroutine decorator. We can understand this implementation of coroutine with decorator through the following example.

Example: Look at the following Python program:

```
# Importing asyncio module in the program
import asyncio
# Using @asyncio.coroutine decorator to implement coroutines
@asyncio.coroutine
# Using a default function with coroutine implementation
def operationCoroutine():
    print("This text is present inside the coroutine of the asyncio module!") # Printing text inside corou
loop = asyncio.get_event_loop() # Using get_event_loop() method to print text
try:
    loop.run_until_complete(operationCoroutine()) # Using run_until_complete() loop method on defa
finally:
    loop.close() # closing the loop
```

Output

```
This text is present inside the coroutine of the asyncio module!
```

Explanation:

We have used the @asyncio.coroutine decorator after importing the asyncio module. Then, we used a default function to use the coroutine method to get the text. After that, we used the get_event_loop() method from the event loop to print the text in the output. Last, we used the 'try and finally' method on the default function and closed the loop in the program using the close() function.

2. async def function()

We can say that the async def function() is the most generalized method for implementing the coroutines by the asyncio module. We can understand this method of implementing coroutines with def function() through the following example.

Example: Look at the following Python program:

```
# Importing asyncio module in the program
import asyncio
# Using async def function() to implement coroutines
async def operationCoroutine():
   print("This text is present inside the coroutine of the asyncio module!") # Printing text inside corou
loop = asyncio.get_event_loop() # Using get_event_loop() method to print text
try:    loop.run_until_complete(operationCoroutine()) # Using run_until_complete() loop method on
finally:
   loop.close() # closing the loop
```

Output

```
This text is present inside the coroutine of the asyncio module!
```

Explanation:

Like the first way of implementing coroutines, we have followed the same path in this method too. In this method, instead of using a decorator and then defining the default function for coroutines, we have directly used the async def operationCoroutine() function of the asyncio module to implement coroutines.

Tasks

Tasks is a subclass given in the asyncio module responsible for executing asyncio coroutines inside an event loop in a parallel manner of execution. We can understand the working of tasks subclass by using a Python program to execute coroutines with it.

Example

```
# Importing asyncio module in the program
import asyncio
# Importing time module
import time
# Using async default function()
async def Task_ex(n):
  time.sleep(2) # sleep() function of time module
  print("Loop event is processing coroutine no: {}".format(n)) # given printing tasks to print in output
# Generating tasks with async default function
async def Generator_task():
      # looping over tasks using for loop
      for i in range(10):
           asyncio.ensure_future(Task_ex(i))
       # After completing loop
      print("All given tasks are completed")
      asyncio.sleep(2)
loop = asyncio.get_event_loop() # printing in output
loop.run_until_complete(Generator_task()) # Running
loop.close() # Closing the loop
```

Output

```
Loop event is processing coroutine no: 0
Loop event is processing coroutine no: 1
Loop event is processing coroutine no: 2
Loop event is processing coroutine no: 3
Loop event is processing coroutine no: 4
Loop event is processing coroutine no: 5
Loop event is processing coroutine no: 6
Loop event is processing coroutine no: 7
Loop event is processing coroutine no: 8
Loop event is processing coroutine no: 9
All given tasks are completed
```

Explanation:

We have imported asyncio and time modules in the program to use its functions. Then, we used an async default function to set the task of printing processed coroutines.

We used the sleep() function of the time module to give a break of 2 seconds after printing every executed coroutine.

Then, we used another async default function to set the loop over tasks in it. After completing loops, the function will print 'task completed.' In last, we used the event loop method for running and closing the loop in the program.

Transports

Transports are the classes provided to us in the asyncio module, and we can use them to implement various types of communication in the program. The transport classes are not thread-safe, and we always have to pair them with a protocol instance after the communication channel is established.

In the asyncio transport class, the following types of tr
the base transport class:

1. **Datagram transport**: Datagram transport is an i
2. **Read Transport:** Read transport is an interface
   mode.

3. **Write Transport:** This transport is an interface for the inherited transport classes with only write-only mode.
4. **Base Subprocess transport:** The base Subprocess transport class functions very similarly to the base transport class.

In all the above-mentioned inherited transport classes, only the following distinct types of methods are subsequently transient from the base transport class:

1. **is_closing():** This method will return true only if the given transport class in the argument is already closed on is closing now.
2. **close():** This method is used to close the current transport class running in the program.
3. **get_protocol():** We can use the get_protocol() method in the transport class to get the current protocol in return.
4. **get_extra_info(className, default = none**): We can use this method to get some additional information about the transport class we gave in the argument.

Protocols

In the asyncio module, we are provided with several base classes which we can use to implement our network protocols in the subclasses. We can use such classes in conjunction with transport classes. The protocol will ask for the outgoing data and parses the incoming data, whereas the transport class is responsible for the buffering and actual I/O.

Following are the three classes of protocols:

1. **Protocol class**: It is the base class in protocols, and we can use it for implementing streaming protocols to use with the SSL and TCP transports.
2. **Datagram protocol class:** It is another base class in protocols that we can use for implementing the datagram protocols to use with the UDP transports.
3. **Subprocess protocol class:** We can use this base class from protocols to implement various protocols for communicating the child processes

← Prev