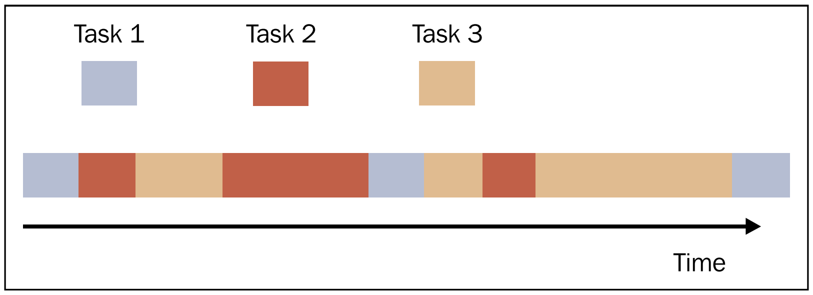
How it works...

In this example, we defined three asynchronous tasks, where each task calls the subsequent in the order, as shown in the following figure:



Task execution in the example

To accomplish this, we need to capture the event loop:

```
loop = asyncio.get_event_loop()
```

Then, we schedule the first call to `function_1()` by the `call_soon` construct:

```
end_loop = loop.time() + 9.0
loop.call_soon(function_1, end_loop, loop)
```

Let's note the definition of `function_1`:

```
def function_1(end_time, loop):
    print ("function_1 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_2, end_time, loop)
    else:
        loop.stop()
```

This defines the asynchronous behavior of the application with the following arguments:

- `end_time`: This defines the upper time limit within `function_1` and makes the call to `function_2` through the `call_later` method
- `loop`: This is the loop event that was captured previously with the `get_event_loop()` method

The task of `function_1` is pretty simple, which is to print its name, but it could also be more computationally intensive:

```
        print ("function_1 called")
```

After performing the task, it is compared to `loop.time ()` with the total length of the run; the total number of the cycles is `12` and if it is not passed this time, then it is executed with the `call_later` method with a delay of 1 second:

```
     if (loop.time() + 1.0) < end_time:
            loop.call_later(1, function_2, end_time, loop)
        else:
            loop.stop()
```

For `funcion_2()` and `function_3()`, the operation is the same.

If the running time expires, then the loop event must end:

```
    loop.run_forever()
    loop.close()
```

Handling coroutines with Asyncio

We saw, in the course of the various examples presented, that when a program becomes very long and complex, it is convenient to divide it into subroutines, each of which realizes a specific task for which it implements a suitable algorithm. The subroutine cannot be executed independently, but only at the request of the main program, which is then responsible for coordinating the use of subroutines. Coroutines are a generalization of the subroutine. Like a subroutine, the coroutine computes a single computational step, but unlike subroutines, there is no main program that can be used to coordinate the results. This is because the coroutines link themselves together to form a pipeline without any supervising function responsible for calling them in a particular order. In a coroutine, the execution point can be suspended and resumed later after keeping track of its local state in the intervening time. Having a pool of coroutines, it is possible to interleave their computations: run the first one until it yields the control back, then run the second, and so on down the line.

The control component of the interleave is the even loop, which was explained in the previous recipe. It keeps track of all the coroutines and schedules when they will be executed.

The other important aspects of coroutines are, as follows:

- Coroutines allow multiple entry points that can be yielded multiple times
- Coroutines can transfer the execution to any other coroutines

The term "yield" is used to describe a coroutine that pauses and passes the control flow to another coroutine. Since coroutines can pass values along with the control flow to another coroutine, the phrase "yielding a value" is used to describe the yielding and passing of a value to the coroutine that receives the control.

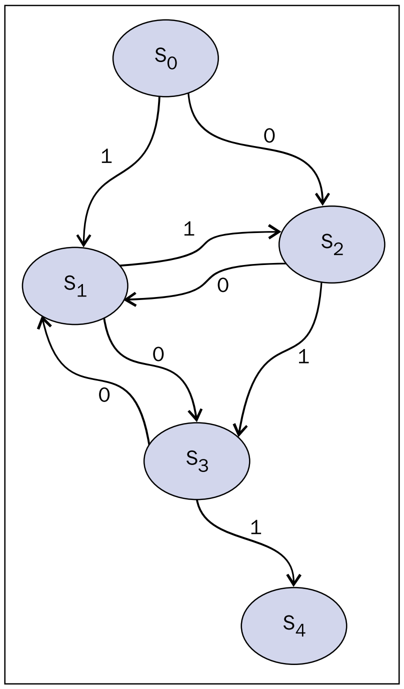
Getting ready

To define a coroutine with the Asyncio module, we simply use an annotation:

```
import asyncio
@asyncio.coroutine
def coroutine_function( function_arguments ) :
    # DO_SOMETHING
```

How to do it...

In this example, we will see how to use the coroutine mechanism of Asyncio to simulate a finite state machine of five states. A **finite state machine or automaton** (**FSA**) is a mathematical model that is widely used not only in engineering disciplines, but also in sciences, such as mathematics and computer science. The automata through which we want to simulate the behavior is as follows:



Finite state machine

In the preceding diagram, we have indicated with **S0**, **S1**, **S2**, **S3**, and **S4** the states of the system. Here, **0** and **1** are the values for which the automata can pass from one state to the next (this operation is called a transition). So for example, the state **S0** can be passed to the state **S1** only for the value **1** and **S0** can be passed to the state **S2** only for the value **0.** The Python code that follows, simulates a transition of the automaton from the state **S0**, the so-called **Start State**, up to the state **S4**, the **End State**:

```
#Asyncio Finite State Machine

import asyncio
import time
from random import randint


@asyncio.coroutine
def StartState():
    print ("Start State called \n")
    input_value = randint(0,1)
    time.sleep(1)
    if (input_value == 0):
        result = yield from State2(input_value)
    else :
        result = yield from State1(input_value)
    print("Resume of the Transition : \nStart State calling "\
          + result)


@asyncio.coroutine
def State1(transition_value):
    outputValue =  str(("State 1 with transition value = %s \n"\
                        %(transition_value)))
    input_value = randint(0,1)
    time.sleep(1)
    print("...Evaluating...")
    if (input_value == 0):
        result =  yield from State3(input_value)
    else :
        result = yield from State2(input_value)
    result = "State 1 calling " + result
    return (outputValue + str(result))


@asyncio.coroutine
def State2(transition_value):
        outputValue =  str(("State 2 with transition value = %s \n" \
                            %(transition_value)))
        input_value = randint(0,1)
        time.sleep(1)
        print("...Evaluating...")
        if (input_value == 0):
            result = yield from State1(input_value)
        else :
            result = yield from State3(input_value)
        result = "State 2 calling " + result
        return (outputValue + str(result))


    @asyncio.coroutine
    def State3(transition_value):
        outputValue =  str(("State 3 with transition value = %s \n" \
                            %(transition_value)))
        input_value = randint(0,1)
        time.sleep(1)
        print("...Evaluating...")
        if (input_value == 0):
            result = yield from State1(input_value)
        else :
            result = yield from EndState(input_value)
        result = "State 3 calling " + result
        return (outputValue + str(result))


    @asyncio.coroutine
    def EndState(transition_value):
        outputValue =  str(("End State with transition value = %s \n"\
                            %(transition_value)))
        print("...Stop Computation...")
        return (outputValue )

    if __name__ == "__main__":
        print("Finite State Machine simulation with Asyncio Coroutine")
        loop = asyncio.get_event_loop()
        loop.run_until_complete(StartState())
```

After running the code, we have an output similar to this:

```
C:\Python CookBook\Chapter 4- Asynchronous Programming\codes - Chapter
4>python asyncio_state_machine.py
```

```
Finite State Machine simulation with Asyncio Coroutine
Start State called
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Stop Computation...
Resume of the Transition :
Start State calling State 1 with transition value = 1
State 1 calling State 3 with transition value = 0
State 3 calling State 1 with transition value = 0
State 1 calling State 2 with transition value = 1
State 2 calling State 3 with transition value = 1
State 3 calling State 1 with transition value = 0
State 1 calling State 2 with transition value = 1
State 2 calling State 1 with transition value = 0
State 1 calling State 3 with transition value = 0
State 3 calling State 1 with transition value = 0
State 1 calling State 2 with transition value = 1
State 2 calling State 3 with transition value = 1
State 3 calling End State with transition value = 1
```

How it works...

Each state of the automata has been defined with the following annotation:

```
@asyncio.coroutine
```

For example, the state `S0` is defined as:

```
@asyncio.coroutine
def StartState():
    print ("Start State called \n")
    input_value = randint(0,1)
```

```
        time.sleep(1)
        if (input_value == 0):
            result = yield from State2(input_value)
        else :
            result = yield from State1(input_value)
```

The transition to the next state is determined by `input_value`, which is defined by the `randint(0,1)` function of Python's module `random`. This function provides randomly the value 0 or 1. In this manner, it randomly determines to which state the finite state machine will be passed:

```
input_value = randint(0,1)
```

After determining the value at which state the finite state machine will be passed, the coroutine calls the next coroutine using the command `yield` from:

```
if (input_value == 0):
        result = yield from State2(input_value)
    else :
        result = yield from State1(input_value)
```

The variable result is the value that each coroutine returns. It is a string, and by the end of the computation, we can reconstruct the transition from the initial state of the automation, the Start State, up to the final state, the End State.

The main program starts the evaluation inside the event loop as:

```
if __name__ == "__main__":
    print("Finite State Machine simulation with Asyncio Coroutine")
    loop = asyncio.get_event_loop()
    loop.run_until_complete(StartState())
```

Task manipulation with Asyncio

Asyncio is designed to handle asynchronous processes and concurrent task executions on an event loop. It also provides us with the `asyncio.Task()` class for the purpose of wrapping coroutines in a task. Its use is to allow independently running tasks to run concurrently with other tasks on the same event loop. When a coroutine is wrapped in a task, it connects the task to the event loop and then runs automatically when the loop is started, thus providing a mechanism to automatically drive the coroutine.