4
Asynchronous Programming

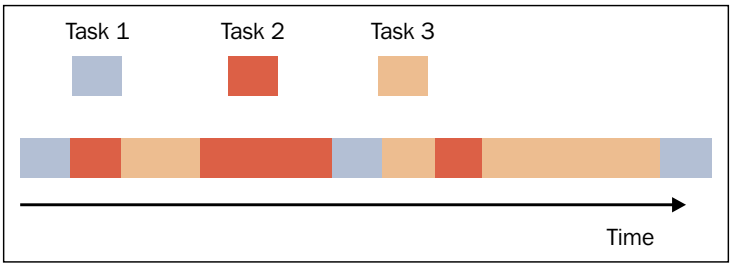
In this chapter, we will cover the following recipes:

- How to use the `concurrent.futures` Python module
- Event loop management with Asyncio
- Handling coroutines with Asyncio
- Task manipulation with Asyncio
- Dealing with Asyncio and Futures

Introduction

With the sequential and parallel execution model, there is a third model, called the asynchronous model, that is of fundamental importance to us along with the concept of event programming. The execution model of asynchronous activities can be implemented using a single stream of main control, both in uniprocessor systems and multiprocessor systems.

In the asynchronous model of a concurrent execution, various tasks intersect with the timeline, and all of this happens under the action of a single flow of control (single-threaded). The execution of a task can be suspended and then resumed, but this alternates the time of other tasks. The following figure expresses this concept in a clear manner:



Asynchronous programming model

As you can see, the tasks (each with a different color) are interleaved with one another, but they are in a single thread of control; this implies that when one task is in execution, the other tasks are not. A key difference between the multithreaded programming model and the single-threaded asynchronous concurrent model is that in the first case, the operating system decides on the timeline, whether to suspend the activity of a thread and start another, while in the second case, the programmer must assume that a thread may be suspended and replaced with another at any time.

The programmer can program a task as a sequence of smaller steps that are executed intermittently; so if a task uses the output of another, the dependent task must be written to accept its input.

Using the concurrent.futures Python modules

With the release of Python 3.2, the `concurrent.future` module was introduced, which allows us to manage concurrent programming tasks, such as process and thread pooling, nondeterministic execution flows, and processes and thread synchronization.

This package is built by the following classes:

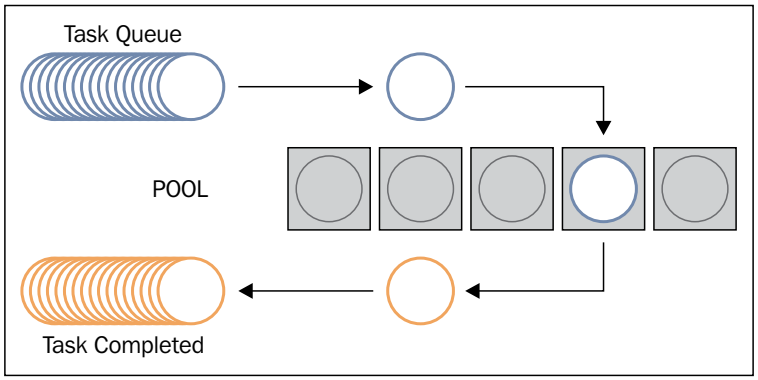
- `concurrent.futures.Executor`: This is an abstract class that provides methods to execute calls asynchronously.
- `submit (function ,argument)`: This schedules the execution of a function (called callable) on the arguments.

- `map (function,argument)`: This executes the function on arguments in an asynchronous mode.
- `shutdown (Wait = True)`: This signals the executor to free any resource.
- `concurrent.futures.Future`: This encapsulates the asynchronous execution of a callable function. Future objects are instantiated by submitting tasks (functions with optional parameters) to executors.

Executors are abstractions that are accessed through their subclasses: thread or process `ExecutorPools`. In fact, instantiation of threads and process is a resource-demanding task, so it is better to pool these resources and use them as repeatable launchers or executors (hence, the executors concept) for parallel or concurrent tasks.

Dealing with the process and thread pool

A thread or process pool (also called pooling) indicates a software manager that is used to optimize and simplify the use of threads and/or processes within a program. Through the pooling, you can submit the task (or tasks) that are to be executed to the pooler. The pool is equipped with an internal queue of tasks that are pending and a number of threads or processes that execute them. A recurring concept in pooling is reuse: a thread (or process) is used several times for different tasks during its lifecycle. It decreases the overhead of creating and increasing the performance of the program that takes advantage of the pooling. Reuse is not a rule, but it is one of the main reasons that lead a programmer to use pooling in his/her application.



Pooling management

Getting ready

The `current.Futures` module provides two subclasses of the `Executor` class, respectively, which manipulates a pool of threads and a pool of processes asynchronously. The two subclasses are as follows:

- `concurrent.futures.ThreadPoolExecutor(max_workers)`
- `concurrent.futures.ProcessPoolExecutor(max_workers)`

The `max_workers` parameter identifies the max number of workers that execute the call asynchronously.

How to do it...

The following example shows you the functionality of process and thread pooling. The task to be performed is that we have a list of numbers from one to 10, `number_list`. For each element of the list, a count is made up to 10,000,000 (just to waste time) and then the latter number is multiplied with the *i*th element of the list.

By doing this, the following cases are evaluated:

- Sequential execution
- A thread pool with 5 workers

Consider the following code:

```
#
# Concurrent.Futures Pooling - Chapter 4 Asynchronous Programming
#

import concurrent.futures
import time

number_list = [1,2,3,4,5,6,7,8,9,10]

def evaluate_item(x):
    #count...just to make an operation
    result_item = count(x)
    #print the input item and the result
    print ("item " + str(x) + " result " + str(result_item))

def count(number) :
    for i in range(0,10000000):
        i=i+1
    return i*number
```

```
if __name__ == "__main__":

##Sequential Execution
    start_time = time.clock()
    for item in number_list:
        evaluate_item(item)
    print ("Sequential execution in " + \
           str(time.clock() - start_time), "seconds")

    ##Thread pool Execution
    start_time_1 = time.clock()
    with concurrent.futures.ThreadPoolExecutor(max_workers=5)\
         as executor:
        for item in number_list:
            executor.submit(evaluate_item, item)
    print ("Thread pool execution in " + \
           str(time.clock() - start_time_1), "seconds")

    ##Process pool Execution
    start_time_2 = time.clock()
    with concurrent.futures.ProcessPoolExecutor(max_workers=5)\
         as executor:
        for item in number_list:
            executor.submit(evaluate_item, item)
    print ("Process pool execution in " + \
           str(time.clock() - start_time_2), "seconds")
```

After running the code, we have the following results with the execution time:

```
C:\Python CookBook\Chapter 4- Asynchronous Programming\ >python
Process_pool_with_concurrent_futures.py
item 1 result 10000000
item 2 result 20000000
item 3 result 30000000
item 4 result 40000000
item 5 result 50000000
item 6 result 60000000
item 7 result 70000000
item 8 result 80000000
item 9 result 90000000
item 10 result 100000000
Sequential execution in 17.241238674183425 seconds
```

```
item 4 result 40000000
item 2 result 20000000
item 1 result 10000000
item 5 result 50000000
item 3 result 30000000
item 7 result 70000000
item 6 result 60000000
item 8 result 80000000
item 10 result 100000000
item 9 result 90000000
Thread pool execution in 17.14648646290675 seconds

item 3 result 30000000
item 1 result 10000000
item 2 result 20000000
item 4 result 40000000
item 5 result 50000000
item 6 result 60000000
item 7 result 70000000
item 9 result 90000000
item 8 result 80000000
item 10 result 100000000
Process pool execution in 9.913172716938618 seconds
```

How it works...

We build a list of numbers stored in `number_list` and for each element in the list, we operate the counting procedure until 100,000,000 iterations. Then, we multiply the resulting value for 100,000,000:

```
def evaluate_item(x):
    #count...just to make an operation
    result_item = count(x)

def count(number) :
    for i in range(0,10000000):
        i=i+1
    return i*number
```

In the main program, we execute the task that will be performed in a sequential mode:

```
if __name__ == "__main__":
   for item in number_list:
       evaluate_item(item)
```

Also, in a parallel mode, we will use the `concurrent.futures` module's pooling capability for a thread pool:

```
    with concurrent.futures.ThreadPoolExecutor(max_workers=5)\
         as executor:
        for item in number_list:
            executor.submit(evaluate_item, item)
```

The `ThreadPoolExecutor` executes the given task using one of its internally pooled threads. It manages five threads working on its pool. Each thread takes a job out from the pool and executes it. When the job is executed, it takes the next job to be processed from the thread pool.

When all the jobs are processed, the execution time is printed:

```
print ("Thread pool execution in " + \
           str(time.clock() - start_time_1), "seconds")
```

For the process pooling implemented by the `ProcessPoolExecutor` class, we have:

```
    with concurrent.futures.ProcessPoolExecutor(max_workers=5)\
         as executor:
        for item in number_list:
            executor.submit(evaluate_item, item)
```

Like `ThreadPoolExecutor`, the `ProcessPoolExecutor` class is an executor subclass that uses a pool of processes to execute calls asynchronously. However, unlike `ThreadPoolExecutor`, the `ProcessPoolExecutor` uses the multiprocessing module, which allows us to outflank the global interpreter lock and obtain a shorter execution time.

There's more...

The pooling is used in almost all server applications, where there is a need to handle more simultaneous requests from any number of clients. Many other applications, however, require that each task should be performed instantly or you have more control over the thread that executes it. In this case, pooling is not the best choice.

Event loop management with Asyncio

The Python module Asyncio provides facilities to manage events, coroutines, tasks and threads, and synchronization primitives to write concurrent code. The main components and concepts of this module are:

- **An event loop**: The Asyncio module allows a single event loop per process
- **Coroutines**: This is the generalization of the subroutine concept. Also, a coroutine can be suspended during the execution so that it waits for external processing (some routine in I/O) and returns from the point at which it had stopped when the external processing was done.
- **Futures**: This defines the `Future` object, such as the `concurrent.futures` module that represents a computation that has still not been accomplished.
- **Tasks**: This is a subclass of Asyncio that is used to encapsulate and manage coroutines in a parallel mode.

In this recipe, the focus is on handling events. In fact, in the context of asynchronous programming, events are very important since they are inherently asynchronous.

What is an event loop

Within a computational system, the entity that can generate events is called an event source, while the entity that negotiates to manage an event is called the event handler. Sometimes, there may be a third entity called an event loop. It realizes the functionality to manage all the events in a computational code. More precisely, the event loop acts cyclically during the whole execution of the program and keeps track of events that have occurred within a data structure to queue and then process them one at a time by invoking the event handler if the main thread is free. Finally, we report a pseudocode of an event loop manager:

```
while (1) {
events = getEvents();
for (e in events)
processEvent(e);
}
```

All the events in the `while` loop are caught and then processed by the event handler. The handler that processes an event is the only activity that takes place in the system. When the handler has ended, the control is passed on to the next event that is scheduled.

Getting ready

Asyncio provides the following methods that are used to manage an event loop:

- `loop = get_event_loop()`: Using this, you can get the event loop for the current context.
- `loop.call_later(time_delay,callback,argument)`: This arranges for the callback that is to be called after the given `time_delay` seconds.
- `loop.call_soon(callback,argument)`: This arranges for a callback that is to be called as soon as possible. The callback is called after `call_soon()` returns and when the control returns to the event loop.
- `loop.time()`: This returns the current time, as a float value, according to the event loop's internal clock.
- `asyncio.set_event_loop()`: This sets the event loop for the current context to loop.
- `asyncio.new_event_loop()`: This creates and returns a new event loop object according to this policy's rules.
- `loop.run_forever()`: This runs until `stop()` is called.

How to do it...

In this example, we show you how to use the loop event statements provided by the Asyncio library to build an application that works in an asynchronous mode. Let's consider the following code:

```
import asyncio
import datetime
import time

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()
```

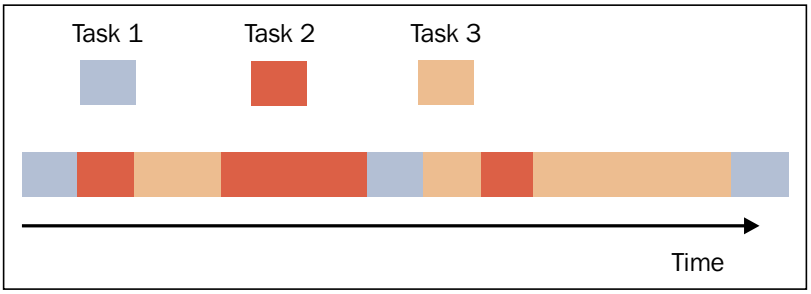
```
def function_2(end_time, loop):
    print ("function_2 called ")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_3, end_time, loop)
    else:
        loop.stop()

def function_3(end_time, loop):
    print ("function_3 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_1, end_time, loop)
    else:
        loop.stop()

def function_4(end_time, loop):
    print ("function_5 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_4, end_time, loop)
    else:
        loop.stop()

loop = asyncio.get_event_loop()

end_loop = loop.time() + 9.0
loop.call_soon(function_1, end_loop, loop)
#loop.call_soon(function_4, end_loop, loop)

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

The output of the preceding code is as follows:

```
C:\Python Parallel Programming INDEX\Chapter 4- Asynchronous
Programming >python asyncio_loop.py
function_1 called
function_2 called
function_3 called
function_1 called
function_2 called
function_3 called
function_1 called
function_2 called
function_3 called
```

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.

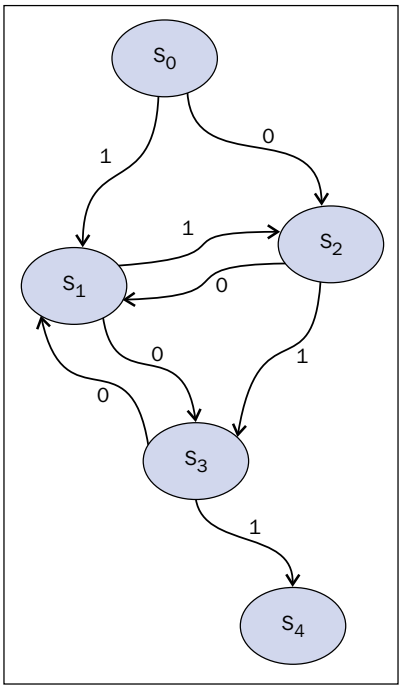
Getting ready

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.

Getting ready

The Asyncio module provides us with the `asyncio.Task(coroutine)` method to handle computations with tasks. It schedules the execution of a coroutine. A task is responsible for the execution of a coroutine object in an event loop. If the wrapped coroutine yields from a future, the task suspends the execution of the wrapped coroutine and waits for the completion of the future.

When the future is complete, the execution of the wrapped coroutine restarts with the result or the exception of the future. Also, we must note that an event loop only runs one task at a time. Other tasks may run parallelly if other event loops run in different threads. While a task waits for the completion of a future, the event loop executes a new task.

How to do it...

In the following sample code, we've shown you how three mathematical functions can be executed concurrently by the `Asyncio.Task()` statement:

```
"""
Asyncio using Asyncio.Task to execute three math function in parallel
"""
import asyncio

@asyncio.coroutine
def factorial(number):
    f = 1
    for i in range(2, number+1):
        print("Asyncio.Task: Compute factorial(%s)" % (i))
        yield from asyncio.sleep(1)
        f *= i
    print("Asyncio.Task - factorial(%s) = %s" % (number, f))

@asyncio.coroutine
def fibonacci(number):
    a, b = 0, 1
    for i in range(number):
        print("Asyncio.Task: Compute fibonacci (%s)" % (i))
        yield from asyncio.sleep(1)
        a, b = b, a + b
    print("Asyncio.Task - fibonacci(%s) = %s" % (number, a))
```

```
@asyncio.coroutine
def binomialCoeff(n, k):
    result = 1
    for i in range(1, k+1):
        result = result * (n-i+1) / i
        print("Asyncio.Task: Compute binomialCoeff (%s)" % (i))
        yield from asyncio.sleep(1)
    print("Asyncio.Task - binomialCoeff(%s , %s) = \
                    %s" % (n,k,result))

if __name__ == "__main__":
    tasks = [asyncio.Task(factorial(10)),
             asyncio.Task(fibonacci(10)),
             asyncio.Task(binomialCoeff(20,10))]
    loop = asyncio.get_event_loop()
    loop.run_until_complete(asyncio.wait(tasks))
    loop.close()
```

The result of the preceding code is:

```
C:\ Python CookBook \Chapter 4- Asynchronous Programming\codes -
Chapter 4> python asyncio_Task.py
Asyncio.Task: Compute factorial(2)
Asyncio.Task: Compute fibonacci (0)
Asyncio.Task: Compute binomialCoeff (1)
Asyncio.Task: Compute factorial(3)
Asyncio.Task: Compute fibonacci (1)
Asyncio.Task: Compute binomialCoeff (2)
Asyncio.Task: Compute factorial(4)
Asyncio.Task: Compute fibonacci (2)
Asyncio.Task: Compute binomialCoeff (3)
Asyncio.Task: Compute factorial(5)
Asyncio.Task: Compute fibonacci (3)
Asyncio.Task: Compute binomialCoeff (4)
Asyncio.Task: Compute factorial(6)
Asyncio.Task: Compute fibonacci (4)
Asyncio.Task: Compute binomialCoeff (5)
Asyncio.Task: Compute factorial(7)
Asyncio.Task: Compute fibonacci (5)
Asyncio.Task: Compute binomialCoeff (6)
Asyncio.Task: Compute factorial(8)
Asyncio.Task: Compute fibonacci (6)
Asyncio.Task: Compute binomialCoeff (7)
Asyncio.Task: Compute factorial(9)
Asyncio.Task: Compute fibonacci (7)
```

```
Asyncio.Task: Compute binomialCoeff (8)
Asyncio.Task: Compute factorial(10)
Asyncio.Task: Compute fibonacci (8)
Asyncio.Task: Compute binomialCoeff (9)
Asyncio.Task - factorial(10) = 3628800
Asyncio.Task: Compute fibonacci (9)
Asyncio.Task: Compute binomialCoeff (10)
Asyncio.Task - fibonacci(10) = 55
Asyncio.Task - binomialCoeff(20 , 10) = 184756.0
```

How it works...

In this example, we defined three `coroutines`, `factorial`, `fibonacci`, and `binomialCoeff` each of which, as explained previously, is identified by the `@asyncio.coroutine` decorator:

```
@asyncio.coroutine
def factorial(number):
do Something

@asyncio.coroutine
def fibonacci(number):
do Something

@asyncio.coroutine
def binomialCoeff(n, k):
do Something
```

To perform these three tasks parallelly, we first put them in the list tasks, in the following manner:

```
if __name__ == "__main__":
    tasks = [asyncio.Task(factorial(10)),
             asyncio.Task(fibonacci(10)),
             asyncio.Task(binomialCoeff(20,10))]
```

Then, we get the `event_loop`:

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

Next, we run the tasks:

```
loop.run_until_complete(asyncio.wait(tasks))
```

Here, `asyncio.wait statement (tasks)` waits for the given coroutines to complete.

In the last statement, we close the event loop:

```
loop.close()
```

Dealing with Asyncio and Futures

Another key component of the Asyncio module is the `Future` class. This is very similar to `concurrent.futures.Futures`, but of course, it is adapted in the main mechanism of Asyncio's event loop. The `asyncio.Future` class represents a result (but can also be an exception) that is not yet available. It therefore represents an abstraction of something that is yet to be accomplished.

Callbacks that have to process any results are in fact added to the instances of this class.

Getting ready

To manage an object `Future` in Asyncio, we must declare the following:

```
import asyncio
future = asyncio.Future()
```

The basic methods of this class are:

- `cancel()`: This cancels the future and schedules callbacks
- `result()`: This returns the result that this future represents
- `exception()`: This returns the exception that was set on this future
- `add_done_callback(fn)`: This adds a callback that is to be run when `future` is executed
- `remove_done_callback(fn)`: This removes all instances of a callback from the "call when done" list
- `set_result(result)`: This marks the future as complete and sets its result
- `set_exception(exception)`: This marks the future as complete and sets an exception

How to do it...

The following example shows you how to use the `Futures` class for the management of two coroutines `first_coroutine` and `second_coroutine` that perform the tasks, such as the sum of the first *n* integers and second the factorial of *n*. The code is as follows:

```
"""
Asyncio.Futures -  Chapter 4 Asynchronous Programming
"""
import asyncio
import sys

#SUM OF N INTEGERS
@asyncio.coroutine
def first_coroutine(future,N):
    count = 0
    for i in range(1,N+1):
        count=count + i
    yield from asyncio.sleep(4)
    future.set_result("first coroutine (sum of N integers) result = "\
                      + str(count))


#FACTORIAL(N)
@asyncio.coroutine
def second_coroutine(future,N):
    count = 1
    for i in range(2, N+1):
        count *= i
    yield from asyncio.sleep(3)
    future.set_result("second coroutine (factorial) result = "\
                      + str(count))

def got_result(future):
    print(future.result())
```

```
if __name__ == "__main__":
    N1 = int(sys.argv[1])
    N2 = int(sys.argv[2])

    loop = asyncio.get_event_loop()
    future1 = asyncio.Future()
    future2 = asyncio.Future()

    tasks = [
        first_coroutine(future1,N1),
        second_coroutine(future2,N2)]

    future1.add_done_callback(got_result)
    future2.add_done_callback(got_result)

    loop.run_until_complete(asyncio.wait(tasks))
    loop.close()
```

The following output is obtained after multiple runs:

```
C:\Python CookBook\Chapter 4- Asynchronous Programming\codes - Chapter
4>python Asyncio_future.py 1 1
first coroutine (sum of N integers) result = 1
second coroutine (factorial) result = 1

C:\Python CookBook\Chapter 4- Asynchronous Programming\codes - Chapter
4>python Asyncio_future.py 2 2
first coroutine (sum of N integers) result = 3
second coroutine (factorial) result = 2

C:\ Python CookBook\Chapter 4- Asynchronous Programming\codes -
Chapter 4>python Asyncio_future.py 3 3
first coroutine (sum of N integers) result = 6
second coroutine (factorial) result = 6

C:\ Python CookBook\Chapter 4- Asynchronous Programming\codes -
Chapter 4>python Asyncio_future.py 5 5
first coroutine (sum of N integers) result = 15
second coroutine (factorial) result = 120
```

How it works...

In the main program, we define the objects' future to associate the coroutines:

```
if __name__ == "__main__":

future1 = asyncio.Future()
future2 = asyncio.Future()
```

While defining the tasks, we pass the object future as an argument of coroutines:

```
tasks = [first_coroutine(future1,N1),
         second_coroutine(future2,N2)]
```

Finally, we add a callback that is to be run when the `future` gets executed:

```
future1.add_done_callback(got_result)
future2.add_done_callback(got_result)
```

Here, `got_result` is a function that prints the final result of the future:

```
def got_result(future):
    print(future.result())
```

In the coroutine wherein we pass the object future as an argument, after the computation, we set a sleep time of 3 seconds for the first coroutine and 4 seconds for the second coroutine:

```
yield from asyncio.sleep(sleep_time)
```

Then, we mark the future as complete and set its result with the help of `future.set_result()`.

There's more...

Swapping the sleep time between the coroutines, we invert the output results (we first do that for the second coroutine output):

```
C:\Python CookBook\Chapter 4- Asynchronous Programming\codes - Chapter
4>python Asyncio_future.py 1 10
second coroutine (factorial) result = 3628800
first coroutine (sum of N integers) result = 1
```