**Asynchronous programming**

**Coroutines:**

A coroutine is a function that can suspend execution to be resumed later.

To create a coroutine we should add an **async** keyword before a function.

Example:

|  |
| --- |
| async def printing\_hello():  print("Hello world") |

We cannot run a coroutine function like normal functions. If we run like a normal function it will return a coroutine object.

To run a coroutine function we can use **asyncio.run()**

|  |
| --- |
| import asyncio  async def printing\_hello():  print("Hello world")  asyncio.run(printing\_hello())  # Output  Hello World |

**Tasks:**

*Tasks* are used to schedule coroutines *concurrently*.

To create a task we can use **asyncio.create\_task()**

|  |
| --- |
| import asyncio  async def nested(value):  print(“Printing the value”, end= “ ”)  await asyncio.sleep(0.5)  print(value)  async def main():  # Schedule nested() to run soon concurrently  # with "main()".  task = asyncio.create\_task(nested(10))   # "task" can now be used to cancel "nested()", or  # can simply be awaited to wait until it is complete:  await task  asyncio.run(main())  # Output  Printing the value 10 |

**Await:**

The **await** keyword will ask the execution to wait until the defined task gets executed.

If we don’t use the **await task** statement then the output will be

|  |
| --- |
| import asyncio  async def nested(value):  print(“Printing the value”, end= “ ”)  await asyncio.sleep(0.5)  print(value)  async def main():  task = asyncio.create\_task(nested(10))  # Output Printing the value |

Because as soon as the task is created the print statement in nested will be executed and then we are awaiting the execution to sleep for 0.5 seconds. Meanwhile in this time there are no statements to execute so the main() will exit. So it won’t print the value.

**Running multiple tasks:**

To run multiple tasks concurrently we use **asyncio.gather():**

|  |
| --- |
| import asyncio  async def factorial(name, number):  f = 1  for i in range(2, number + 1):  print(f"Task {name}: Compute factorial({number}), currently i={i}...")  await asyncio.sleep(1)  f \*= i  print(f"Task {name}: factorial({number}) = {f}")  return f  async def main():  # Schedule three calls \*concurrently\*:  L = await asyncio.gather(  factorial("A", 2),  factorial("B", 3),  factorial("C", 4),  )  print(L)  asyncio.run(main())  # Expected output: # # Task A: Compute factorial(2), currently i=2... # Task B: Compute factorial(3), currently i=2... # Task C: Compute factorial(4), currently i=2... # Task A: factorial(2) = 2 # Task B: Compute factorial(3), currently i=3... # Task C: Compute factorial(4), currently i=3... # Task B: factorial(3) = 6 # Task C: Compute factorial(4), currently i=4... # Task C: factorial(4) = 24 # [2, 6, 24 |

**Timeouts:**

**asyncio.wait\_for()**

To complete a task within the given time, if don’t it will raise an exception like

**asyncio.TimeoutError**

|  |
| --- |
| async def eternity():  # Sleep for one hour  await asyncio.sleep(3600)  print('yay!')  async def main():  # Wait for at most 1 second  try:  await asyncio.wait\_for(eternity(), timeout=1.0)  except asyncio.TimeoutError:  print('timeout!')  asyncio.run(main())  # Expected output: # # timeout! |

We can also use **asyncio.wait(*aws*, *\**, *loop=None*, *timeout=None*, *return\_when=ALL\_COMPLETED*)**

**Futures**

Future is an awaitable object. Coroutines can await on Future objects until they either have a result or an exception set, or until they are cancelled. ... Return the result of the Future.

|  |
| --- |
| import asyncio  async def set\_after(fut, delay, value):  # Sleep for \*delay\* seconds.  await asyncio.sleep(delay)   # Set \*value\* as a result of \*fut\* Future.  fut.set\_result(value)  async def main():  # Get the current event loop.  loop = asyncio.get\_running\_loop()   # Create a new Future object.  fut = loop.create\_future()   # Run "set\_after()" coroutine in a parallel Task.  # We are using the low-level "loop.create\_task()" API here because  # we already have a reference to the event loop at hand.  # Otherwise we could have just used "asyncio.create\_task()".  loop.create\_task(  set\_after(fut, 1, '... world'))   print('hello ...')   # Wait until \*fut\* has a result (1 second) and print it.  print(await fut)  asyncio.run(main()) |

**Event loop**

Event loop is the core of Python asyncio . Every coroutine , Future , or Task would be scheduled as a callback and be executed by an event loop.

On each iteration, the event loop tries to synchronously pull out the next callback from the queue. If there is no callback to execute at the moment, pop() blocks the main thread. When the callback is ready, the event loop executes it. The execution of a callback always happens synchronously.

|  |
| --- |
| import asyncio  def hello\_world(loop):  """A callback to print 'Hello World' and stop the event loop"""  print('Hello World')  loop.stop()  loop = asyncio.get\_event\_loop()  # Schedule a call to hello\_world() loop.call\_soon(hello\_world, loop)  # Blocking call interrupted by loop.stop() try:  loop.run\_forever() finally:  loop.close() |