

# Asyncio Internals

Wing

October 23, 2020

# Table of contents

- 1 Generator
- 2 Event loop
- 3 Task
- 4 Code flow
- 5 References

# Generator internals

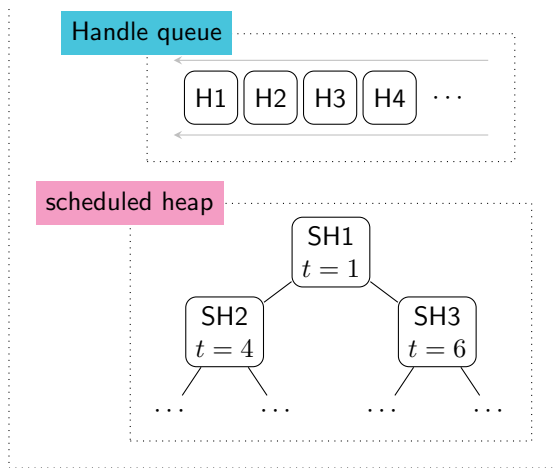
Secret: In cpython, generator = coroutine!

- `gen.send(?)`
- `next(gen) ≡ gen.send(None)`
- `gen.throw(exc)`
- `yield` ⇒ “pausing” of coroutine
- closure
- stack frame in heap

demo `gen_send.py`

# Event loop

Event loop



Handle wraps a callback (e.g. `Task.__step`)

# Task

Task

Future

callbacks

cb1

cb2

cb3

...

&loop

coroutine

## asyncio.run(main)

```
1 def run(main):
2     loop = new_event_loop()
3     return loop.run_until_complete(main)
4
5 class Loop:
6     def run_until_complete(coro_or_fut):
7         task = ensure_future(coro_or_fut,
8                               loop=self)
9         task.add_done_callback(<<stop loop>>)
10        loop.run_forever()
11        return task.result()
12
13 def ensure_future(coro_or_fut, *, loop):
14     if isinstance(coro_or_fut, Future):
15         return coro_or_fut
16     else: # coro
17         return Task(coro_or_fut, loop=loop)
```

```
class Task(Future):  
    def __init__(self, coro, loop, ...):  
        super().__init__(loop)  
        ...  
        self._coro = coro  
        self._loop.call_soon(self.__step, ...)
```

# Future

```
1
2 class Future:
3     def __init__(self, loop):
4         self._loop = loop
5
6     def __iter__(self):
7         if not self.done():
8             yield self # future is blocking
9         return self.result()
10
11    def add_done_callback(self, fn, ...):
12        if self.done():
13            self._loop.call_soon(fn, self)
14        else:
15            self._callbacks.append((fn,))
16    ...
```



# Future

```
1
2 class Future:
3     ...
4
5     def set_result(self, result): # similar for set_exception
6         self._result = result
7         self._state = _FINISHED # done
8         self.__schedule_callbacks()
9
10    def __schedule_callbacks(self):
11        callbacks = self._callbacks[:]
12        self._callbacks[:] = [] # clear callbacks
13        for callback in callbacks:
14            self._loop.call_soon(callback, self)
15
16    def result(self):
17        if self._exception is not None:
18            raise self._exception
19        return self._result
```

`__schedule_callbacks()` effectively moves all callbacks to **Handle queue**

```
loop.call_soon(callback, *args)
```

```
class Loop:
    def call_soon(self, callback, *args, ...):
        handle = Handle(callback, *args)
        Handle queue.append(handle)
    return handle
```

```
loop.run_forever()
```

```
class Loop:
    def run_forever(self):
        while True:
            self._run_once()
            if self._stopping:
                break

    def _run_once(self):
        ...
```

## loop.\_run\_once()

```
def _run_once():
```

```
    timeout =  $\begin{cases} 0, & \text{if Handle queue is not empty} \\ \text{minimal timeout,} & \text{if scheduled heap is not empty} \\ \text{None,} & \text{otherwise} \end{cases}$ 
```

```
    // block if timeout is None
```

```
    ev_list = self._selector.select(timeout)
```

```
    self._process_events(ev_list)
```

```
    Handle queue += handles from scheduled heap which the  
    time is up
```

```
    handles = pop all from Handle queue
```

```
    for handle: handles do
```

```
        | handle._run() // may run task.__step
```

## Task.\_\_step(exc)

```
1 def __step(exc):
2     coro = self._coro
3     try:
4         result = coro.send(None)
5     except StopIteration as exc:
6         self.set_result(exc.value)
7     except BaseException as exc:
8         self.set_exception(exc)
9     else:
10        if result <<is a blocking future>>:
11            result.add_done_callback(self.__wakeup)
12        elif result is None: # bare yield used
13            self._loop.call_soon(self.__step)
14
15 def __wakeup(self, future):
16     # check if cancelled or errored
17     ...
18     self.__step()
```

● `await`  $\equiv$  `yield from`

- `run_once_demo.py`
- `timeout_zero_future.py`



Talk by Saúl Ibarra Corretgé in PyGrunn 2014

<https://www.youtube.com/watch?v=HppNu0-ANYw>

Code based on Python 3.8.2