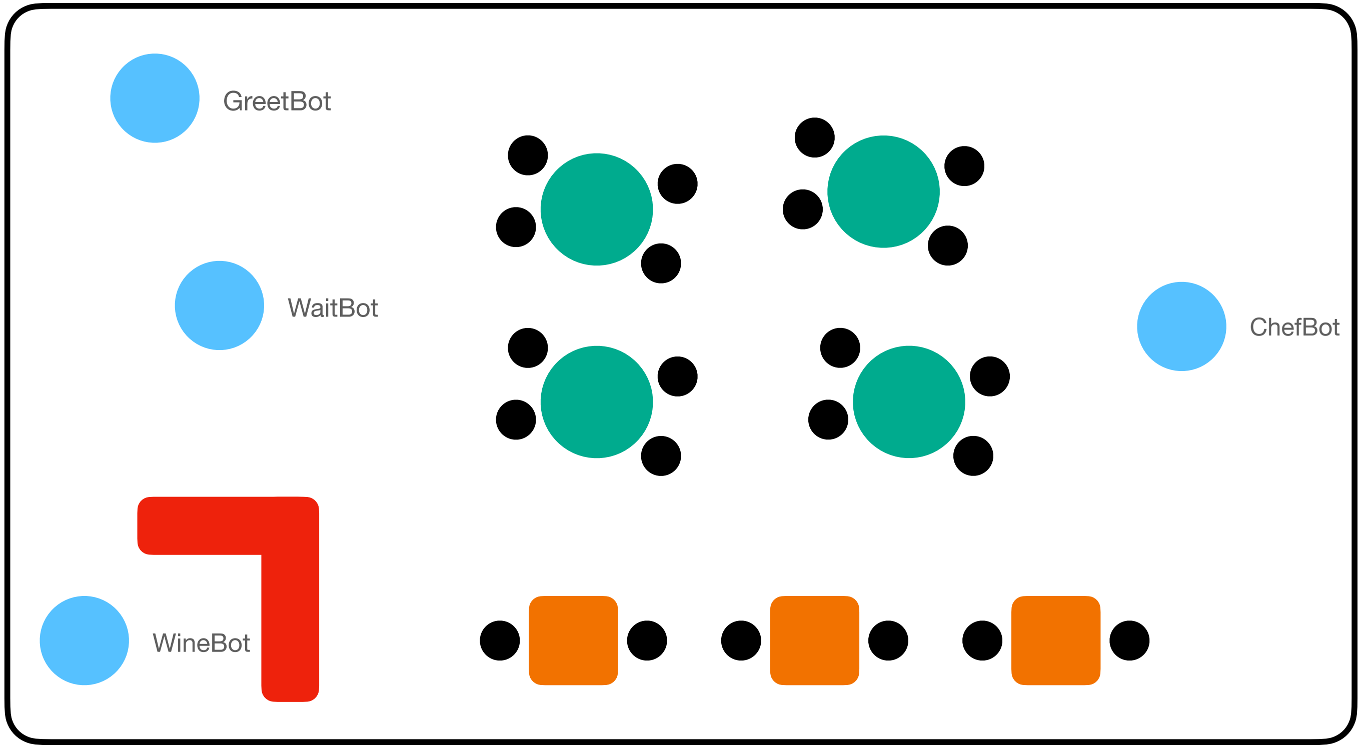
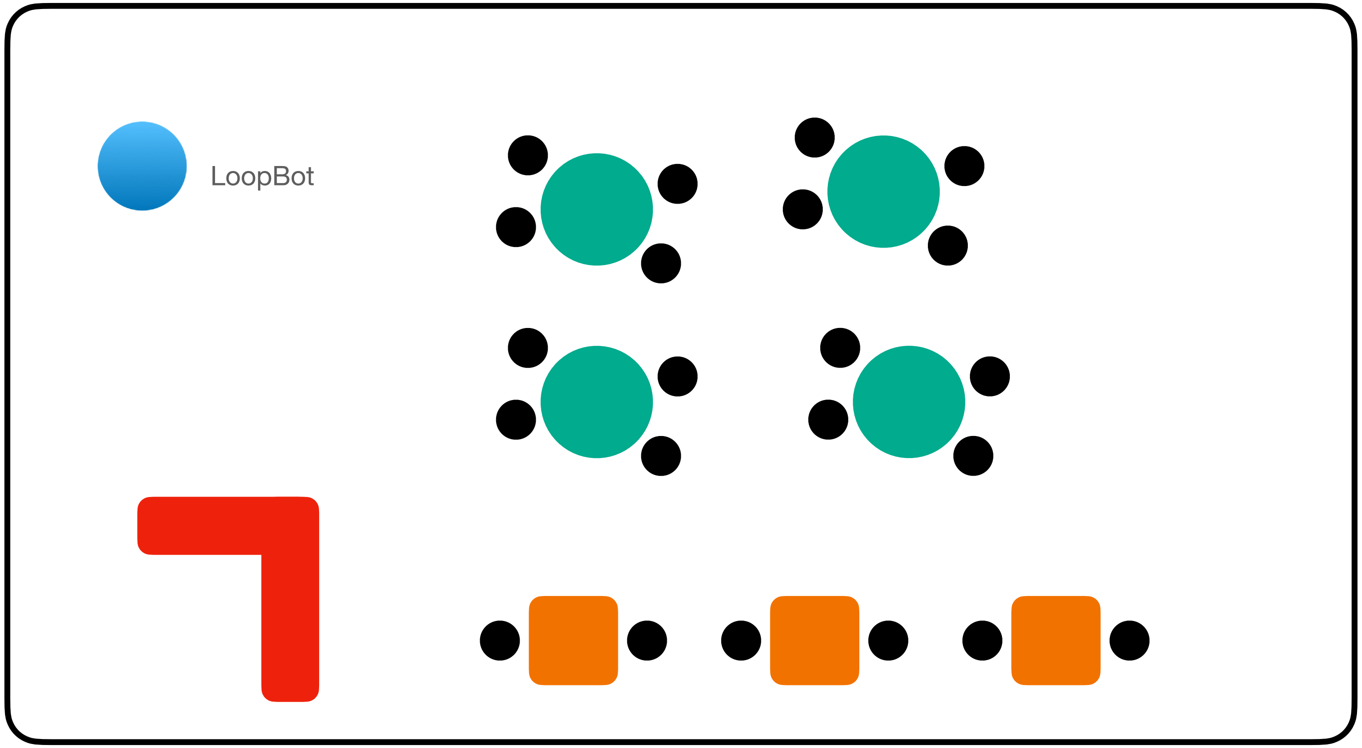
Asyncio

Coroutine Task Future

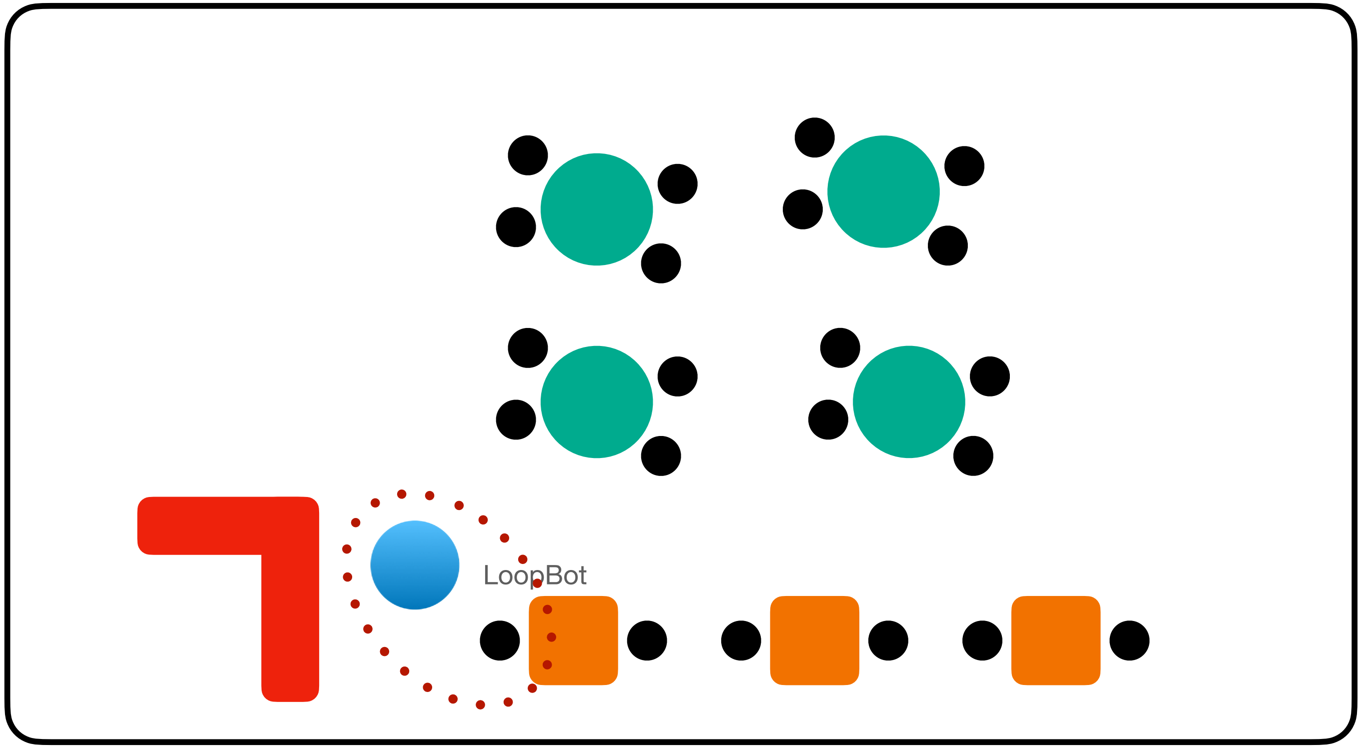
- GreetBot, greet diners at the front desk
- WaitBot, wait tables and take orders
- ChefBot, do the cooking
- WineBot, manage the bar



- LoopBot
  - greet diners at the front desk
  - wait tables and take orders
  - do the cooking
  - manage the bar



- LoopBot
  - Effectively only if task is short!



What problem is Asyncio Trying to solve?

For I/O-bound workloads, there are exactly (only!)

- Asyncio offers a safe alternative to preemptive multitasking.
  - Threads that have bugs, race conditions
- Asyncio offiers a simple way to support many thousands of simultaneous socket connections, many long-lived connections
  - newer technologies like WebSockets
  - MQTT for Internet of Things (IoT)

Asyncio misinformation

- Asyncio will make my code blazing fast
  - Asyncio for large numbers of concurrent socket connection
  - Operating system have limits threads can be created, and significant lower than the number of socket connections
- Asyncio makes threading redundant
  - Threading able to write multi-CPU programs
- Asyncio prevents all race conditions
  - Race conditions is always present with any concurrent programming
  - Asyncio control of execution is transferred between coroutines, by await

The Truth About Threads

- Threads are a features provided by an operating system (OS)
- Threads may indicated to OS which parts of their program may be run in parallel.

Benefits of Threading

- Easy of reading code - simple, top-down linear sequence
- Parallelism with shared memory - code can export multiple CPUs while still having threads share memory.
- Know-how and existing code - best practices available

Drawbacks of Threading

- Threading is difficult - Bugs and race conditions
- Threads are resource-intensive - require extra OS resources to create such as preallocated, per-thread stack space
- Threading can affect throughput - high concurrency level (> 5,000 threads)
- Threading is inflexible - OS share CPU time with all threads regardless of whether a thread is ready to do work or not.

The Tower of Asyncio

Table 3-1. Features of asyncio arranged in a hierarchy; for end-user developers, the most important tiers are highlighted in bold

Level	Concept	Implementation
Tier 9	**Network: streams**	`StreamReader`, `StreamWriter`, `asyncio.open_connection()`, `asyncio.start_server()`
Tier 8	Network: TCP & UDP	`Protocol`
Tier 7	Network: transports	`BaseTransport`
Tier 6	**Tools**	`asyncio.Queue`
Tier 5	**Subprocesses & threads**	`run_in_executor()`, `asyncio.subprocess`
Tier 4	Tasks	`asyncio.Task`, `asyncio.create_task()`
Tier 3	Futures	`asyncio.Future`
Tier 2	**Event loop**	`asyncio.run()`, `BaseEventLoop`
Tier 1 (Base)	**Coroutines**	`async def`, `async with`, `async for`, `await`

Coroutines

Example 3-4. Async functions are functions, not coroutines

```
>>> async def f():   ❶
...     return 123
...
>>> type(f)   ❷
<class 'function'>
>>> import inspect   ❸
>>> inspect.iscoroutinefunction(f)   ❹
True
```

Example 3-5. An async def function returns a coroutine object

```
>>> coro = f()
>>> type(coro)
<class 'coroutine'>
>>> inspect.iscoroutine(coro)
True
```

Coroutines

- A **Coroutine** is an **object** that encapsulates the ability to resume an underlining function that has been suspended before completion.

Coroutines

- A coroutine is initiated by "**sending**" it a None
- When the coroutine returns, a special kind of exception is raised, called **StopIteration**

Example 3-6. Coroutine internals: using send() and StopIteration

```
>>> async def f():
...     return 123
>>> coro = f()
>>> try:
...   coro.send(None)  ❶
... except StopIteration as e:
...   print('The answer was:', e.value)  ❷
...
The answer was: 123
```

Coroutines

- Calling **f()** produces a **coroutine**; this means we are allowed to **await** it. The value of the result variable will be **123** when f() completes
-

Example 3-7. Using await on a coroutine

```
async def f():
    await asyncio.sleep(1.0)
    return 123

async def main():
    result = await f()  ❶
    return result
```

Coroutines

- 1) Our coroutine function now handles an **exception**
- 3) Here we throw() the **CancelledError** exception
- 4) As expected, we see our **cancellation message** being printed.

Example 3-9. Coroutine cancellation with CancelledError

```
>>> import asyncio
>>> async def f():
...     try:
...         while True: await asyncio.sleep(0)
...     except asyncio.CancelledError:  ❶
...         print('I was cancelled!')  ❷
...     else:
...         return 111
>>> coro = f()
>>> coro.send(None)
>>> coro.send(None)
>>> coro.throw(asyncio.CancelledError) ❸
I was cancelled!  ❹
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration  ❺
```

Coroutines

Example 3-10. For educational purposes only—don't do this!

```
>>> async def f():
...     try:
...         while True: await asyncio.sleep(0)
...     except asyncio.CancelledError:
...         print('Nope!')
...         while True: await asyncio.sleep(0)  ❶
...     else:
...         return 111
>>> coro = f()
>>> coro.send(None)
>>> coro.throw(asyncio.CancelledError)  ❷
Nope!
>>> coro.send(None)  ❸
```

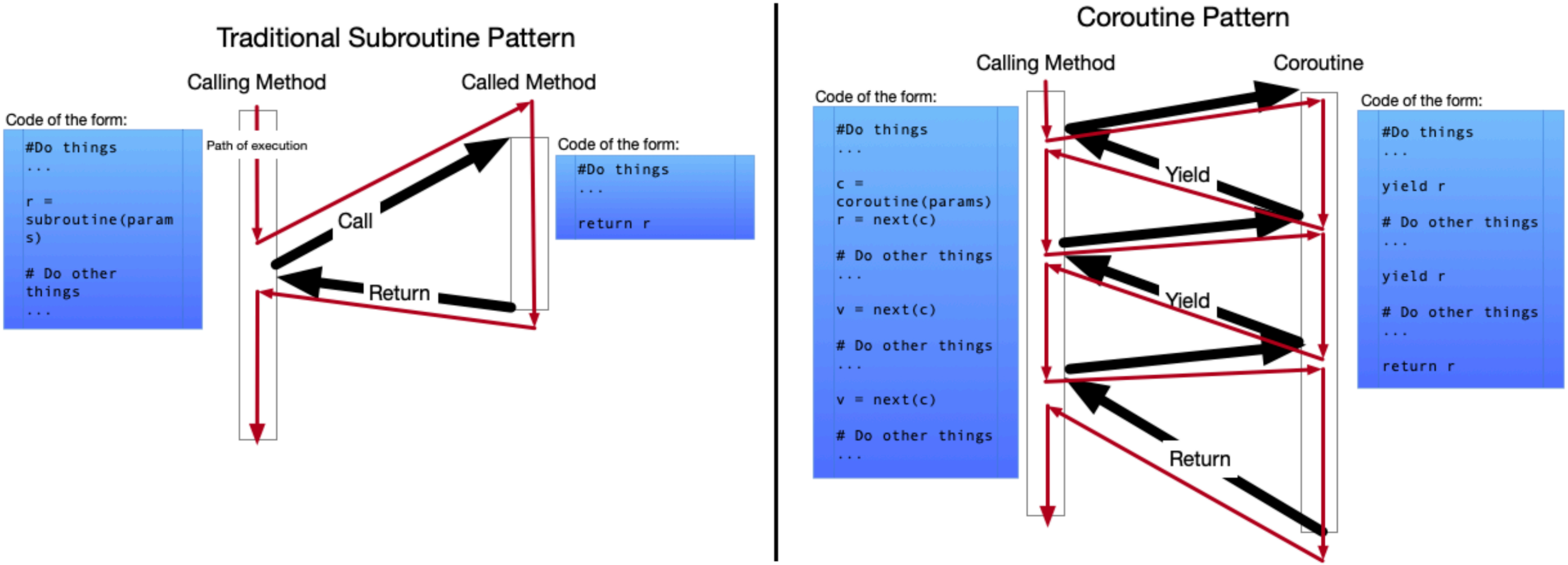
Coroutines

- 1) **Obtain** a loop.
- 2) Run the **coroutine** to **completion**

Example 3-11. Using the event loop to execute coroutines

```
>>> async def f():
...     await asyncio.sleep(0)
...     return 111
>>> loop = asyncio.get_event_loop()  ❶
>>> coro = f()
>>> loop.run_until_complete(coro)  ❷
111
```

Coroutines


Traditional Subroutine Pattern
Calling Method
Called Method
Code of the form:
#Do things
...
r =
subroutine(param
s)
Do other
things
...
Path of execution
Call
Return
Code of the form:
#Do things
...
return r
Coroutine Pattern
Calling Method
Coroutine
Code of the form:
#Do things
...
c =
coroutine(params)
r = next(c)
# Do other things
...
v = next(c)
# Do other things
...
v = next(c)
# Do other things
...
Yield
Yield
Return
Code of the form:
#Do things
...
yield r
Do other things
...
yield r
Do other things
...
return r

Event Loop

- The **event loop** in **asyncio** handles all of the **switching** between **coroutines**, as well as catching those **StopIteration** exceptions
- You can get by **without** ever needing to work with the **event loop directly**
  - Recommended
    asyncio.get_running_loop(), callable from inside the context of a coroutine
  - Discouraged
    asyncio.get_event_loop(), callable from anywhere

Event Loop

Example 3-12. Always getting the same event loop

```
>>> loop = asyncio.get_event_loop()
>>> loop2 = asyncio.get_event_loop()
>>> loop is loop2  ❶
True
```

❶ Both identifiers, `loop` and `loop2`, refer to the same instance.

Event Loop

- The **get_event_loop()** method works only within the **same thread**
- **get_running_loop()** (the recommended method) it can be called only within the context of a coroutine, a task, or a function called of those
- **get_running_loop()** always provides the **current running event loop**

Event Loop

Example 3-14. Creating tasks the modern way

```
import asyncio

async def f():
    # Create some tasks!
    for i in range():
        asyncio.create_task(<some other coro>)
```

Tasks and Futures

Future

- Future represents a future completion state of some activity and is managed by the loop
- A Task is exactly the same, but the specific “activity” is a coroutine
- Future instance as a **toggle** for **completion status.**
- When a Future instance **is created**, the toggle is set to “**not yet completed**,” but at some later time it **will be “completed.”**

Async Context Managers: async with

context managers to make your own code cleaner and easier

- FlowProvider and provider.open_read are **not** coroutine methods
- But return asynchronous context manager object

```
async with FlowProvider(store_url) as provider:
    async with provider.open_read(flow_id, config=config) as reader:
        frames = await reader.read(720, count=480)

        # Do other things using reader
        ...

    # Do other things using provider
    ...

# Do something with frames
...
```

Async Context Managers: async with

context managers to make your own code cleaner and easier

```python
# Perform some IO operations synchronously
with RemoteResource(*some_parameters) as connection:
    connection.send(some_data)
    new_data = connection.recv()

# Perform the same IO operations asynchronously
async with RemoteResource(*some_parameters) as connection:
    await connection.send(some_data)
    new_data = await connection.recv()
```

Async Iterator: async for

- async iterable represents a source of data which can be looped over with an async for loop
- asynchronous iterator derived from the iterable is an asynchronous coroutine method, and its output is awaited

```
async for grain in reader.get_grains():
    # Do something with each grain object
    ...
```

Async Generators

- An async generator can be used as a shorthand method for defining an asynchronous iterator
- the method must contain at least one use of the keyword yield

Asyncio - Library Support

Make HTTP requests with aiohttp

- Python library **requests**, which provides a traditional synchronous http client interface
- **aiohttp** is a library which is designed to make interacting with the http protocol via asyncio
- **aiohttp** contains support for both client and server implementations

Make HTTP requests with aiohttp

```
import aiohttp
import asyncio

async def main():
    async with aiohttp.ClientSession(trust_env=True) as session:
        async with session.get(
            'https://www.bbc.co.uk/rd/projects/cloud-fit-production'
        ) as resp:
            print(resp.status)
            print(await resp.text())

asyncio.run(main())
```

The Rules of Asyncio

- The syntax async def introduces ether a native coroutine or an asynchronous generator
- The keyword await passes function control back to the event loop.

```
async def g():
    # Pause here and come back to g() when f() is ready
    r = await f()
    return r
```