asynccontextmanager for Python 3.7

Sammy Wen
About Me

- Sammy Wen 文祥宇
- [https://gamekingga.com](https://gamekingga.com)
- 🇨🇦 University of Waterloo ECE graduate student
- Sinitic Inc. senior backend engineer
- Use Flask and Autobahn-python to create chatbot platform.
asyncio

❖ Assumption:
  ❖ 24 opponents
  ❖ Player makes each chess move in 5 seconds
  ❖ Opponents each take 55 seconds to make a move
  ❖ All games have 30 pair-moves.

❖ Synchronous
  ❖ One game at a time
  ❖ Each game takes \((5 + 55) \times 30 = 1800\) seconds \(\Rightarrow\) 30 minutes
  ❖ Playing with 24 opponents cost \(30 \times 24\) minutes \(\Rightarrow\) 12 hours

❖ Asynchronous
  ❖ Move from table to table
  ❖ 1 pair-moves for 24 opponents cost \(24 \times 5\) seconds \(\Rightarrow\) 2 minutes
  ❖ 30 pair-moves (a full game) for 24 opponents cost \(30 \times 2\) minutes
    \(\Rightarrow\) 1 hour

asyncio:
- Concurrency in a Single Thread
- Run Tasks (wrapped coroutines) in an event loop

# Asynchronous Python for the Complete Beginner
import asyncio
from autobahn.asyncio.websocket import (
    WebSocketServerProtocol, WebSocketServerFactory
)

class WSConnection(WebSocketServerProtocol):
    ...

    async def process_message_from_ws(self, payload):<=" coroutine
        ...

        # Override onMessage in WebSocketServerProtocol
        async def onMessage(self, payload):
            payload = payload.decode('utf8')
            await self.process_message_from_ws(payload)

    factory = WebSocketServerFactory('ws://127.0.0.1:9000')
factory.protocol = WSConnection
loop = asyncio.get_event_loop()
coro = loop.create_server(factory, '0.0.0.0', 9000)
server_ws = loop.run_until_complete(coro)
Context Manager

- Allow you to allocate and release resources precisely when you want to.
Access database

```python
from sqlalchemy import create_engine

db_settings = {...}
DB = create_engine(db_url, **db_settings)

with DB.connect() as conn:
    conn.execute('SELECT ...')
```

Open and write on a file

```python
with File('demo.txt', 'w') as opened_file:
    opened_file.write('Hola!')
```

Request a webpage

```python
import requests

with requests.Session() as s:
    r = s.get('https://www.google.com/')
    print(r.text)
```

Handle WebSocket connection

```python
import websockets

uri = 'ws://127.0.0.1:9000'
with websockets.connect(uri) as ws:
    while True:
        message = yield from ws.recv()
        yield from ws.send(message)
```

PEP 343 -- The "with" Statement
Context Manager

❖ Allow you to allocate and release resources precisely when you want to.

❖ Could implement it into 2 types:
  ❖ Class
    • Implement __enter__ method and __exit__ method.
  ❖ Generator
    • Add the decorator @contextlib.contextmanager onto the generator’s definition.
asynccontextmanager?

- Asyncio + ContextManager as a Generator
- Asyncio + ContextManager as a Class
Context Manager as a Class

1. Create class instance
2. Call `__init__` method, and the file is opened
3. `with` statement stores the `__exit__` method
4. Call `__enter__` method
5. `self.file_obj` is assigned to `opened_file`
6. Do `.write()`
7. Call the stored `__exit__` method, and the file is closed
   • If the error happens, the exception should be dealt here.
   (The reason why there are 3 arguments on `__exit__`)

Just by defining `__enter__` and `__exit__` methods we can use our new class in a `with` statement.

```python
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)
    def __enter__(self):
        return self.file_obj
    def __exit__(self, type, value, traceback):
        self.file_obj.close()
with File('pycon.txt', 'w') as opened_file:
    opened_file.write('Hello 2019!')
```
Context Manager as a Generator

1. Call `open_file` function

2. The @contextmanager decorator returns the generator wrapped by the _GeneratorContextManager object

3. The _GeneratorContextManager is assigned to the `open_file` function

=> open_file function have related __enter__ method and __exit__ method

4. Call __enter__ method

5. The file object `f` is yielded

6. Do .write()

7. Call __exit__ method

@contextmanager is called with the function name “open_file” as it’s argument

```python
1 import contextlib
2 3 @contextlib.contextmanager
4 def open_file(name):
5     f = open(name, 'w+')
6     yield f
7     f.close()
8
9 with open_file('pycon.txt') as f:
10     f.write('Hello 2019!')
```
In `_GeneratorContextManager` class:

```python
1 def __enter__(self):
2     del self.args, self.kwds, self.func
3     try:
4         return next(self.gen)
5     # self.gen is assigned in `_GeneratorContextManagerBase`
6     except StopIteration:
7         raise RuntimeError("generator didn't yield") from None
8
9 def __exit__(self, type, value, traceback):
10    if type is None:
11        try:
12            next(self.gen)
13        except StopIteration:
14            return False
15        else:
16            raise RuntimeError("generator didn't stop")
17    ...|
```
asynccontextmanager

=> Run/Use multiple resources with contextmanager simultaneously

❖ Decorator `@asynccontextmanager`
  • Asyncio + ContextManager as a Generator

❖ Class with `__aenter__` and `__aexit__` method
  • Asyncio + ContextManager as a Class
asynccontextmanager as a Generator

1. Call `open_session`.

2. The `@asynccontextmanager` decorator returns the generator wrapped by the `_AsyncGeneratorContextManager` object 

`open_file` function have related `__aenter__` method and `__aexit__` method

3. Call `__aenter__` method, which iterates `open_session` generator

4. The session object `s` is yielded

5. Await `session.get()` and Await `text()` 

6. Call `__aexit__` method, which finish the rest part of `open_session` generator

```python
import aiohttp
from contextlib import asynccontextmanager

url = 'https://tw.pycon.org/2019/

async def open_session():
    s = aiohttp.ClientSession()
    yield s
    await s.close()

async def main():
    async with open_session() as session:
        response = await session.get(url)
        text = await response.text()

import asyncio
asyncio.run(main())
```
asynccontextmanager as a Class

1. Create class instance
2. Call `__init__` method, and the session opened
3. `async with` statement stores the `__aexit__` method
4. Call `__aenter__` method
5. `self.obj` is assigned to `session`
6. Await `.session.get()` and Await `.text()`
7. Call the stored `__aexit__` method, and the session is closed
   - If the error happens, the exception should be dealt here. (The reason why there are 3 arguments on `__aexit__`)

```python
1 from aiohttp import ClientSession
2
3 the_url = 'https://tw.pycon.org/2019/'
4
5 class TheSession:
6     def __init__(self):
7         self.obj = None
8     async def __aenter__(self):
9         self.obj = ClientSession()
10        return self.obj
11     async def __aexit__(self, typ, value, tb):
12        await self.obj.close()
13
14 async def main():
15     async with TheSession() as session:
16        response = await session.obj.get(the_url)
17        text = await response.text
18
19 import asyncio
20 asyncio.run(main())
```
Live Demo
Learn More

- **contextlib**
  - ExitStack & AsyncExitStack (new in 3.7)
    - [https://www.rath.org/on-the-beauty-of-pythons-exitstack.html](https://www.rath.org/on-the-beauty-of-pythons-exitstack.html)
  - nullcontext (new in 3.7)

- **other new features on asyncio in 3.7**
  - async and await become keywords
  - asyncio.run()
  - Support for Context Variable (PEP-567, new in 3.7)
  - Support for loop.call_soon()/loop.call_later()/loop.call_at()
  - Shortcuts (asyncio.create_task(), asyncio.current_task(), asyncio.all_task(), etc.)
  - Network/Socket/Streaming (improvement in BufferedProtocol, asyncio.Server, etc.)
Hiring Full-stack Engineer (Python Flask + Vue.js)

JD Link: https://secure.collage.co/jobs/sinitic/10484
References

- https://www.python.org/dev/peps/pep-0343/
- https://docs.python.org/3/library/contextlib.html
- https://docs.python.org/3/library/asyncio-task.html
- https://docs.python.org/3/whatsnew/3.7.html#whatsnew37-asyncio
- https://stackoverflow.com/questions/37433157/asynchronous-context-manager