# asyncio

#### asyncio

- Python offers higher-level support for network connections
- Hides many lower-level details from the programmer
- Uses coroutines, which is a form of co-operative multitasking
- Coroutines are declared using the keyword *async*
- Coroutines can await the result from/yield to another coroutine
- See https://realpython.com/async-io-python/ for supplemental reading

### Threads vs Coroutines (asyncs)

```
#!/usr/bin/python3.11
from threading import Thread, get_ident

def function(thread_id: int) -> None:
    while True:
        print(thread_id, get_ident())

def main() -> None:
    for i in range(2):
        Thread(target=function, args=(i,)).start()

main()
```

### Threads vs Coroutines (asyncs)

```
#!/usr/bin/python3.11
from threading import get ident # threading only used for get ident call
from asyncio import create task, run, sleep
async def function(thread id: int) -> None:
    while True:
        print(thread id, get ident())
        await sleep(0) # or won't switch tasks
async def main() -> None:
    tasks = []
    for i in range(2):
        t = create task(function(i))
        tasks.append(t)
    for t in tasks:
        await t
run(main())
```

#### asyncio Server

```
#!/usr/bin/python3.11
from asyncio import run, start server, StreamReader, StreamWriter
async def echo(reader: StreamReader, writer: StreamWriter) -> None:
    data = await reader.readline()
    message = data.decode()
    addr = writer.get extra info('peername')
    print(f"Received {message} from {addr}")
    writer.write(data) # starts to write the data to the stream
    await writer.drain() # waits until the data is written
    writer.close()
    await writer.wait closed()
async def main() -> None:
    server = await start server(echo, '127.0.0.1', 12345)
    await server.serve forever() # without this, the program
terminates
run(main())
```

#### Server Notes

- await asyncio.start\_server(echo, '127.0.0.1', 12345) starts a new instance of a call to echo; this looks like multithreading, but in reality, there is a single thread jumping between different instances of echo
- drain waits until it is safe to write to the stream again (or safe to close it); implements flow control, to avoid buffer overflows

#### asyncio Client

```
#!/usr/bin/python3.11
from asyncio import open connection, run
from sys import argv
async def client(message: str) -> None:
    reader, writer = await open connection('127.0.0.1', 12345)
    writer.write(message.encode() + b'\n')
    data = await reader.readline()
    print(f'Received: {data.decode("utf-8")}')
    writer.close() # reader has no close() function
    await writer.wait closed() # wait until writer completes close()
if len(argv) != 2:
    print(f'{argv[0]} needs 1 argument to transmit')
    exit(-1)
run(client(argv[1]))
```

#### Client Notes

- asyncio.open\_connection('127.0.0.1', 12345) can throw an exception (e.g., ConnectionRefusedError)
- On success, returns a tuple containing a reader for reading from the stream, and a writer for writing to the stream
- No drain is needed on the client end because it writes only once, then waits until data comes back from the server (which means that all the client's data has been written)

#### Client Notes

- Calling recv(1) is actually quite inefficient, since it can involve a context switch
- asyncio provides convenient functions (quoting from python.org):
  - coroutine read(n=-1)
     Read up to n bytes. If n is not provided, or set to -1, read until EOF and return all read bytes. If EOF was received and the internal buffer is empty, return an empty bytes object.
  - coroutine readline()
     Read one line, where "line" is a sequence of bytes ending
     with \n. If EOF is received and \n was not found, the method
     returns partially read data. If EOF is received and the
     internal buffer is empty, return an empty bytes object.

#### Client Notes

- coroutine readexactly(n)
   Read exactly n bytes. Raise an IncompleteReadError if EOF is reached before n can be read. Use the IncompleteReadError.partial attribute to get the partially read data.
- coroutine readuntil(separator=b'\n')
  Read data from the stream until separator is found. On success, the data and separator will be removed from the internal buffer (consumed). Returned data will include the separator at the end. If the amount of data read exceeds the configured stream limit, a LimitOverrunError exception is raised, and the data is left in the internal buffer and can be read again. If EOF is reached before the complete separator is found, an IncompleteReadError exception is raised, and the internal buffer is reset. The IncompleteReadError.partial attribute may contain a portion of the separator.

#### **Exercises**

- On your own:
  - Work on the questions in the asyncio section of *Practice Questions and Solutions*

#### Another asyncio Server

```
#!/usr/bin/python3.11
from asyncio import run, start server, StreamReader, StreamWriter
cnx = 0
async def echo(reader: StreamReader, writer: StreamWriter) -> None:
    global cnx
    try:
        local id = cnx
        cnx += 1
        while True:
            data = await reader.readline()
            if data == b'':
                break
            message = data.decode()
            print(f"{local id} {data}")
            writer.write(data) # starts to write the data to the stream
            await writer.drain() # waits until the data is written
        writer.close()
        await writer.wait closed()
    except Exception:
        pass
```

## Another asyncio Server

```
async def main() -> None:
    server = await start_server(echo, '127.0.0.1', 12345)
    await server.serve_forever() # without this, the program terminates
run(main())
```

### Another asyncio Client

- Let us write a client that:
  - gets the number of clients c and number of repetitions r from the command line
  - starts c clients, each of which connects to a server listening on localhost port 12345 and sends a message to the server r times
  - messages should be sent in an interleaved fashion, not in sequential fashion

#### Approach 1

```
#!/usr/bin/python3.11
from asyncio import open connection, run, sleep
from sys import argv
HOST = 'localhost'
PORT = 12345
async def send message(client id: int, num reps: int) -> None:
    for i in range(num reps):
        reader, writer = await open connection(HOST, PORT)
        writer.write(('This is ' + str(client id) + '\n').encode())
        await writer.drain()
        writer.close()
        await writer.wait closed()
        await sleep(0)
async def main(num clients: int, num reps: int) -> None:
    for i in range(num clients):
        await send message(i, num reps)
```

#### Approach 1

```
if len(argv) != 3:
    print(argv[0], '<Number of clients> <Number of repetitions>')
    exit(-1)

try:
    run(main(int(argv[1]), int(argv[2])))
except Exception as e:
    print(e)
```

#### Output 1

```
0 b'This is 0'
1 b'This is 0'
2 b'This is 1'
3 b'This is 1'
4 b'This is 2'
5 b'This is 2'
6 b'This is 3'
7 b'This is 3'
```

- Note that the connection numbers change and that the messages are not interleaved
- The client is not taking advantage of asynchronous transmissions; it is still largely operating sequentially

#### Approach 2

```
async def main(num_clients: int, num_reps: int) -> None:
    tasks = []
    for i in range(num_clients):
        tasks.append(create_task(send_message(i, num_reps)))

for t in tasks:
    await t
```

- server = await asyncio.start\_server(echo, '127.0.0.1', 12345) starts echo as a task so that it becomes possible to switch between (interleave) different running instances of echo
- On the client side, we have to create these tasks explicitly

#### Output 2

```
8 b'This is 3'
9 b'This is 0'
10 b'This is 1'
11 b'This is 2'
12 b'This is 2'
13 b'This is 0'
15 b'This is 1'
14 b'This is 3'
```

- Note that the messages are now interleaved
- It's more efficient if we re-use the existing connections

#### Approach 3

```
async def send_message(client_id: int, num_reps: int) -> None:
    reader, writer = await open_connection(HOST, PORT)
    for i in range(num_reps):
        writer.write(('This is ' + str(client_id) + '\n').encode())
        await writer.drain()
        await sleep(0)
    writer.close()
    await writer.wait_closed()
```

### Output 3

```
16 b'This is 2'
17 b'This is 3'
16 b'This is 2'
18 b'This is 1'
17 b'This is 3'
19 b'This is 0'
18 b'This is 0'
19 b'This is 0'
```

• Note that the connection numbers repeat as well

# Debugging Strategies

## Echo Client/Server Sample Run

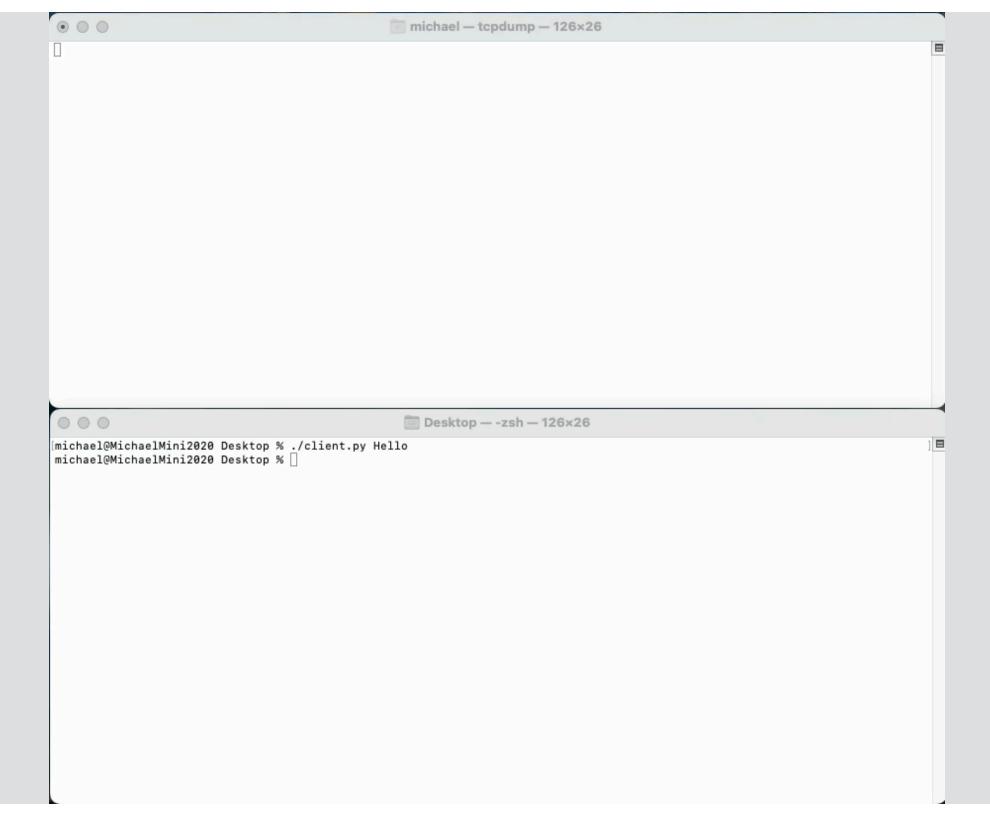


### **Debugging Strategies**

- print out values before (or after) using them
- python3 -m trace -t --ignore-dir=/usr client.py
   macOS: python3 -m trace -t --ignore-dir=/Applications client.py
- tcpdump -i lo -A port 12345 (Divide and conquer: Client or Server?)
- lsof -i -P

#### Debugging Strategies

- print out values before (or after) using them
- python3 -m trace -t --ignore-dir=/usr client.py
   macOS: python3 -m trace -t --ignore-dir=/Applications client.py
- tcpdump -i lo -A port 12345 (Divide and conquer: Client or Server?)
- lsof -i -P



### Observations & Hypotheses

- No traffic detected on port 12345
- Likely a client issue

### **Debugging Strategies**

- print out values before (or after) using them
- python3 -m trace -t --ignore-dir=/usr client.py macOS: python3 -m trace -t --ignore-dir=/Applications client.py
- tcpdump -i lo -A port 12345 (Divide and conquer: Client or Server?)
- lsof -i -P

#### trace



#### Observations & Hypotheses

- Client is opening a connection
- Looks like an invalid value being used by the client
- The other issue is that we don't print out the exception

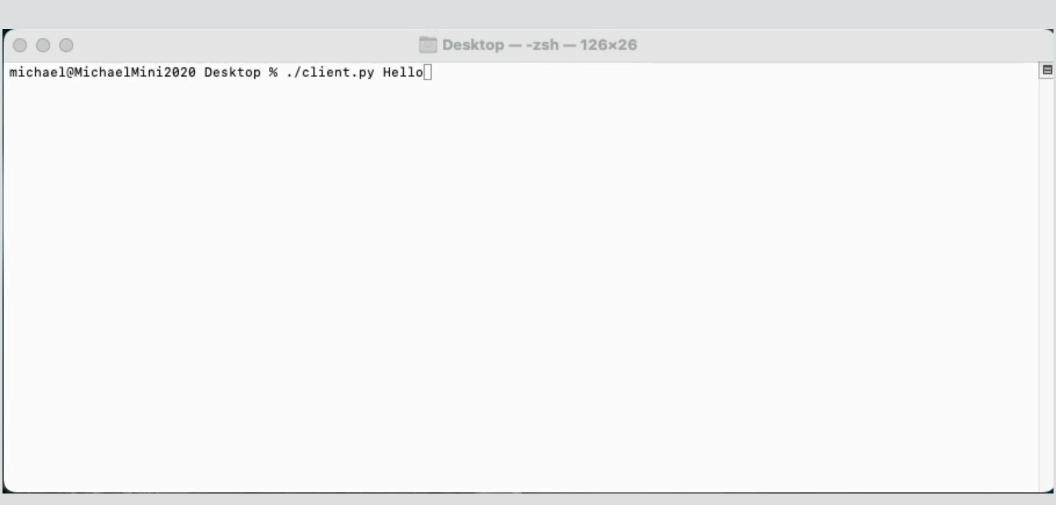
#### Correct the Code

```
#!/usr/bin/python3
import asyncio
import sys
async def client(message):
    try:
        reader, writer = await asyncio.open connection('127.0.0.1', 12435)
        writer.write(message.encode('utf-8') + b'\n')
        data = await reader.readline() # more on this on the next slides
        print(f'Received: {data.decode("utf-8")}')
        writer.close() # reader has no close() function
        await writer.wait closed() # wait until writer completes close()
    except Exception as details:
           pass
if len(sys.argv) != 2:
    print(f'{sys.argv[0]} needs a message to transmit')
    sys.exit(-1)
asyncio.run(client(sys.argv[1]))
```

#### Correct the Code

```
#!/usr/bin/python3
import asyncio
import sys
async def client(message):
    try:
        reader, writer = await asyncio.open connection('127.0.0.1', 12345)
        writer.write(message.encode('utf-8') + b'\n')
        data = await reader.readline() # more on this on the next slides
        print(f'Received: {data.decode("utf-8")}')
        writer.close() # reader has no close() function
        await writer.wait closed() # wait until writer completes close()
    except Exception as details:
           print(details)
if len(sys.argv) != 2:
    print(f'{sys.argv[0]} needs a message to transmit')
    sys.exit(-1)
asyncio.run(client(sys.argv[1]))
```

#### Test the Correction



# Observations & Hypotheses

- Connection failed
- Likely a server issue

### **Debugging Strategies**

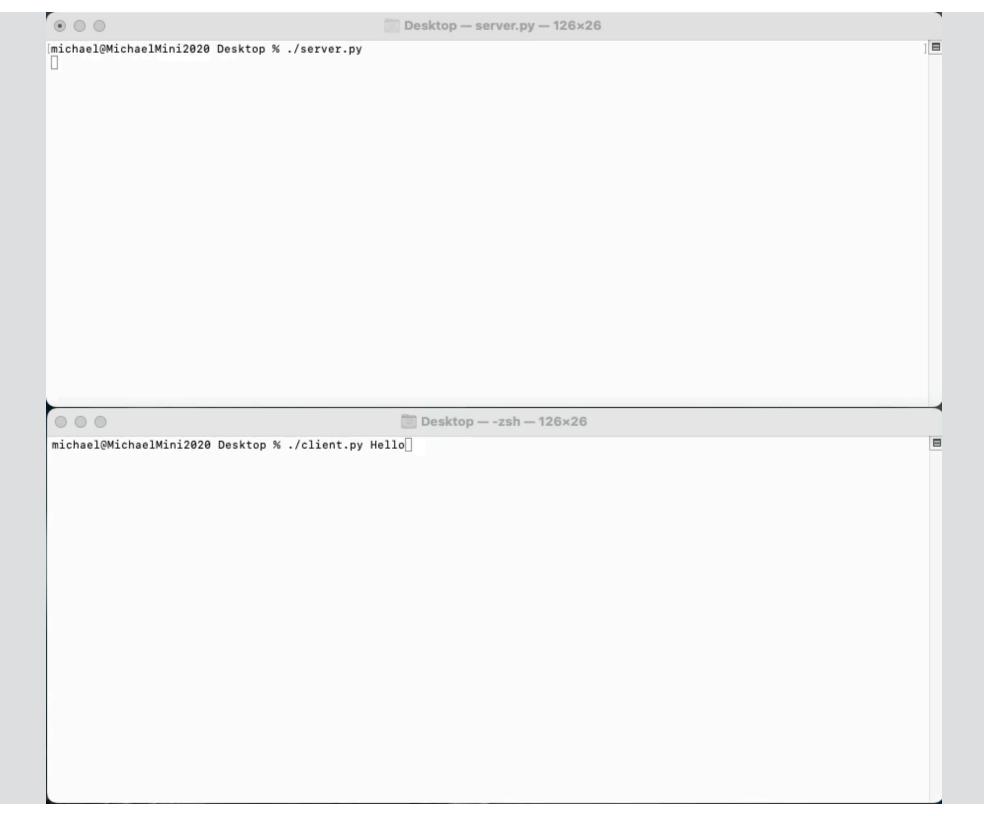
- print out values before (or after) using them
- python3 -m trace -t --ignore-dir=/usr client.py
   macOS: python3 -m trace -t --ignore-dir=/Applications client.py
- tcpdump -i lo -A port 12345 (Divide and conquer: Client or Server?)
- lsof -i -P

### lsof

Python 4357 michael 6u IPv4 0x57f2102cbcf70821 0t0 TCP localhost:12346 (LISTEN)

### Observations & Hypotheses

- Incorrect port was used
- Correct the issue and confirm it has been resolved

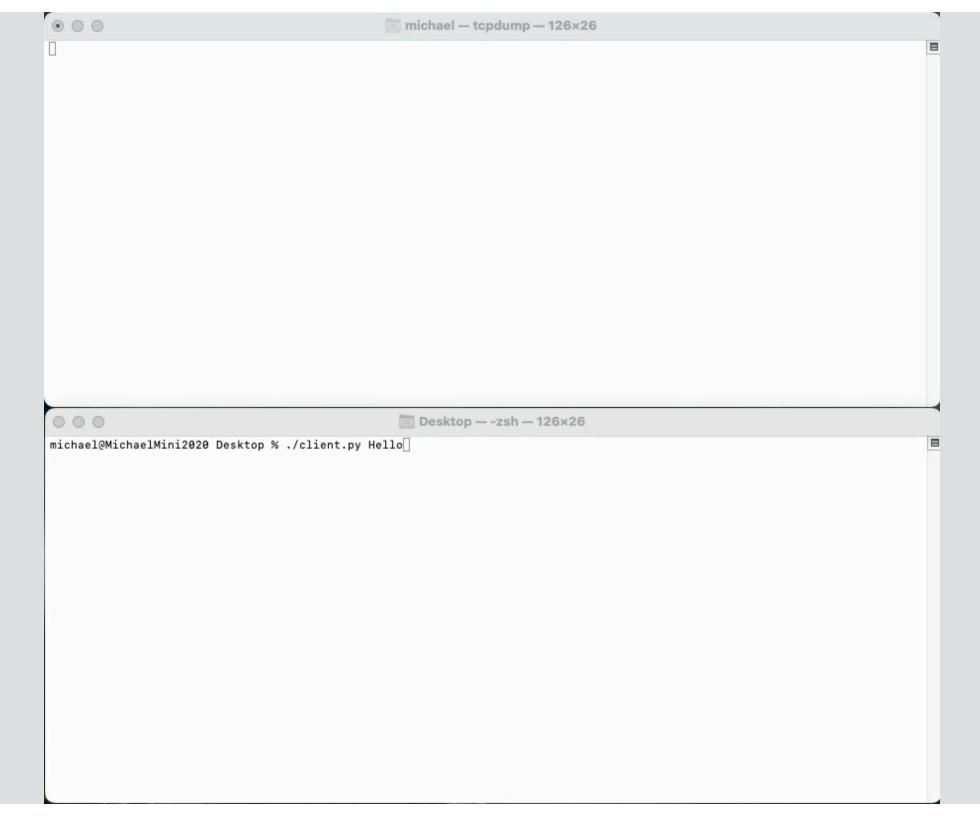


# Observations & Hypotheses

- Client and server are hanging
- Likely a communication issue

### **Debugging Strategies**

- print out values before (or after) using them
- python3 -m trace -t --ignore-dir=/usr client.py
   macOS: python3 -m trace -t --ignore-dir=/Applications client.py
- tcpdump -i lo -A port 12345 (Divide and conquer: Client or Server?)
- lsof -i -P



#### Observations & Hypotheses

- Three-way handshake completes
- Hello is transmitted by the client (Line 5)
- No reply is sent by the server
- Correct the issue and confirm it has been resolved

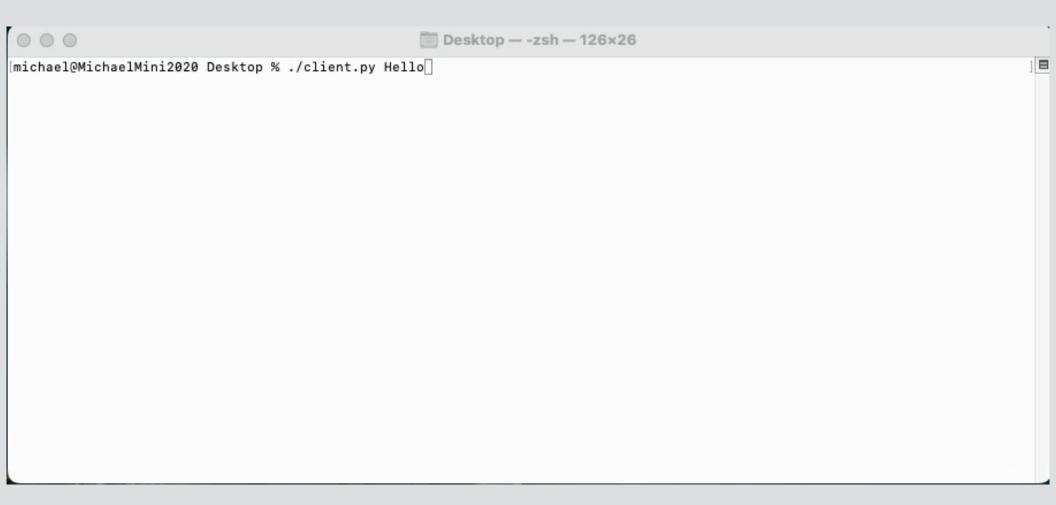
#### Correct the Code

```
#!/usr/bin/python3
import asyncio
async def read(reader):
    return await reader.readline()
async def write(writer, data):
    writer.write(data)
    await writer.drain()
    writer.close()
    await writer.wait_closed()
async def echo(reader, writer):
    message = await read(reader)
async def main():
    server = await asyncio.start_server(echo, '127.0.0.1', 12345)
    await server.serve_forever()
asyncio.run(main())
```

#### Correct the Code

```
#!/usr/bin/python3
import asyncio
async def read(reader):
    return await reader.readline()
async def write(writer, data):
    writer.write(data)
    await writer.drain()
    writer.close()
    await writer.wait_closed()
async def echo(reader, writer):
    message = await read(reader)
    await write(writer, message)
async def main():
    server = await asyncio.start_server(echo, '127.0.0.1', 12345)
    await server.serve forever()
asyncio.run(main())
```

#### Test the Correction



#### Lab 6

 Modify the game so that it is using asyncio instead of threaded sockets

### Key Skills

- Be able to program network clients and servers using asyncio
- Employ various debugging strategies