Programming Language Support for Threading
Most modern programming languages provide language-level support for threading:

```python
import asyncio
async def fibonacci(x, tid):
    # Base Cases:
    if x == 0: return 0
    if x == 1: return 1
    print(f"{tid}: Calculating fibonacci({x})...")
    await asyncio.sleep(0.1)
    fx_minus1 = await fibonacci(x - 1, tid)
    fx_minus2 = await fibonacci(x - 2, tid)
    return fx_minus1 + fx_minus2
```

The `async` keyword wraps the function (formally called a coroutine) as an `Future` object.

- A `Future` object:

A `Future` has three states:

1. Unfulfilled:
2. Fulfilled:
3. Failed:

As a procedural programming language, the `await` keyword exists to synchronize your code based on the result of a `Future`:

```python
async def main():
    r = await asyncio.gather(
        fibonacci(15, "A"),
        fibonacci(14, "B"),
        fibonacci(13, "C"),
    )
    print(r)
```

You can “race” all multiple `Future` objects:

```python
asyncio.run(main())
```

Q: What output do we get?

Since every `async` function is just `Future`, you must `asyncio.run` your first one `async` function (often a function called `main`):

```python
asyncio.run(main())
```

Otherwise: Python does nothing (but does provide a warning):
Multithreading in Python
Python is multi-threaded, but ___________________________

25/countup.py

```python
import asyncio

countup.py

ct = 0
THREAD_COUNT_AMOUNT = 1000000

async def countup():
    global ct
    for i in range(THREAD_COUNT_AMOUNT):
        ct += 1
async def main():
    await asyncio.gather(countup(), countup(), countup(),)
    print(ct)
asyncio.run(main())
```

Q: When we did this in C, what happened?

Q: What do we expect to happen in Python?

Q: What is the difference between `countup` and `countup2`?

Q: What happens when we run this code with :15 commented out?

....and if it's not commented out?

Q: What can we learn about how Python handles threading verses C?