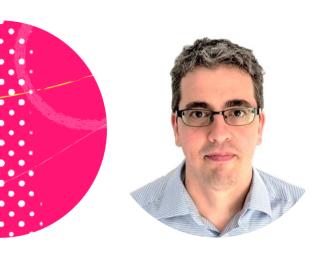
Using More Threads



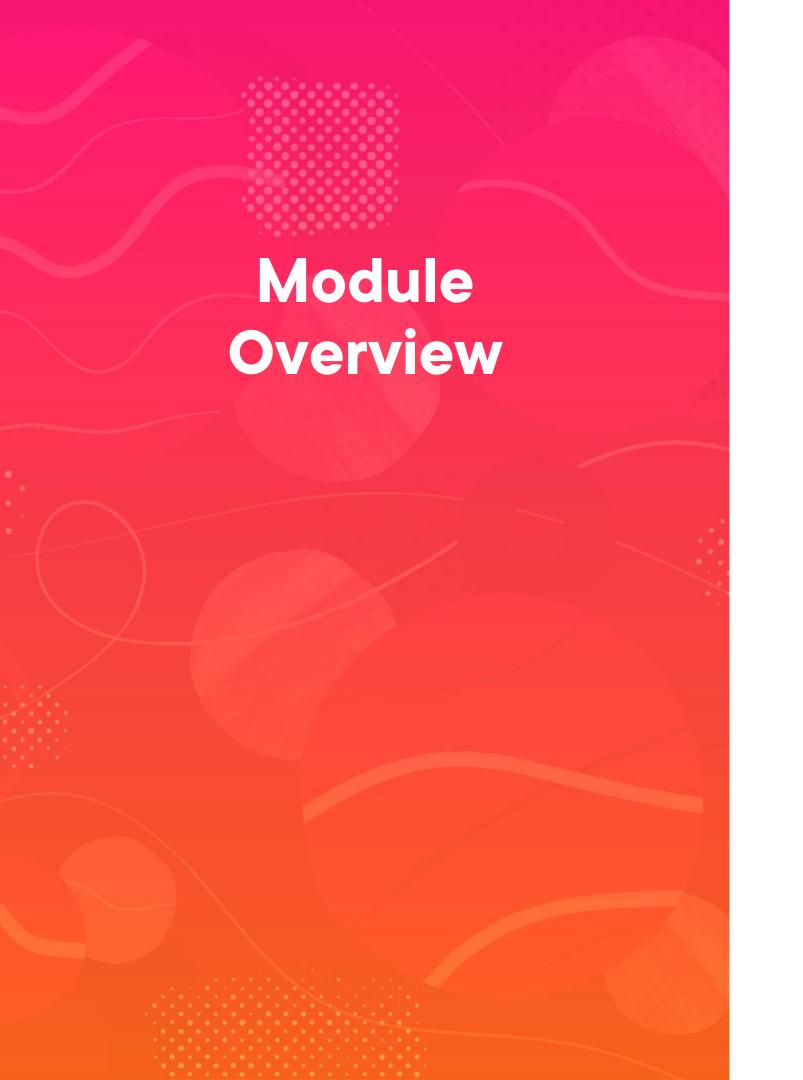
Dan Tofan

Software Engineer, PhD

@dan_tofan | www.programmingwithdan.com







What are threads?

Challenges of working with threads

When to use multithreading

Threads

Separate execution flows

Part of a process

Concurrency

Improve performance



Threads vs Processes

Threads VS Processes

Lightweight

Shared memory

High potential for bugs

GIL constraint

Heavyweight

Separate memory

Low potential for bugs

No GIL constraint

```
class OrderProcessing(threading.Thread):
    def run(self):
        print(f"Processing")
thread = OrderProcessing()
thread.start()
def process():
    print(f"Processing...")
t = threading.Thread(target=process)
t.start()
```

import threading

◄ Import the threading module

■ Create a subclass of Thread

◄ Create new instance

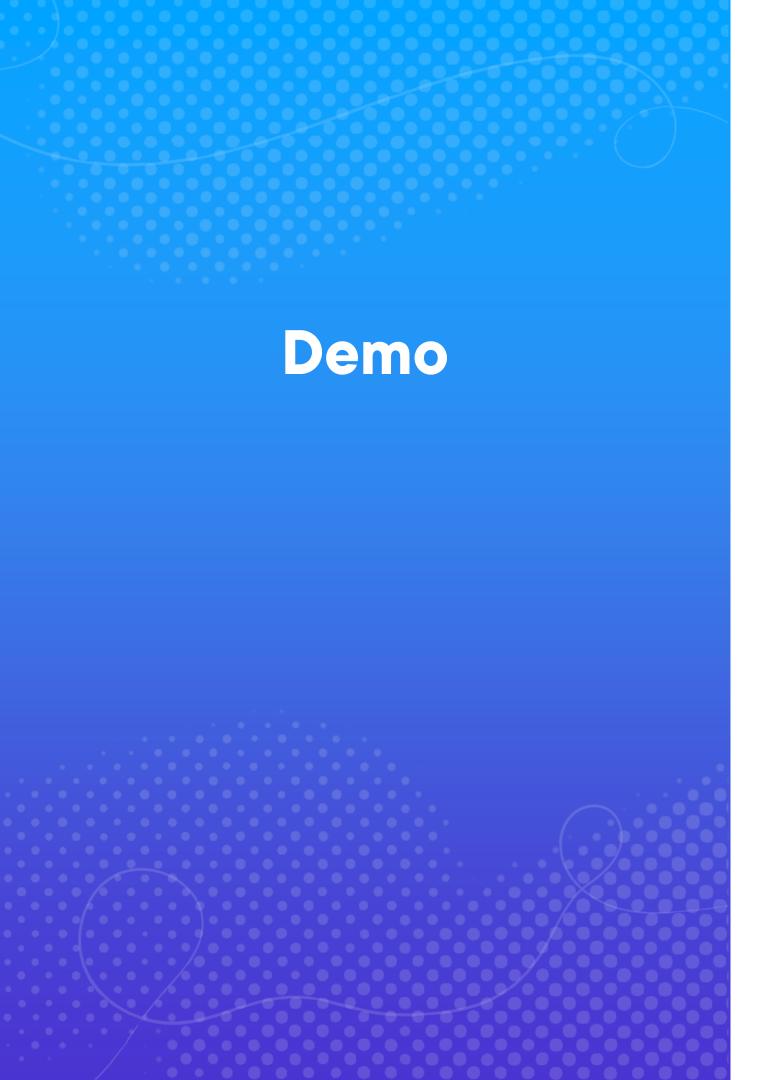
◄ Start the new thread

◄ Python function

◄ Create new thread

■ Start the new thread





Process orders

Use the threading module

Create two threads

Challenges of Working with Threads

Synchronizing threads

Troubleshooting

GIL



Synchronizing Threads

Race conditions Deadlocks Starvation Livelocks



Troubleshooting

Reproducing

Debugging

Finding the root cause of issues



Global Interpreter Lock

Only one thread is running

The sleep() function releases the GIL

Most impact on CPU-intensive tasks

Demo

Process orders

Check performance impact of GIL

- One thread
- Two threads

How to Synchronize Threads

Locks

Semaphores

Condition variables



When to Use Threads

Use threads

Tasks that wait for external events

Blocking I/O

Simple logic

VS

Avoid threads

No waiting for external events

CPU-intensive tasks

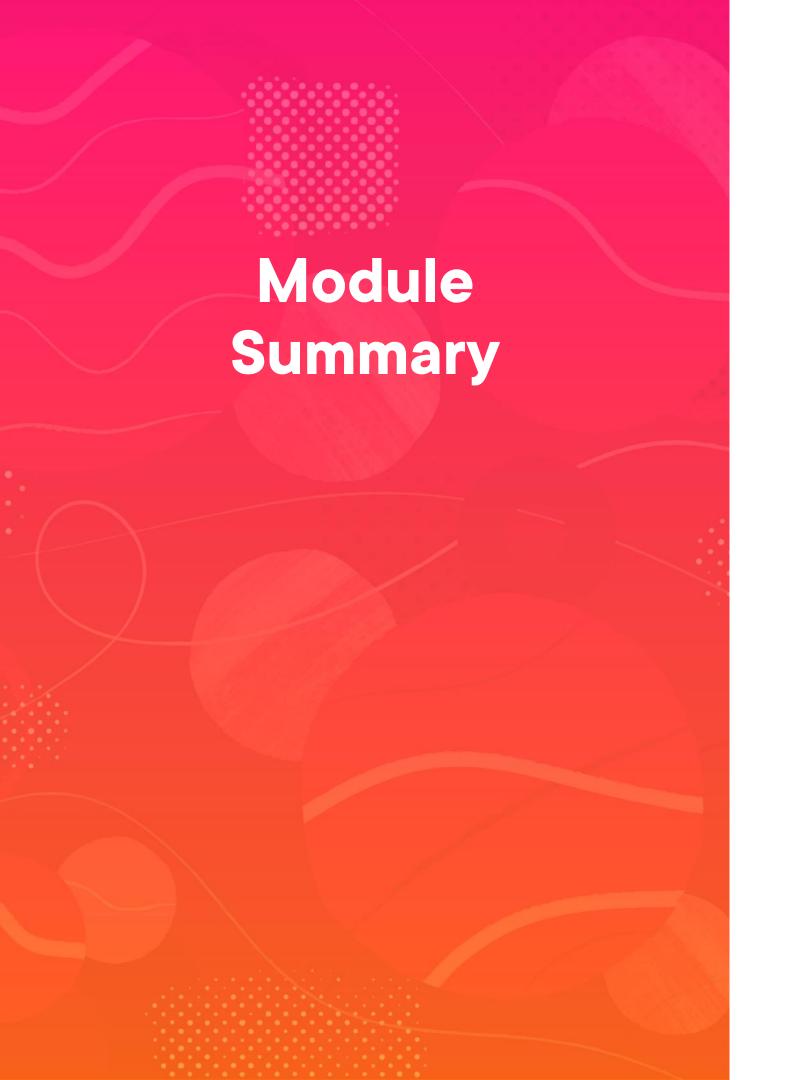
Complex logic

Demo

Download from URL

Compare performance

- Single thread
- Multiple threads



What are threads?

Challenges of working with threads

When to use multithreading

Up Next:

Using Asynchronous Code

