Page 1: Threads in Python

What is a Thread in Python?

A **thread** is the smallest unit of a program that can execute independently. In Python, threads are used to achieve concurrency (tasks running seemingly at the same time).

Key Characteristics:

- Threads share the same memory space.
- Useful for I/O-bound tasks like web scraping, file operations, or API calls.
- Managed by the **Global Interpreter Lock (GIL)** in Python.

What is the GIL?

The **Global Interpreter Lock (GIL)** is a mutex that protects access to Python objects. It ensures that only one thread executes Python bytecode at a time.

This means **no true parallelism** for CPU-bound tasks but works well for I/O-bound tasks.

Example Code: Threads in Action

```
import threading
import time

def task(name):
   print(f"Thread {name} starting")
   time.sleep(2)  # Simulate a time-consuming task (e.g., I/O)
   print(f"Thread {name} finished")

# Create and start multiple threads
threads = [threading.Thread(target=task, args=(i,)) for i in range(3)]

for t in threads:
   t.start()

for t in threads:
   t.join()
# Output:
# Thread 0 starting
# Thread 1 starting
# Thread 2 starting
# (After ~2 seconds)
# Thread 0 finished
# Thread 1 finished
# Thread 2 finishe
```

Page 2: Processes in Python

What is a Process in Python?

A **process** is an independent unit of execution that runs in its own memory space. Each process:

- Has its own Python interpreter.
- Can achieve **true parallelism** on multi-core CPUs.
- Does not share memory with other processes.

When to Use Processes?

Processes are ideal for **CPU-bound tasks** (e.g., image processing, mathematical computations).

Example Code: Processes in Action

```python
from multiprocessing import Process
import os

def task(name):
    print(f"Process {name} running on PID: {os.getpid()}")

# Create and start multiple processes
processes = [Process(target=task, args=(i,)) for i in range(3)]

for p in processes:
    p.start()

for p in processes:
    p.join()

# Output:
# Process 0 running on PID: 12345
# Process 1 running on PID: 12346
# Process 2 running on PID: 12347
```

Page 3: Multithreading vs. Multiprocessing

Threads

- Shared memory space.
- Concurrency but no true parallelism (due to the GIL).
- Lightweight and faster to create.
- Ideal for **I/O-bound tasks**.

Processes

- Separate memory space.
- True parallelism on multi-core CPUs.
- Heavier and slower to create compared to threads.
- Ideal for **CPU-bound tasks**.

Comparing Use Cases

Task Type	Recommended Approach
Web scraping	Multithreading
Image processing	Multiprocessing
File operations	Multithreading
Machine Learning	Multiprocessing

Page 4: ThreadPoolExecutor and ProcessPoolExecutor

Simplifying Multithreading and Multiprocessing

Python's `concurrent.futures` module provides:

- **ThreadPoolExecutor**: Manages a pool of threads.
- **ProcessPoolExecutor**: Manages a pool of processes.

This abstracts away the complexity of managing threads or processes manually.

Example Code: ThreadPoolExecutor

```
from concurrent.futures import ThreadPoolExecutor
import time
def task(name):
    print(f"Thread {name} starting")
    time.sleep(2)  # Simulate I/O
    print(f"Thread {name} finished")

with ThreadPoolExecutor(max_workers=3) as executor:
    executor.map(task, [0, 1, 2])
# Output:
# Thread 0 starting
# Thread 1 starting
# Thread 2 starting
# (After ~2 seconds)
# Thread 0 finished
# Thread 1 finished
# Thread 2 finished
```

Example Code: ProcessPoolExecutor

```
from concurrent.futures import ProcessPoolExecutor
import os

def task(name):
    print(f"Process {name} running on PID: {os.getpid()}")

with ProcessPoolExecutor(max_workers=3) as executor:
    executor.map(task, [0, 1, 2])

# Output:
# Process 0 running on PID: 12345
# Process 1 running on PID: 12346
# Process 2 running on PID: 12347
```

Page 5: Key Takeaways

1. **Threads** are best for **I/O-bound tasks**.
2. **Processes** are best for **CPU-bound tasks**.
3. Use **ThreadPoolExecutor** or **ProcessPoolExecutor** to simplify code.
4. Understand the **GIL** to design better solutions in Python.

Choose the right approach based on your task's requirements!