Advance Review

- Parallel Programming
- Tests Driven Development

Parallel Programming

https://www.youtube.com/watch?v=X7vBbelRXn0

High performance programming

Multiprocessing

Pros

- Separate memory space
- Code is usually straightforward
- Takes advantage of multiple CPUs & cores
- Avoids GIL limitations for cPython
- Eliminates most needs for synchronization primitives unless if you use shared memory (instead, it's more of a communication model for IPC)
- Child processes are interruptible/killable
- Python `multiprocessing` module includes useful abstractions with an interface much like `threading.Thread`
- A must with cPython for CPU-bound processing

Cons

- IPC a little more complicated with more overhead (communication model vs. shared memory/objects)
- Larger memory footprint

Example

```
from __future__ import annotations


import os.path
import time
from multiprocessing import Pool


import numpy as np
import scipy.io.wavfile
```

```
def gen_fake_data(filenames):
    print("generating fake data")
    try:
        os.mkdir("sounds")
    except FileExistsError:
        pass

    for filename in filenames:  # homework: convert this loop to pool too!
        if not os.path.exists(filename):
            print(f"creating {filename}")
            gen_wav_file(filename, frequency=440, duration=60.0 * 4)


def gen_wav_file(filename: str, frequency: float, duration: float):
    samplerate = 44100
    t = np.linspace(0., duration, int(duration * samplerate))
    data = np.sin(2. * np.pi * frequency * t) * 0.0
    scipy.io.wavfile.write(filename, samplerate, data.astype(np.float32))


def etl(filename: str) -> tuple[str, float]:
    # extract
    start_t = time.perf_counter()
    samplerate, data = scipy.io.wavfile.read(filename)

    # do some transform
    eps = .1
    data += np.random.normal(scale=eps, size=len(data))
    data = np.clip(data, -1.0, 1.0)

    # load (store new form)
    new_filename = filename.removesuffix(".wav") + "-transformed.wav"
    scipy.io.wavfile.write(new_filename, samplerate, data)
    end_t = time.perf_counter()

    return filename, end_t - start_t


def etl_demo():
```

```
    filenames = [f"sounds/example{n}.wav" for n in range(24)]
    gen_fake_data(filenames)
    start_t = time.perf_counter()

    print("starting etl")
    with Pool() as pool:
        results = pool.map(etl, filenames)

        for filename, duration in results:
            print(f"{filename} completed in {duration:.2f}s")

    end_t = time.perf_counter()
    total_duration = end_t - start_t
    print(f"etl took {total_duration:.2f}s total")


def run_normal(items, do_work):
    print("running normally on 1 cpu")
    start_t = time.perf_counter()
    results = list(map(do_work, items))
    end_t = time.perf_counter()
    wall_duration = end_t - start_t
    print(f"it took: {wall_duration:.2f}s")
    return results


def run_with_mp_map(items, do_work, processes=None, chunksize=None):
    print(f"running using multiprocessing with {processes=}, {chunksize=}")
    start_t = time.perf_counter()
    with Pool(processes=processes) as pool:
        results = pool.imap(do_work, items, chunksize=chunksize)
    end_t = time.perf_counter()
    wall_duration = end_t - start_t
    print(f"it took: {wall_duration:.2f}s")
    return results
```

Threading

Pros

- Lightweight - low memory footprint
- Shared memory - makes access to state from another context easier
- Allows you to easily make responsive UIs
- cPython C extension modules that properly release the GIL will run in parallel
- Great option for I/O-bound applications

Cons

- cPython - subject to the GIL
- Not interruptible/killable
- If not following a command queue/message pump model (using the `Queue` module), then manual use of synchronization primitives become a necessity (decisions are needed for the granularity of locking)
- Code is usually harder to understand and to get right - the potential for race conditions increases dramatically

Asyncio

The Global Interpreter Lock (GIL)

The Global Interpreter Lock (GIL) is a mechanism used in Python to ensure that only one thread executes Python bytecode at a time in a single Python process. This means that, despite having multiple threads, only one thread can execute Python code at any given moment. The GIL prevents race conditions and ensures thread safety by serializing access to Python objects, which helps simplify the implementation of the Python interpreter and makes it easier to write thread-safe Python code.

Key points about the GIL:

1. **Concurrency vs. Parallelism:** While threads can run concurrently (appear to run simultaneously), they do not run in parallel on multiple CPU cores due to the GIL. This means that multithreading in Python may not always lead to performance improvements for CPU-bound tasks, as only one thread can execute Python bytecode at a time.
2. **Impact on I/O-bound Tasks:** The GIL has less impact on I/O-bound tasks (tasks that spend a lot of time waiting for input/output operations, such as network requests or file I/O), as threads can overlap their waiting times.
3. **Impact on CPU-bound Tasks:** For CPU-bound tasks (tasks that require a lot of CPU computation), the GIL can become a bottleneck, limiting the performance gains from using multithreading.

4. **Circumventing the GIL:** Python's multiprocessing module allows bypassing the GIL by spawning multiple processes instead of threads. Each process has its own Python interpreter and memory space, enabling true parallelism across multiple CPU cores.
5. **Trade-offs:** While the GIL simplifies memory management and ensures thread safety, it can limit the scalability of multithreaded Python programs, especially on multi-core systems. Developers need to consider the trade-offs between simplicity and performance when choosing between threading and multiprocessing in Python.

Overall, the GIL is a characteristic feature of Python's CPython interpreter and has implications for multithreading and parallelism in Python programs. It's important for developers to understand its behavior and its impact on the performance of their Python applications.

Tests Driven Development

```
from dataclasses import dataclass, field
from enum import Enum



class OrderStatus(Enum):
    OPEN = "open"
    PAID = "paid"



@dataclass
class LineItem:
    name: str
    price: int
    quantity: int = 1

    @property
    def total(self) -> int:
        return self.price * self.quantity



@dataclass
class Order:
    line_items: list[LineItem] = field(default_factory=list)
    status: OrderStatus = OrderStatus.OPEN

    @property
    def total(self) -> int:
        return sum(item.total for item in self.line_items)

    def pay(self) -> None:
        self.status = OrderStatus.PAID
```

```
from pay.order import LineItem



def test_line_item_total() -> None:
    line_item = LineItem(name="Test", price=100)
    assert line_item.total == 100



def test_line_item_total_quantity() -> None:
    line_item = LineItem(name="Test", price=100, quantity=2)
    assert line_item.total == 200
```