# Terms

- CPU bound - Code that does most of the work through the CPU e.g. crunching numbers, transforming data, etc.
- I/O bound - Code that reads or writes data through the input/ouput interface e.g. making calls to an API, reading from disk, etc.
- Thread - A sequence of instructions within a process that can be executed independently.
- Process - An instance of a program running on a computer, encapsulating its own memory space and resources.
- Parallel vs Concurrent - Parallel means that code is being executed at the same time, while concurrent means that code is run together (the CPU switches between one task and another quickly). Python processes are parallel. Python threads are concurrent.
- Memory leak - The unintentional escape or misuse of memory i.e. forgetting to free memory after using it.
- Memory corruption - Altering data in a fashion that makes it unreadable or harmful to read by the program.
- Interpreter - The software that reads, translates, and executes Python. One interpreter per Python process.
- Lock - An object used to control access to a shared resource, usually another object.
- Semaphores - These are locks with counters, they keep track of the number of calls made to acquire() and release().

- Global Interpreter Lock (GIL) - A mechanism that ensures only one thread executes at a time, limiting parallelism in multi-threaded programs. One GIL per interpreter.
- Deadlock - A situation in which two or more processes are unable to proceed because each one is waiting for the other to release a resource (call release() on a lock).
- Race condition —An insidious bug where two or more threads both make conflicting changes to the same data, but the order of those changes is dictated by when the threads start so the outcome becomes unpredictable. Locks prevent this.
- Data corruption - Unlike memory corruption, there is nothing wrong with the data in memory, but it may be a value that is unexpected by the program e.g. a thread emptying a list while another thread is using it.

# Questions

1. What are the gotchas I should look out for?
2. What is a thread?
3. What is multithreading?
4. How does multithreading differ from multiprocessing?
5. When should I use threads vs processes?
6. What is the Global Interpreter Lock (GIL)?
7. What is the Global Interpreter Lock (GIL) on multithreading?
8. How do I create a new thread?
9. How do I pass arguments into a thread?
10. How do I start a thread? What are the three gotchas for starting threads?
11. What are the different types of threads?
12. What is a daemon thread?
13. How do I create a daemon thread?
14. How do I check if a thread has started? If it is running? Stopped?
15. How do I wait for a thread to finish? What is thread joining?
16. Can I timeout thread.join() - how do I wait X seconds for a thread to finish?
17. How do I stop a thread?
18. Can I restart a thread?
19. Does a thread exit when it runs into an unhandled exception?
20. What happens to unhandled exceptions in threads? Does it cause an error in the parent thread?
21. How do I handle exceptions thrown in threads?
22. What are the four limitations of threading.excepthook?

23. How do I exchange information between threads?
24. What is thread safety?
25. What happens if I don’t use thread-safe objects?
26. Why do I need thread-safe objects if Python has the Global Interpreter Lock (GIL)?
27. Do all objects passed into threads have to be thread-safe?
28. How do you synchronize threads to avoid race conditions? How do you tell one thread to wait for another?
29. How can I keep track of resources used by multiple threads? What are semaphores?
30. How can I schedule a thread to start at a particular time?

# What are the gotchas I should look out for?

1. Don’t use threading.Thread, there are better alternatives that require less setup, have friendly interfaces, and are more forgiving - only use threading.Threads if the two options below are insufficient (both use threading.Thread underneath).
    a. Asyncio - gives you the option to switch to using processes instead of threads
    b. concurrent.futures.ThreadPoolExecutor - handles creating, executing, and keeping track of threads
2. Threads do NOT execute code simultaneously because the Global Interpreter Lock limits the CPU to execute only one thread per Python process. Threads are not run in parallel.
3. Threads aren't meant for everything
    a. If you CPU heavy tasks (math, data transformations, etc.), use multiprocessing, not multithreading - if you I/O heavy tasks (network calls, DB queries, disk reads/writes), use multithreading.
    b. But if you need to do a lot of communication between your tasks, use threads - shared memory makes it easier to communicate vs Inter Process Calls (IPC), network calls, third-party services, pickling, etc., for processes
    c. Need to use shared memory i.e. work on one object, threads
4. You have to keep track of the thread’s state.
    a. You can only call thread.start() once.
    b. You can call thread.join() multiple times but…
    c. It will error when calling thread.join() on itself i.e. you can’t join a thread on itself because it will create a deadlock.
    d. You can’t call thread.join() on a thread that has not started - causes a runtime error.
5. Stopping threads is tricky.
    a. Threads can cause your program to freeze. Once you start a thread, it does NOT stop until it is completed or runs into an unhandled error - even if the process tries to exit (shutdown the

program), the thread keeps running and prevents the exit (unless it is a daemon thread).
    b. Daemon threads don’t freeze your program - you can exit the current thread, and if the daemon thread is the only thing alive, the Python will kill, but this may cause memory leaks (DB connections, etc. won't be cleaned).
6. Threads don’t start and stop when you want them to but when they can.
    a. thread.start() does NOT immediately execute the function passed into the thread - it attempts to find a free thread, but if all the process’s threads are busy, it waits for them to be free.
    b. thread.is_alive() will return False even if the thread is not freed (after the code is run, the thread needs to be cleaned up and waits to do so). Be careful if you want to reuse that same CPU thread to run a different piece of code or if you want to immediately start another job thinking you have a thread that is free.
7. You can’t rely on Python to gracefully handle errors in threads.
    a. Unhandled errors thrown in child threads will be silent unless you catch them or use threading.excepthook (in the parent) to listen for errors.
    b. threading.excepthook will not be called when a system kill message is sent i.e. when the program is abruptly closed.
    c. Errors in threading.excepthook will be sent to the System errors handler, so make sure to set that up if you want to do complex logging/alerting in excepthook.
    d. Delete the trace and error objects in excepthook, don’t pass them around or you will get a circular dependency and eventually a memory leak.
    e. Start/Run/Join don’t propagate exceptions thrown within the thread.
8. Debugging threads can be hard; don’t treat errors in threads like regular errors.
    a. Threads share the same memory but NOT the same callstack - error traces will be available but out of context in

threading.excepthook. The error traces won't show who started the thread, only what function the thread ran and where the error occurred in that function.

   b. Once the thread throws an error, it goes through the clean up process. This means you won't have access to actual objects (or any other objects from within the thread) that caused the error when you handle the error in excepthook.
   c. Race conditions are a pain - this bug occurs when two threads try to access the same resource, and the order of their access is completely random, resulting in unpredictable states. Imagine a banking app that uses two separate threads for deposits and withdrawals. If a customer decides to deposit and withdraw at the same time but these two threads are not synchronized with each other (e.g. do deposits first always), the customer can end up with a negative balance. This is what locks are meant to solve but it is very hard to detect race conditions since the outcome is so random - you have to wait until something goes wrong.

9. Forgetting that threads need thread-safe objects to avoid race conditions and data corruption.
   a. It is easy to pass objects into threads and start changing them without realizing that other threads may depend on this object. Before you pass anything into a thread, remember to ask yourself what you will do to it.
10. Forgetting to release locks - deadlocks.
   a. Every call to acquire() should have a mirror call to release(). This very is easy to forget. A good coding habit is to never write one of these methods alone—always write them together, even if you don’t know when you call the other method.

# What is a thread?

A thread is a semi-isolated stream of operations - a single worker at a factory is a thread. They can do work without supervision, but since they work in the factory, they have to share the same resources as everyone else in the factory. Threads share memory with the parent thread that created them and their sibling threads. The call stack, the series of functions executed by the thread, is unique and separate for each thread - this means you won't be able to find where the thread was started using the error trace, only what function the thread executed.

# What is multithreading?

Multithreading is an approach to solving problems by splitting them into multiple independent execution paths. Imagine a worker at a car factory, that worker can do everything to build the car or we can have multiple workers each building different parts of the car separately then putting them together. The latter approach is more efficient under certain circumstances. Multithreading is just that, using more workers (threads) to do a task concurrently (together but not at the same time).

But don’t just use multithreading everywhere. It is more expensive resource-wise, and the code is more complex. Knowing when to use multithreading is 80% of the journey. Once you know when, creating and managing threads becomes easy. Even synchronizing them can be simple with the correct mental model.

# How does multithreading differ from multiprocessing?

If a thread is a worker at a factory. A process is the factory. It is an isolated stream of operations and resources that contains at least one thread.

Multithreading

- Belongs to one process
- Semi-isolated - Shares the same memory as other threads in the process
- Separate callstack/Isolated execution context - separate function callstack
- Semi-parallel - Limited by Global Interpreter Lock (GIL), all threads share the same Python interpreter, so only one thread can be executed by the CPU at a time
- Resilient - Errors can only take down the thread they occur in. The process continues and can create more threads
- Shared memory - Threads can share variables with each other, which makes communication between them easier. Only threads within the same process.

Multiprocessing

- Autocephalous - Does not belong to another process
- Fully-parallel - Each process has its own GIL, so each process can be run in parallel on the CPU
- Error-prone - If an error is unhandled, takes down the entire process
- Isolated - Has its own memory space and callstack

- Hard communication - Have to use IPC, HTTP, Pickling, created shared memory, or another mechanism to communicate between processes

## When should I use threads vs processes?

- I/O heavy tasks - reading and writing to/from disk, DB, network, etc. where there is a lot of waiting
- Error-prone tasks - it is cheap and easy to restart threads, and one thread failing won't take down the entire process. If you have a task that fails unpredictably, run it in a separate thread
- Communication heavy - if you have a task that relies a lot on moving data from one worker to another use threads, the savings in latency by communicating via shared memory will overcome the latency of GIL
- Need many workers - if you need lots of workers, than threads are usually the way to go, the number of threads your computer can spin up and run is always higher than the number of processes (although the later is virtually infinite, the number process that can actually do work is limited by the number of CPUs and cores)

But you shouldn't use threads or processes directly; you should use the AsyncIO module - whatever your problem is, you probably don't need to implement a solution from scratch using threads or processes. AsyncIO is an abstraction over multithreading and multiprocessing - it turns your python script into an event-driven script; instead of waiting for network or DB calls to be finished, you can pass those calls a callback and continue running the script. AsyncIO will spin up a thread or process (if you want it, sometimes, it can get away with just using the main thread) to do the waiting and response handling. AsyncIO handles all the multithreading and multiprocessing use cases. It is well-tested and documented. If that's not

enough, there are other libraries built on top of AsyncIO for specific tasks like HTTP communication.

# What is the Global Interpreter Lock (GIL)?

It is the toll gate at the heart of Python. For each Python process, there is exactly one Python interpreter (the thing that reads and executes your Python code). However, processes can have multiple threads running at the same time. What if two threads try to use the same location in memory, e.g., the same object simultaneously? What if a thread tries to delete an object that is being used by another thread? All these behaviors are unpredictable and would cause errors. The GIL exists to solve these problems in the simplest possible way by making these scenarios impossible or at least less likely. It does this by limiting how many threads can be executed by the interpreter at a time. That's why it is called a lock. It locks the interpreter from all other threads while a thread is executing. You don't have to worry about memory corruption while multithreading in Python.

# What is the Global Interpreter Lock (GIL) impact on multithreading?

The GIL limits one thread per process (each process has one Python interpreter) to execute on the CPU simultaneously. This prevents threads from executing conflicting code e.g. deleting an object another thread is using. But this limits parallelism; operations that use the CPU (math, data manipulation, etc.) are run sequentially when using threads. This means that Python threads not run in parallel - the threads are not running at the same time.

# How do I create a new thread?

1. Import the threading module
2. Create a new instance of threading.Thread class
    a. Pass it the function you want to execute on the thread

```
import threading

# Define a function that will be executed in the new thread
def print_numbers():
    for i in range(1, 6):
        print("Number:", i)

# Create a new thread and specify the function to execute
thread = threading.Thread(target=print_numbers)
```

# How do I pass arguments into a thread?

- One argument - threading.Thread(target=print_numbers, args=[6])
    - args expects only iterables (tuples, lists, etc.) to pass a single variable use (6,) or [6]
    - (6) is not a tuple; it is just 6
- Multiple arguments - threading.Thread(target=print_numbers, args=[ARG1, ARG2, ARG3, ..., ARGN])
- Keyword arguments - threading.Thread(target=print_numbers, kargs={ ‘key1’: val1, ‘key2’: val2 })

```
import threading

def print_numbers(num):
    for i in range(1, num):
            print("Number:", i)

thread = threading.Thread(target=print_numbers, args=[6])
```

# How do I start a thread? What are the three gotchas for starting threads?

Call start() on the thread object to start it:

```
def print_numbers(num):
    for i in range(1, num):
        print("Number:", i)
thread = threading.Thread(target=print_numbers, args=[6])
thread.start()
```

But beware.

- You can only call start() ONCE - it will error if you do it a second time. So, you need to keep track of which threads have started and stopped.
- start() does NOT immediately execute the function passed into the thread - it attempts to create a thread but if the max number of threads for process have been created it waits for a thread to be free.
    - https://github.com/python/cpython/blob/main/Lib/threading.py#L967
- Once you start the thread it does NOT stop until it is completed or runs into an unhandled error - even if the process tries to exit (shutdown the program), the thread keeps running and prevents the exit (unless it is a daemon thread).

## What are the different types of threads?

- Regular threads - will keep the program alive until they are done
- Daemonic threads - can be abruptly killed by the program

# What is a daemon thread?

Daemon (pronounced like demon but not as nefarious, more like spirits) threads are threads meant for background jobs. They are excellent for tasks that can or need to be abruptly closed.

But daemonic threads come with caveats.

1. When the program exits, the daemonic threads are aborted - they don’t keep the program open
2. If only daemonic threads are left, the program will automatically shut down
3. This abrupt shutdown prevents open files, connections, etc., from being released, causes memory leaks, and freezes other programs (the ones on the other end of the connections).
4. By default, daemonic threads only spawn daemonic threads - be careful when creating threads inside of daemonic threads; you may accidentally cause memory leaks (see point 3).

# How do I create a daemon thread?

Python 3.3 and later:

```
thread = threading.Thread(target=print_numbers, daemon=True)
```

All older versions of Python, very error-prone since you can forget to set the flag.

```
thread = threading.Thread(target=print_numbers)
thread.daemon = True
```

A common pattern in older versions of Python to avoid forgetting to set the flag:

```
class DaemonThread(threading.Thread):
  def __init__(self, target=None, args=(), kwargs={}):
      super(DaemonThread, self).__init__(target=target, args=args,
   kwargs=kwargs)
      self.daemon = True
```

# How do I check if a thread has started? If it is running? Stopped?

- Started, use thread.is_alive(): https://docs.python.org/3/library/threading.html#threading.Thread.is_alive
    - Returns true after thread.start() is called
- Running, use thread.is_alive(), it will return true up until the code finishes execution but NOT the thread: https://docs.python.org/3/library/threading.html#threading.Thread.ident
    - This is set in the start() call AFTER a thread has been created or assigned
- Stopped, use thread.is_alive(), it will return false
- thread.is_alive() will return False even if the thread is not freed (after the code is run, the thread needs to be cleaned up and waits to do so). Be careful if you want to reuse that same CPU thread to run a different piece of code or if you want to immediately start another job thinking you have a thread that is free.
    - https://github.com/python/cpython/blob/main/Lib/threading.py#L1047

# How do I wait for a thread to finish? What is thread joining?

You can use join() to wait for a thread to finish - this is called joining?

```
thread.join()
print("Main thread continues...")
```

- thread.join() pauses the current thread
- You can call join() multiple times
- BUT you can only call join() on a started thread - if you have not called thread.start() or if the thread is stopped i.e. if is_alive() returns false, thread.join() will error
- You can’t call thread.join() on the current thread - this will cause an error

# The Rest

All 30 questions can be found here for free on my blog.