**Multithreading in Python**

[[Anand Rathore](https://medium.com/@av44314?source=post_page-----093e89a602c8--------------------------------)](https://medium.com/@av44314?source=post_page-----093e89a602c8--------------------------------)

[[Towards Dev](https://towardsdev.com/?source=post_page-----093e89a602c8--------------------------------)](https://towardsdev.com/?source=post_page-----093e89a602c8--------------------------------)

[Anand Rathore](https://medium.com/@av44314?source=post_page-----093e89a602c8--------------------------------)

·

Follow

Published in

[Towards Dev](https://towardsdev.com/?source=post_page-----093e89a602c8--------------------------------)

·

14 min read

·

Oct 27

184

1

Master Python multithreading with our guide! Learn thread creation, synchronization, and practical examples. Boost your project’s performance.



Photo by [amirali mirhashemian](https://unsplash.com/@amir_v_ali?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

**1. Introduction**

Python is a versatile and powerful programming language, known for its simplicity and ease of use. However, when it comes to handling tasks concurrently and efficiently, Python’s Global Interpreter Lock (GIL) can pose a challenge. Multithreading is one way to overcome this limitation and make the most out of your CPU cores.

**2. Understanding Multithreading in Python**

Multithreading is a programming technique that enables a single process to execute multiple threads concurrently. Each thread runs independently and can perform different tasks simultaneously. This is particularly useful in Python, where the Global Interpreter Lock (GIL) can restrict the execution of multiple threads. Multithreading can help you utilize the full potential of your CPU cores by allowing your program to perform multiple tasks concurrently.

**3. Python Threading Module**

Python provides a built-in threading module that simplifies multithreading. You can create and manage threads using this module. Here are some of the key concepts related to the Python threading module:

**3.1. Thread Creation**

Thread creation in Python involves using the built-in **threading** module, which provides a straightforward way to create and manage threads. Threads are a fundamental unit of concurrency that allows your Python program to execute multiple tasks concurrently. Here’s how to create threads in Python:

**3.1.1 Import the threading Module**

To get started with thread creation, you need to import the **threading** module, which is part of Python’s standard library.

import threading

**3.1.2.Define a Thread Function**

Next, you define a function that will be executed by the thread. This function contains the code you want to run concurrently in a separate thread. It can take any parameters as needed.

def my\_function(arg1, arg2):  
 # Your thread's code here

**3.1.3. Create Thread Objects**

To create a thread, you instantiate a **threading.Thread** object and pass your thread function as the **target** argument. You can also specify any arguments that your thread function requires.

my\_thread = threading.Thread(target=my\_function, args=(arg1, arg2))

Here, my\_thread is a thread object that is ready to execute my\_function when started.

**3.1.4. Start the Thread**

To begin executing the thread, you call the start() method on the thread object.

my\_thread.start()

This will initiate the thread’s execution, and it will run concurrently with the main program.

**3.1.5. Thread Termination (Optional)**

Threads will continue to execute until their target function is completed, or you manually terminate them. To wait for a thread to finish its execution, you can use the join() method on the thread object.

my\_thread.join()

This will block the main program until my\_thread completes its task.

**3.1.6. Example**

Here’s a complete example of creating and starting a thread:

import threading  
import time  
  
def print\_numbers():  
 for i in range(1, 6):  
 print(f"Number {i}")  
 time.sleep(1)  
  
def print\_letters():  
 for letter in 'abcde':  
 print(f"Letter {letter}")  
 time.sleep(1)  
  
# Create thread objects  
thread1 = threading.Thread(target=print\_numbers)  
thread2 = threading.Thread(target=print\_letters)  
  
# Start the threads  
thread1.start()  
thread2.start()  
  
# Wait for both threads to finish  
thread1.join()  
thread2.join()  
  
print("Both threads have finished.")

In this example, two threads (**thread1** and **thread2**) are created and run concurrently, printing numbers and letters.

**3.2. Thread Synchronization**

Thread synchronization in Python is the process of coordinating and controlling the execution of multiple threads to ensure data integrity and prevent conflicts when they access shared resources or manipulate shared data. Without proper synchronization, concurrent threads can lead to issues like data corruption, race conditions, and unpredictable program behavior. Python provides several mechanisms for thread synchronization to address these problems:

**3.2.1. Locks (Mutexes)**

* Locks, often referred to as mutexes (short for mutual exclusion), are the most fundamental synchronization mechanism in Python.
* A lock allows only one thread to acquire it at a time. If another thread attempts to acquire a locked lock, it will block until the lock is released.
* Locks are typically used to protect critical sections of code, ensuring that only one thread can execute that section at any given time.

Example using the **threading** module:

import threading  
  
# Create a lock  
lock = threading.Lock()  
  
def safe\_increment(counter):  
 with lock:  
 counter += 1  
 return counter

**3.2.2. Semaphores**

* Semaphores are used to control access to a limited number of resources, allowing a specified number of threads to access a resource concurrently.
* Semaphores have an internal counter that decrements when a thread acquires the semaphore and increments when the thread releases it.

Example using the **threading** module:

import threading  
  
# Create a semaphore with a limit of 3  
semaphore = threading.Semaphore(3)  
  
def limited\_resource\_access():  
 with semaphore:  
 # Only three threads can access this block at a time  
 # Other threads will block until a slot is available  
 # ...

**3.2.3. Conditions**

* Conditions provide a way for threads to coordinate and communicate with each other.
* They are often used to wait for a certain condition to be met before proceeding.
* A condition has two parts: a lock and a waiting queue. Threads can wait for a condition to be signaled and can signal it when a particular condition is met.

Example using the **threading** module:

import threading  
  
# Create a condition  
condition = threading.Condition()  
  
def producer():  
 with condition:  
 # Produce some data  
 condition.notify() # Signal that data is ready  
  
def consumer():  
 with condition:  
 while not data\_available():  
 condition.wait() # Wait for data to be ready  
 # Consume the data

**3.2.4. RLocks (Reentrant Locks)**

* An RLock (Reentrant Lock) is a variant of a lock that allows a thread to acquire the lock multiple times, which can be helpful in nested thread synchronization scenarios.
* A thread holding an RLock can acquire it again without causing a deadlock, as long as it releases the lock the same number of times it acquired it.

Example using the **threading** module:

import threading  
  
rlock = threading.RLock()  
  
def nested\_lock\_example():  
 with rlock:  
 # This thread holds the lock  
 with rlock:  
 # This thread can acquire the lock again without blocking  
 # ...

Thread synchronization is crucial in multithreaded applications to ensure that threads work cooperatively, avoid conflicts, and maintain data consistency. The choice of synchronization mechanism depends on the specific requirements of your application, and it’s essential to use them correctly to avoid potential issues like deadlocks or performance bottlenecks.

**3.3. Thread Communication**

Thread communication in Python is the process of allowing multiple threads in a program to exchange data, synchronize their execution, or coordinate their activities to work together effectively. Effective thread communication is essential to prevent data races, ensure thread safety, and enable threads to work together harmoniously. Python provides several mechanisms to facilitate thread communication, and here are some of the key ones:

**3.3.1 Queues**

Queues are one of the most common ways to enable communication and data sharing between threads in Python. The **queue** module provides the **Queue** class, which is a thread-safe, FIFO (First-In-First-Out) data structure. Threads can put items into the queue and retrieve items from it in a safe manner.

import threading  
import queue  
  
# Create a thread-safe queue  
my\_queue = queue.Queue()  
  
def producer():  
 # Add items to the queue  
 my\_queue.put(item)  
  
def consumer():  
 # Retrieve items from the queue  
 item = my\_queue.get()

This pattern is useful for scenarios where one or more threads produce data, and one or more threads consume that data.

**3.3.2. Events**

Events are synchronization primitives that allow one or more threads to wait until a particular condition is met. Threads can wait for an event to be set and proceed when another thread signals the event.

import threading  
  
# Create an event  
event = threading.Event()  
  
def thread1():  
 # Wait for the event to be set  
 event.wait()  
 # Continue execution  
  
def thread2():  
 # Set the event to signal thread1 to continue  
 event.set()

Events are useful for scenarios where threads need to coordinate their activities or wait for specific conditions before proceeding.

**3.3.3. Condition Variables**

Condition variables provide a way for threads to wait for a particular condition to become true. Threads can wait for a condition to be signaled and can also notify other threads when the condition is met.

import threading  
  
# Create a condition variable  
condition = threading.Condition()  
  
def producer():  
 with condition:  
 # Produce data  
 condition.notify() # Signal consumer thread  
  
def consumer():  
 with condition:  
 condition.wait() # Wait for producer signal  
 # Consume data

Condition variables are suitable for scenarios where multiple threads need to cooperate based on specific conditions.

**3.3.4. Thread Pools**

Thread pools allow you to manage a group of worker threads efficiently. You can submit tasks to a thread pool, and it will allocate available threads to execute those tasks concurrently.

from concurrent.futures import ThreadPoolExecutor  
  
# Create a thread pool  
with ThreadPoolExecutor(max\_workers=4) as executor:  
 # Submit tasks for execution  
 result1 = executor.submit(task1)  
 result2 = executor.submit(task2)

Thread pools are useful when you have a batch of tasks that can be executed concurrently, and you want to control the number of threads used for parallelism.

These thread communication mechanisms provide a structured and safe way for threads to work together and share data in a multi-threaded Python program. Properly managing thread communication is crucial to avoid race conditions, deadlocks, and other synchronization issues that can arise when multiple threads interact with shared resources.

**3.4. Daemon Threads**

Daemon threads in Python are a special type of threads that run in the background and are not considered essential for the program to continue running. In other words, they are threads that do not prevent the program from exiting, even if they are still executing. Understanding daemon threads is essential when working with multithreaded Python programs.

Here are key points to understand about daemon threads in Python:

**3.4.1. Daemon vs. Non-Daemon Threads**

In Python’s **threading** module, when you create a thread using the **threading.Thread** class, is initially treated as a non-daemon thread. Non-daemon threads are considered essential threads, and the program will wait for all non-daemon threads to complete before it exits. In contrast, daemon threads are not essential, and the program will exit even if there are active daemon threads.

**3.4.2. Creating Daemon Threads**

To create a daemon thread in Python, you can set the **daemon** attribute of a thread object to **True** before starting the thread. Here’s an example:

import threading  
  
def daemon\_function():  
 while True:  
 # Daemon thread's code here  
  
daemon\_thread = threading.Thread(target=daemon\_function)  
daemon\_thread.daemon = True # Set the thread as a daemon  
daemon\_thread.start()

**3.4.3. Use Cases for Daemon Threads**

Daemon threads are useful for background tasks that should not block the program’s exit. Some common use cases for daemon threads include:

* **Loggers:** Threads that continuously log data or monitor logs in the background.
* **Timer Threads:**Threads that perform periodic tasks, such as data cleanup, without affecting the program’s main functionality.
* **Monitoring Threads:**Threads that monitor system resources or external events without stopping the program.

**3.4.4. Caution with Daemon Threads**

While daemon threads can be convenient for handling background tasks, it’s essential to be cautious when using them. Since daemon threads can be abruptly terminated when the program exits, you should avoid using them for tasks that require cleanup or proper termination procedures. Resources held by daemon threads may not be released gracefully.

**3.4.5. Daemon Threads and the Global Interpreter Lock (GIL)**

Daemon threads, like all threads in Python, are subject to the Global Interpreter Lock (GIL). This means that they may not be suitable for CPU-bound tasks that require efficient multithreading. For CPU-bound tasks, you may want to consider using multiprocessing or other concurrency techniques.

**4. Python GIL (Global Interpreter Lock) and Its Impact on Multithreading**

In Python, the Global Interpreter Lock (GIL) is a mutex, or a lock, that protects access to Python objects, preventing multiple threads from executing Python code simultaneously in a single process. The GIL has a significant impact on multithreading in Python and is a topic of discussion and concern for many Python developers.

**4.1. Understanding the GIL**

**4.1.1. What is the GIL?**

The GIL is a mechanism implemented in the CPython interpreter, which is the default and most widely used Python interpreter. It is not present in all Python implementations. The GIL is essentially a mutex that allows only one thread to execute Python bytecode at a time, even on multi-core processors.

**4.1.2. Why does the GIL exist?**

The primary reason for the GIL’s existence is to simplify memory management in CPython. Without the GIL, managing Python objects across multiple threads would become more complex and prone to issues like data corruption and race conditions.

**4.2. Impact of the GIL on Multithreading**

**4.2.1. Limitation in CPU-bound tasks**

The GIL can be a significant limitation when it comes to CPU-bound tasks. Since only one thread can execute Python code at a time, multithreading doesn’t provide a significant performance boost for CPU-bound tasks in Python. In fact, it may even result in slower execution due to the overhead of acquiring and releasing the GIL.

**4.2.2. Beneficial for I/O-bound tasks**

Where the GIL shines is in I/O-bound tasks. In scenarios where threads spend a lot of time waiting for I/O operations to complete (e.g., reading from files or making network requests), the GIL doesn’t hinder performance significantly. In such cases, you can still achieve concurrency and benefit from multithreading.

**4.3. Strategies to Work Around the GIL**

Despite the GIL’s limitations, Python provides alternative approaches to achieve parallelism and concurrency:

**4.3.1. Multiprocessing**

Instead of multithreading, you can use the multiprocessing module to create separate processes. Each process has its own Python interpreter and memory space, effectively bypassing the GIL. This approach is suitable for CPU-bound tasks and can utilize multiple CPU cores.

**4.3.2. Asyncio**

For I/O-bound tasks, you can leverage asynchronous programming with asyncio. Asyncio allows you to write non-blocking code, enabling cooperative multitasking without the need for multiple threads. It’s efficient for handling a large number of I/O-bound operations concurrently.

**5. Real-world multithreading in Python**

Real-world multithreading in Python involves using the threading module to solve practical problems that benefit from concurrent execution. Multithreading is particularly useful when you have tasks that can be executed independently and can run concurrently to improve efficiency. Here are some real-world scenarios where multithreading can be applied in Python:

**5.1. Web Scraping**

* In web scraping, you often need to fetch data from multiple web pages simultaneously.
* Create a thread for each web page to fetch data concurrently, reducing the overall scraping time.
* Be cautious about web scraping ethics and website terms of service to avoid legal issues.

import threading  
import requests  
  
def fetch\_data(url):  
 response = requests.get(url)  
 # Process the response data  
  
urls = ["https://example.com/page1", "https://example.com/page2", "https://example.com/page3"]  
  
threads = []  
for url in urls:  
 thread = threading.Thread(target=fetch\_data, args=(url,))  
 threads.append(thread)  
 thread.start()  
  
for thread in threads:  
 thread.join()

**5.2. Image Processing**

* When processing a batch of images, apply filters or transformations to each image concurrently.
* Multithreading can reduce the time required to process a large number of images.

import threading  
from PIL import Image  
  
def process\_image(image\_path):  
 img = Image.open(image\_path)  
 # Apply image processing operations  
 img.save(output\_path)  
  
image\_paths = ["image1.jpg", "image2.jpg", "image3.jpg"]  
  
threads = []  
for path in image\_paths:  
 thread = threading.Thread(target=process\_image, args=(path,))  
 threads.append(thread)  
 thread.start()  
  
for thread in threads:  
 thread.join()

**5.3. Concurrent API Requests**

* When making multiple API requests, each request can be handled by a separate thread.
* Multithreading can help speed up data retrieval from various APIs concurrently.

import threading  
import requests  
  
def fetch\_data\_from\_api(api\_url):  
 response = requests.get(api\_url)  
 # Process the API response data  
  
api\_urls = ["https://api.example.com/data1", "https://api.example.com/data2", "https://api.example.com/data3"]  
  
threads = []  
for url in api\_urls:  
 thread = threading.Thread(target=fetch\_data\_from\_api, args=(url,))  
 threads.append(thread)  
 thread.start()  
  
for thread in threads:  
 thread.join()

**5.4. Real-time Data Processing**

In applications that require real-time data processing, such as chat applications or sensor data analysis, multithreading can be used to handle incoming data streams concurrently.

import threading  
import socket  
  
def handle\_client(client\_socket):  
 while True:  
 data = client\_socket.recv(1024)  
 # Process and respond to incoming data  
  
server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
server.bind(("0.0.0.0", 8080))  
server.listen(5)  
  
while True:  
 client, addr = server.accept()  
 client\_thread = threading.Thread(target=handle\_client, args=(client,))  
 client\_thread.start()

**5.5. Parallel File Downloads**

* When downloading multiple files from the internet, each download can be done in a separate thread.
* Multithreading speeds up the file download process.

import threading  
import requests  
  
def download\_file(url, save\_path):  
 response = requests.get(url)  
 with open(save\_path, "wb") as file:  
 file.write(response.content)  
file\_urls = ["https://example.com/file1.zip", "https://example.com/file2.zip", "https://example.com/file3.zip"]  
save\_paths = ["file1.zip", "file2.zip", "file3.zip"]  
threads = []  
  
for url, path in zip(file\_urls, save\_paths):  
 thread = threading.Thread(target=download\_file, args=(url, path))  
 threads.append(thread)  
 thread.start()  
  
for thread in threads:  
 thread.join()

**6. Best Practices for Multithreading in Python**

**1. Understand the GIL:**One of the first things you should do when working with multithreading in Python is to understand the Global Interpreter Lock (GIL). The GIL limits the execution of multiple threads simultaneously for CPU-bound tasks. If your application relies heavily on CPU-bound operations, consider using multiprocessing or other concurrency approaches that bypass the GIL.

**2. Use Thread Pools:** When dealing with a large number of tasks that can be executed concurrently, it’s often better to use a thread pool rather than creating individual threads for each task. Python’s **concurrent.futures** module provides **ThreadPoolExecutor** and **ProcessPoolExecutor**classes for easy thread and process pooling.

from concurrent.futures import ThreadPoolExecutor  
  
with ThreadPoolExecutor(max\_workers=4) as executor:  
 results = executor.map(my\_function, args)

**3. Thread Safety:** Ensure thread safety by using locks, semaphores, or other synchronization mechanisms when multiple threads access shared resources. This prevents data races and ensures that your program behaves predictably.

**4. Avoid Global Variables:** Minimize the use of global variables, as they can lead to unintended consequences when multiple threads modify them simultaneously. Instead, encapsulate shared data within objects and use synchronization mechanisms to control access.

**5. Error Handling:**Proper error handling is crucial in multithreaded programs. Make sure to catch exceptions within threads, and log errors, and handle them gracefully to prevent crashes and unexpected behavior.

**6. Resource Cleanup:**Threads should release any acquired resources (e.g., files, database connections) when they are done using them. Failing to release resources properly can lead to resource leaks and application instability.

**7. Thread Naming:** Give your threads meaningful names to make debugging and monitoring easier. You can set thread names using **threading.Thread’**s **name** parameter.

import threading  
  
thread = threading.Thread(target=my\_function, name="WorkerThread")

**8. Avoid Excessive Thread Count:** Creating too many threads can lead to high memory usage and increased overhead. Use thread pooling or a reasonable number of threads based on the available CPU cores and the nature of your tasks.

**7. Pitfalls to Avoid in Multithreading**

**1. Deadlocks:**Deadlocks occur when two or more threads wait indefinitely for resources that each holds while refusing to release them. To prevent deadlocks, use proper locking strategies and avoid circular dependencies in resource acquisition.

**2. Race Conditions:** Race conditions happen when multiple threads access shared data simultaneously, leading to unpredictable behavior. Always use synchronization mechanisms like locks or semaphores to protect shared resources.

**3. Thread Starvation:** Thread starvation occurs when one or more threads do not get the opportunity to execute due to poor thread management. Properly manage thread execution and scheduling to avoid this issue.

**4. Inefficient Thread Creation:** Creating and destroying threads can be expensive. Avoid creating threads for short-lived tasks and consider using thread pools or other concurrency patterns for better efficiency.

**5. Over-Optimization:**Premature optimization can lead to complex and error-prone code. Focus on optimizing critical sections of your code where performance gains are most significant, rather than attempting to optimize every part of your program.

**6. Lack of Testing:** Multithreaded programs can be challenging to debug and test. Invest in comprehensive testing, including stress testing and concurrency testing, to identify and resolve issues early in the development process.

**7. Ignoring CPU vs. I/O Bound:**Be aware of the nature of your tasks. If your tasks are I/O bound (e.g., reading/writing files, making network requests), multithreading can be effective. For CPU-bound tasks, consider alternative concurrency approaches such as multiprocessing or async/await.

**8. Not Monitoring and Profiling:**Use profiling and monitoring tools to identify performance bottlenecks, contention points, and areas that may benefit from optimization. Tools like **cProfile** and **threading**’s built-in instrumentation can be invaluable for this purpose.

**8. Conclusion**

Multithreading in Python can significantly enhance the performance of your applications, especially in scenarios where parallel execution is crucial. Despite the challenges posed by the Global Interpreter Lock (GIL), Python’s threading module provides a robust framework for managing threads and synchronizing their activities.

By applying the knowledge gained here, you’ll be well-prepared to leverage the power of multithreading in your Python projects. Whether you’re building a web crawler, optimizing image processing, or managing concurrent I/O operations, mastering multithreading will undoubtedly be a valuable skill in your Python programming journey.

Also, read [Multithreading in Java](https://codesarray.com/view/Multithreading-in-Java)