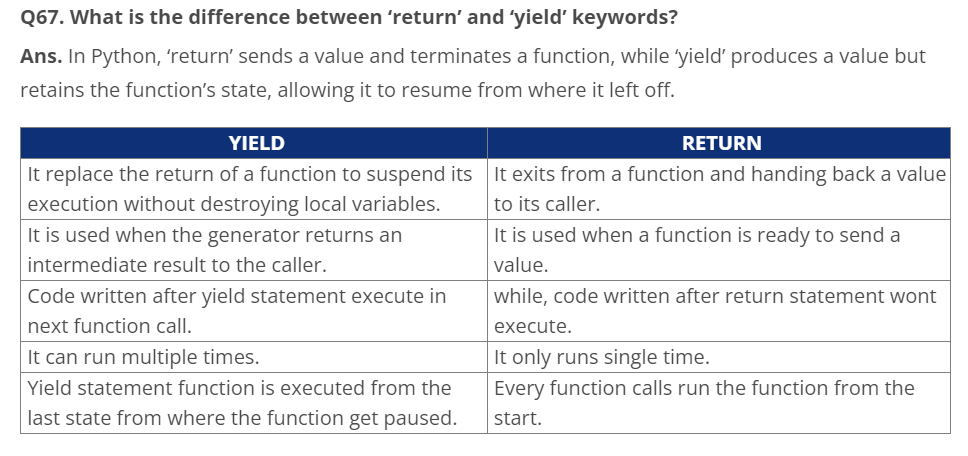
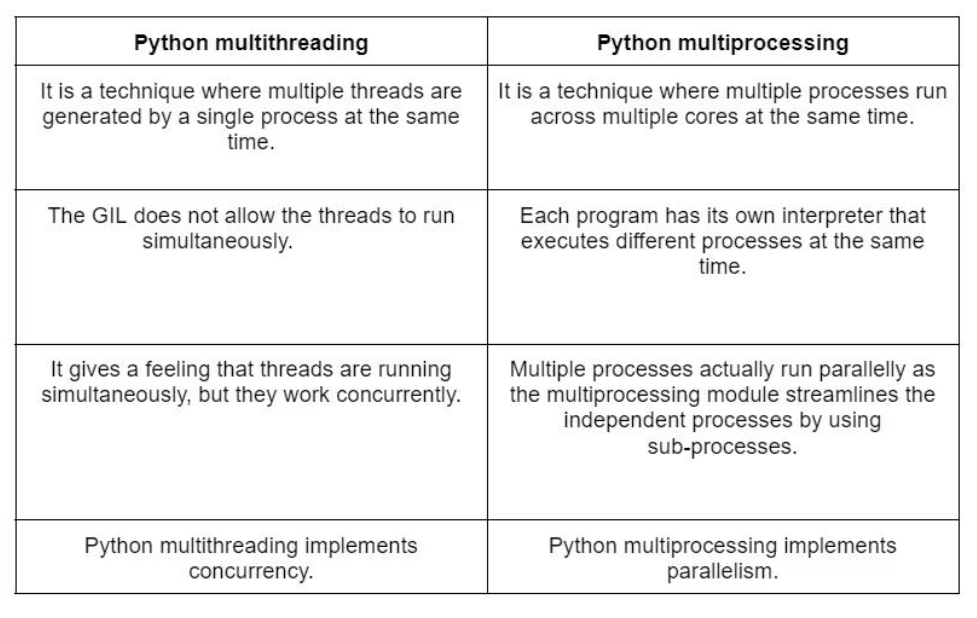
# **Python Notes**





## **asyncio in Python:**

Asyncio is a Python library that is used for concurrent programming, including the use of async iterator in Python. It is not multi-threading or multi-processing.

**Key Concepts**

* **Event Loop:** The core of every asyncio application. It runs asynchronous tasks and callbacks, performs network IO operations, and runs subprocesses.
* **Coroutine:** Special functions defined with async def. They can use await to yield control back to the event loop, allowing other tasks to run.
* **Task:** A wrapper for coroutines. It allows the event loop to execute the coroutine concurrently with other tasks.
* **Future:** A low-level awaitable object that represents a result that may not be available yet.

## **Difference Between Asynchronous and Multi-Threading Programming**

Asynchronous programming allows only one part of a program to run at a specific time.

Consider three functions in a Python program: fn1(), fn2(), and fn3().

In asynchronous programming, if fn1() is not actively executing (e.g., it’s asleep, waiting, or has completed its task), it won’t block the entire program.

Instead, the program optimizes CPU time by allowing other functions (e.g., fn2()) to execute while fn1() is inactive.

Only when fn2() finishes or sleeps, the third function, fn3(), starts executing.

This concept of asynchronous programming ensures that one task is performed at a time, and other tasks can proceed independently.

In contrast, in multi-threading or multi-processing, all three functions run concurrently without waiting for each other to finish.

With asynchronous programming, specific functions are designated as asynchronous using the async keyword, and the asyncio Python library helps manage this asynchronous behavior.

## **Multi-Processing:**

Multiprocessing in Python is a module that allows you to run multiple processes simultaneously, taking advantage of multiple CPU cores to perform tasks concurrently. This is particularly useful for CPU-bound tasks where the Global Interpreter Lock (GIL) in Python would otherwise be a bottleneck.

Here's a basic overview of how to use the multiprocessing module:

**Key Concepts**

* Process: An independent sequence of execution. Each process has its own memory space.
* Pool: A convenient way to parallelize the execution of a function across multiple input values.
* Queue: Allows safe exchange of information between processes.
* Pipe: Another way to allow processes to communicate.
* Manager: Helps in sharing state between processes.

**Basic Example**

Here’s a simple example of using the multiprocessing module to run a function in parallel using multiple processes.

import multiprocessing

def worker(num):

"""Thread worker function"""

print(f'Worker: {num}')

if \_\_name\_\_ == '\_\_main\_\_':

jobs = []

for i in range(5):

p = multiprocessing.Process(target=worker, args=(i,))

jobs.append(p)

p.start()

**Using Pool**

The Pool class provides a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism).

import multiprocessing

def square(x):

return x \* x

if \_\_name\_\_ == '\_\_main\_\_':

with multiprocessing.Pool(4) as pool:

result = pool.map(square, range(10))

print(result)

**Using Queue**

The Queue class provides a thread- and process-safe way to exchange information between processes.

import multiprocessing

import time

def worker(q):

"""Thread worker function"""

time.sleep(1)

q.put('Task done')

if \_\_name\_\_ == '\_\_main\_\_':

q = multiprocessing.Queue()

p = multiprocessing.Process(target=worker, args=(q,))

p.start()

print(q.get()) # Will print 'Task done'

p.join()

Using Manager

The Manager class allows you to share state between processes.

import multiprocessing

def worker(d, key, value):

d[key] = value

if \_\_name\_\_ == '\_\_main\_\_':

with multiprocessing.Manager() as manager:

d = manager.dict()

jobs = []

for i in range(5):

p = multiprocessing.Process(target=worker, args=(d, i, i\*i))

jobs.append(p)

p.start()

for job in jobs:

job.join()

print(d)

**Summary**

* Process: Use multiprocessing.Process for running a function in a new process.
* Pool: Use multiprocessing.Pool for simple parallel execution of a function over a list of arguments.
* Queue: Use multiprocessing.Queue for safe inter-process communication.
* Manager: Use multiprocessing.Manager for sharing state between processes.

By using these tools, you can effectively perform parallel computation in Python, making your programs more efficient, especially for CPU-bound tasks.

## **Global Interpreter Lock or**[**GIL**](https://wiki.python.org/moin/GlobalInterpreterLock)**:**

The Python Global Interpreter Lock or [GIL](https://wiki.python.org/moin/GlobalInterpreterLock), in simple words, is a mutex (or a lock) that allows only one [thread](https://realpython.com/intro-to-python-threading/) to hold the control of the Python interpreter.

This means that only one thread can be in a state of execution at any point in time. The impact of the GIL isn’t visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code.

Since the GIL allows only one thread to execute at a time even in a multi-threaded architecture with more than one CPU core, the GIL has gained a reputation as an “infamous” feature of Python.

## **Multi-Threading:**

Just like [multiprocessing](https://www.geeksforgeeks.org/multiprocessing-python-set-1/), multithreading is a way of achieving multitasking. In multithreading, the concept of threads is used. Let us first understand the concept of thread in computer architecture.

## **What is a Process in Python?**

In computing, a [process](https://www.geeksforgeeks.org/introduction-of-process-management/) is an instance of a computer program that is being executed. Any process has 3 basic components:

An executable program.

The associated data needed by the program (variables, workspace, buffers, etc.)

The execution context of the program (State of the process)

## **An Intro to Python Threading**

A thread is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System). In simple words, a thread is a sequence of such instructions within a program that can be executed independently of other code.