Python Threads

Sidheswar Routray
Department of Computer Science & Engineering
School of Technology, PDEU

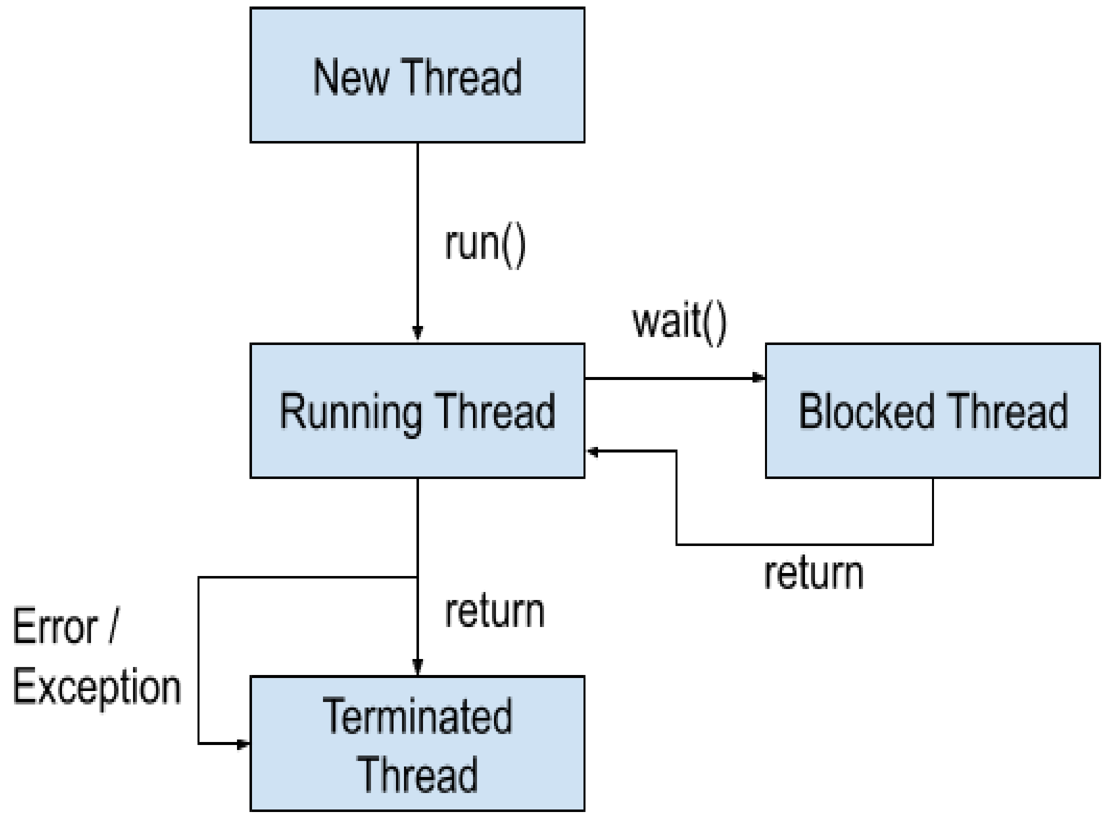
What Are Threads

- Thread: The operating system object that executes the instructions of a process.
- Threading is a sequence of instructions in a program that can be executed independently of the remaining process.
- When we run a Python script, it starts an instance of the Python interpreter that runs our code in the main thread. The main thread is the default thread of a Python process.

Life-Cycle of a Thread

- A thread in Python is represented as an instance of the threading.Thread class.

1. New Thread.
2. Running Thread.
    1. Blocked Thread (optional).
3. Terminated Thread.



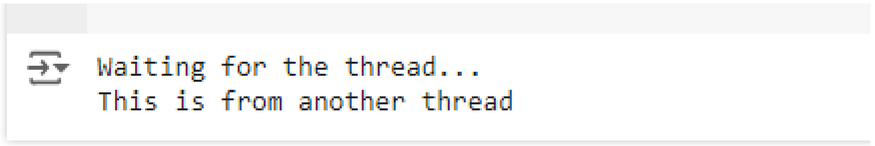
How to Run a Function In a Thread

1.Create an instance of the **threading.Thread** class.

2.Specify the name of the function via the “**target**” argument.

3.Call the **start()** function.

```
# example of running a function in another thread
from time import sleep
from threading import Thread
 # a custom function that blocks for a moment
def task():
    # block for a moment
    sleep(1)
    # display a message
    print('This is from another thread')
 # create a thread
thread = Thread(target=task)
# run the thread
thread.start()
# wait for the thread to finish
print('Waiting for the thread...')
thread.join()
```



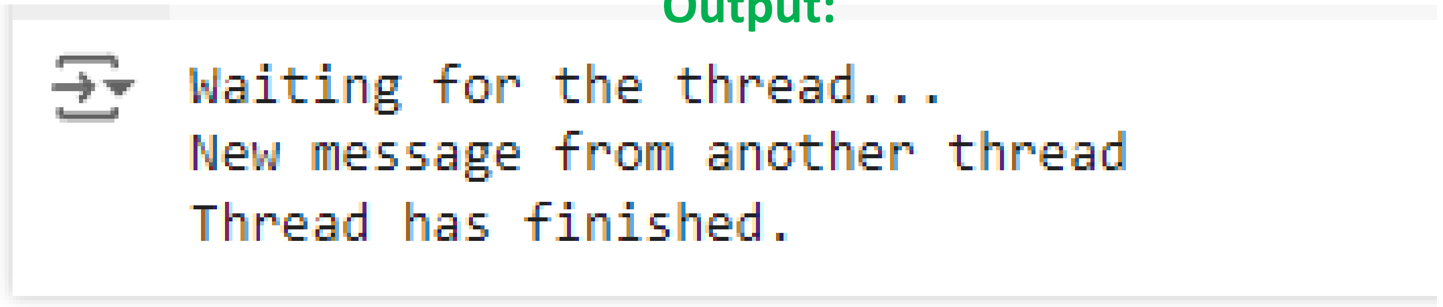
```
Waiting for the thread...
This is from another thread
```

- Imports:sleep from the time module: Pauses the program for a specified number of seconds.
- Thread from the threading module: Allows you to run tasks concurrently.
- Function task:This function sleeps for 1 second and then prints a message.
- Thread Creation:Thread(target=task): Creates a thread that will run the task function.
- thread.start(): Starts the thread and runs the task function concurrently with the main program.
- thread.join(): Waits for the thread to complete before continuing with the rest of the program.
- Output:The program will print "Waiting for the thread..." immediately, and after 1 second, it will print "This is from another thread."

Running a Function in a Thread With Arguments

```python
# example of running a function with arguments in another thread
from time import sleep
from threading import Thread
 # a custom function that blocks for a moment
def task(sleep_time, message):
    # block for a moment
    sleep(sleep_time)
    # display a message
    print(message)
 # create a thread
thread = Thread(target=task, args=(1.5, 'New message from another thread'))
# run the thread
thread.start()
# wait for the thread to finish
print('Waiting for the thread...')
thread.join()
```

Output:



```
Waiting for the thread...
New message from another thread
Thread has finished.
```

Running a Function in a Thread With Arguments

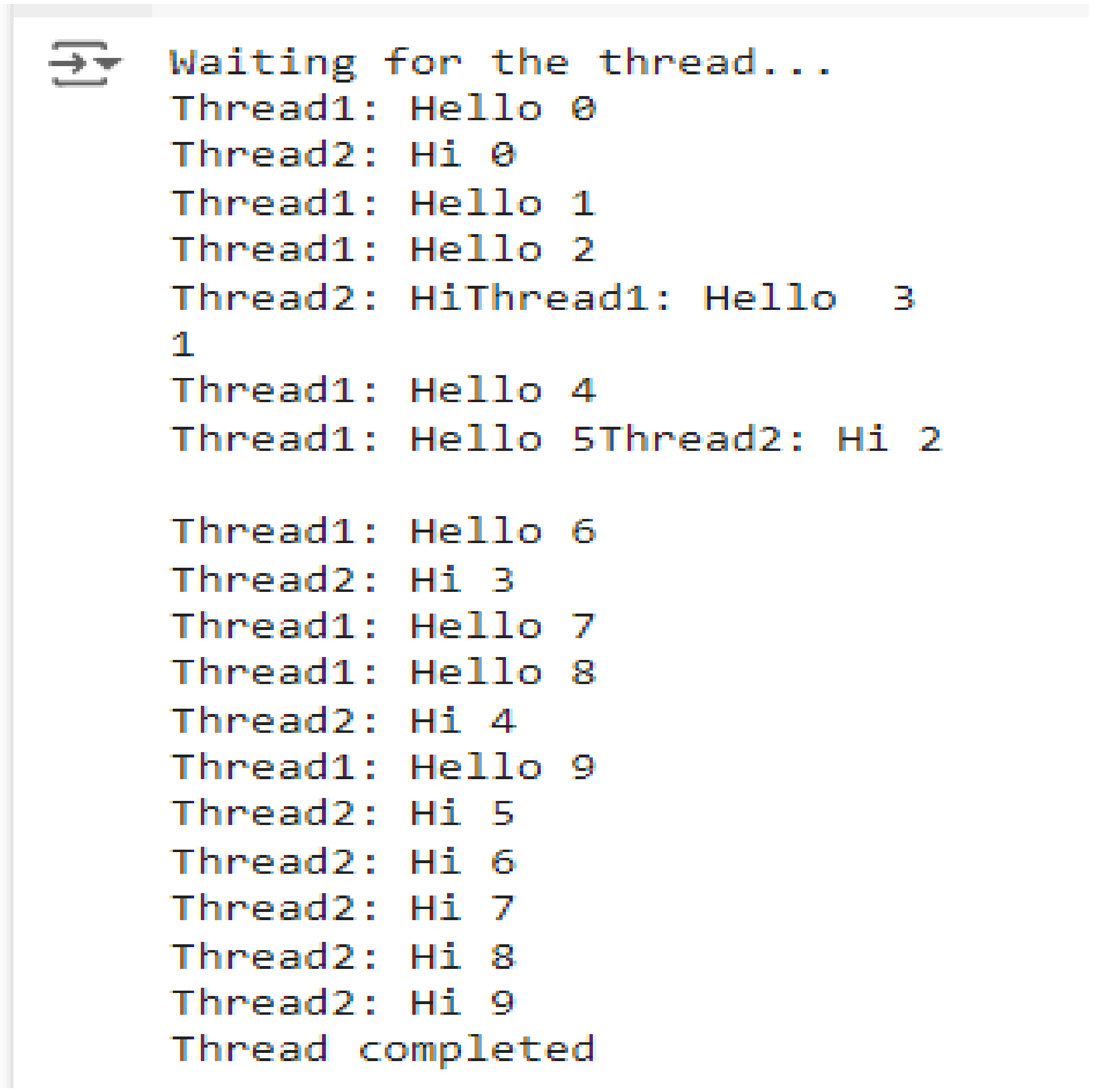
- Thread(target=task, args=(1.5, 'New message from another thread')): Creates a new thread to run the task function, passing the arguments 1.5 and 'New message from another thread'.
- thread.start(): Starts the thread, which runs the task function in parallel to the main thread.
- thread.join(): Waits for the thread to complete execution before the main thread continues.
- You'll first see the "Waiting for the thread..." message, then the thread will finish after 1.5 seconds and print "New message from another thread," followed by "Thread has finished."

Creating multiple Thread

```
from time import sleep
from threading import Thread
 # a custom function that blocks for a moment
def task1(sleep_time, message):
    # block for a moment
    # display a message
    for i in range(10):
        sleep(sleep_time)
        print("Thread1: "+message,i)
def task2(sleep_time, message):
    # block for a moment
    # display a message
    for i in range(10):
        sleep(sleep_time)
        print("Thread2: "+message,i)
```

```
# create a thread
thread1 = Thread(target=task1, args=(1.5,
'Hello'))
thread2 = Thread(target=task2, args=(3,
'Hi'))
# run the thread
thread1.start()
thread2.start()
# wait for the thread to finish
print('Waiting for the thread...')
thread1.join()
thread2.join()
print('Thread completed')
```

Creating multiple Thread



```
Waiting for the thread...
Thread1: Hello 0
Thread2: Hi 0
Thread1: Hello 1
Thread1: Hello 2
Thread2: HiThread1: Hello  3
1
Thread1: Hello 4
Thread1: Hello 5Thread2: Hi 2

Thread1: Hello 6
Thread2: Hi 3
Thread1: Hello 7
Thread1: Hello 8
Thread2: Hi 4
Thread1: Hello 9
Thread2: Hi 5
Thread2: Hi 6
Thread2: Hi 7
Thread2: Hi 8
Thread2: Hi 9
Thread completed
```

Creating multiple Thread

- sleep: This function from the time module is used to pause the execution of the program for a specified number of seconds.
- Thread: This class from the threading module is used to create and manage threads in Python.
- task1: This function takes two arguments: sleep_time and message. It prints a message with a delay of sleep_time seconds for 10 iterations.
- task2: Similar to task1, this function also takes sleep_time and message as arguments and performs a similar operation but prints a different label ("Thread2").
- Thread(target=task1, args=(1.5, 'Hello')):target=task1: Specifies the function to run in the thread.args=(1.5, 'Hello'): Arguments passed to the task1 function.
- Thread(target=task2, args=(3, 'Hi')):target=task2: Specifies the function to run in the thread.args=(3, 'Hi'): Arguments passed to the task2 function.
- thread1.start(): Begins execution of task1 in a separate thread.
- thread2.start(): Begins execution of task2 in a separate thread.
- print('Waiting for the thread...'): Outputs a message indicating that the main thread is waiting for the other threads to complete.
- thread1.join(): Waits for thread1 to finish its execution before proceeding.
- thread2.join(): Waits for thread2 to finish its execution before proceeding
- print('Thread completed'): Outputs a message indicating that all threads have finished their execution.

Custom thread

```
# example of extending the Thread class
from time import sleep
from threading import Thread
 # custom thread class
class CustomThread(Thread):
    # override the run function
    def run(self):
        # block for a moment
        sleep(1)
        # display a message
        print('This is coming from another thread')
 # create the thread
thread = CustomThread()
# start the thread
thread.start()
# wait for the thread to finish
print('Waiting for the thread to finish')
thread.join()
```

Output:

```
Waiting for the thread to finish
This is coming from another thread
Thread has finished executing
```

Extending the Thread Class and Returning Values

```
# example of extending the Thread class and return values
from time import sleep
from threading import Thread
 # custom thread class
class CustomThread(Thread):
    # override the run function
    def run(self):
        # block for a moment
        sleep(1)
        # display a message
        print('This is coming from another thread')
        # store return value
        self.value = 99
```

```
# create the thread
thread = CustomThread()
# start the thread
thread.start()
# wait for the thread to finish
print('Waiting for the thread to finish')
thread.join()
# get the value returned from run
value = thread.value
print(f'Got: {value}')
```

Output:

```
Waiting for the thread to finish
This is coming from another thread
Got: 99
```

Extending the Thread Class and Returning Values

- CustomThread inherits from the Thread class and overrides the run method to perform a task (sleep for 1 second, print a message, and store a value in self.value).
- After creating and starting the thread, the main thread waits for it to finish using join().
- Once the thread is complete, the value stored in self.value is accessed from the main thread.

Problem Statement-1

- Create a simple program to sum of prime numbers from 2 to 1million. Find time to perform this operation.
- Create four threads to divide this task into four parts. One thread 2 to quarter million then another thread quarter to half and so on. Then find the time to perform the task.

Solution

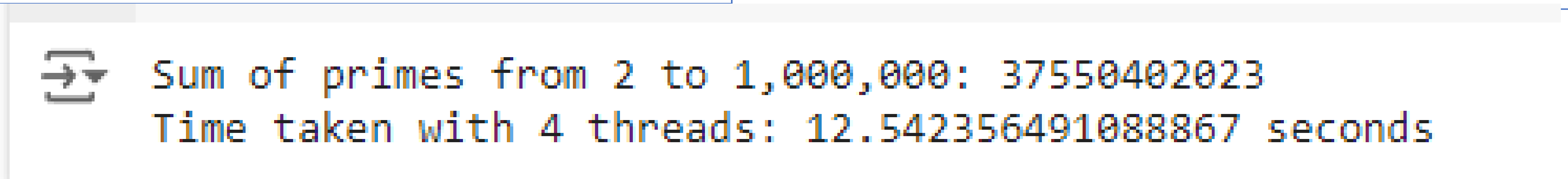
```python
from threading import Thread
import time
import math
# Helper function to check if a number is prime
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True
```

```python
# Function to calculate the sum of primes in a given range
def sum_of_primes(start, end, result, index):
    prime_sum = 0
    for num in range(start, end + 1):
        if is_prime(num):
            prime_sum += num
    result[index] = prime_sum

# Main function to divide the task among threads
def main():
    start_time = time.time()
        # Define the range for each thread
    ranges = [(2, 250000), (250001, 500000),
(500001, 750000), (750001, 1000000)]
```

```
# To store the results from each
thread
    result = [0] * 4
        # Create threads
    threads = []
    for i, (start, end) in
enumerate(ranges):
        thread =
Thread(target=sum_of_primes,
args=(start, end, result, i))
        threads.append(thread)
        thread.start()
```

```
# Wait for all threads to complete
    for thread in threads:
        thread.join()
    # Calculate the total sum of primes
    total_sum = sum(result)
        end_time = time.time()
        # Display the result and time taken
    print(f"Sum of primes from 2 to
1,000,000: {total_sum}")
    print(f"Time taken with 4 threads:
{end_time - start_time} seconds")
if __name__ == "__main__":
    main()
```



```
Sum of primes from 2 to 1,000,000: 37550402023
Time taken with 4 threads: 12.542356491088867 seconds
```

Problem Statement-1

Function Explanation:

- is_prime(n)Parameters:n: The number you want to check if it's prime.
- Logic:Check if n is less than 2:If n is less than 2, the function returns False because numbers less than 2 (like 0 and 1) are not prime.
- Iterate through possible divisors:
- The loop checks for divisors from 2 up to the square root of n. This is efficient because if a number n has a divisor larger than its square root, it will have a corresponding smaller divisor.
- Check divisibility:If n is divisible by any number in this range (meaning n % i == 0), it's not prime, so the function returns False.
- Return True if no divisors were found:
- If the loop completes without finding any divisors, the function returns True, meaning n is prime.
- Example:is_prime(11) will return True because 11 is a prime number.
- is_prime(12) will return False because 12 is divisible by numbers other than 1 and itself.

Problem Statement-1

- sum_of_primes(start, end, result, index):This function calculates the sum of prime numbers within a given range (from start to end).
- The result is stored in the result list at the position defined by index.
- This ensures that each thread can store its result in the right place, allowing the main function to aggregate the results later.
- Parameters:start and end: Define the range in which to calculate prime numbers.
- result: A shared list where each thread's result will be stored.
- index: The position in the result list where the sum of primes for the current thread is stored.
- The main() function sets up the ranges and assigns each range to a different thread.
- The ranges specified in this case are (2, 250000), (250001, 500000), (500001, 750000), and (750001, 1000000), which means the sum of primes will be computed for numbers within these ranges, and the task is split among four threads.

Problem Statement-1

- This initializes a list result with 4 elements, all set to 0. The list is used to store the sum of primes calculated by each thread. Each thread will be assigned an index (from 0 to 3) corresponding to the 4 ranges defined earlier.
- threads = []: An empty list is created to hold all the threads.
- For Loop:The loop iterates over the ranges list and creates a thread for each range.
- enumerate(ranges): This provides both the index (i) and the range (start, end) for each iteration. The index i helps to store the result in the correct position in the result list.
- Starting the Thread:thread.start(): This starts the thread, and it begins executing sum_of_primes() in parallel with other threads.
- Each thread immediately starts working on its assigned range without waiting for the other threads to finish.

Problem Statement-1

Here's a Python program to calculate the sum of prime numbers from 2 to 1 million, utilizing four threads to divide the task into four ranges:

- Thread 1: 2 to 250,000
- Thread 2: 250,001 to 500,000
- Thread 3: 500,001 to 750,000
- Thread 4: 750,001 to 1,000,000
- The is_prime() function checks whether a number is prime.
- The sum_of_primes() function computes the sum of primes within a specific range and stores the result in a shared list result at the given index.
- The main() function defines the ranges for each thread and creates four threads to sum primes in parallel.
- The total sum is calculated by combining the results from all threads.

Problem Statement-2

- Write a program to read file content of multiple file present in folder named as 1.txt, 2.txt ……12.txt. Calculate time for performing this operation.
- Write a program using four thread to read files from same folder. First thread will read 1- 3.txt then second 4-6.txt and so on. Calculate time to read the files.

Solution

```
import time
import os
def read_files_sequentially():
    start_time = time.time()
    folder_path = 'your_folder_path'  # Replace with the path to your folder containing the txt files
    for i in range(1, 13):
        file_name = f"{i}.txt"
        file_path = os.path.join(folder_path, file_name)
        try:
            with open(file_path, 'r') as file:
                content = file.read()
```

```
# You can process the content here if needed
                # For this example, we'll just pass
                pass
        except FileNotFoundError:
            print(f"File {file_name} not found.")
    end_time = time.time()
    total_time = end_time - start_time
    print(f"Sequential reading time: {total_time:.4f} seconds")
if __name__ == "__main__":
    read_files_sequentially()
```

```
import time
import os
from threading import Thread
def read_files_in_range(start, end, folder_path):
    for i in range(start, end + 1):
        file_name = f"{i}.txt"
        file_path = os.path.join(folder_path, file_name)
        try:
            with open(file_path, 'r') as file:
                content = file.read()
```

```
# Process the content if needed
                pass
        except FileNotFoundError:
            print(f"File {file_name} not found by thread {Thread.current_thread().name}.")
def read_files_with_threads():
    start_time = time.time()
    folder_path = 'your_folder_path'  # Replace with the path to your folder containing the txt files
    # Define ranges for each thread
    ranges = [(1, 3), (4, 6), (7, 9), (10, 12)]
    threads = []
    for idx, (start_range, end_range) in enumerate(ranges):
        thread =
Thread(target=read_files_in_range, args=(start_range, end_range, folder_path), name=f"Thread-{idx+1}")
        threads.append(thread)
        thread.start()
    # Wait for all threads to finish
    for thread in threads:
        thread.join()
    end_time = time.time()
    total_time = end_time - start_time
    print(f"Multithreaded reading time: {total_time:.4f} seconds")
if __name__ == "__main__":
    read_files_with_threads()
```

Thread Name

```
# example of accessing the thread
name
from threading import Thread
# create the thread
thread = Thread()
# report the thread name
print(thread.name)
```

```
# example of setting the thread name in the
constructor
from threading import Thread
# create a thread with a custom name
thread = Thread(name='MyThread')
# report thread name
print(thread.name)
```

```
# example of setting the thread name via the property
from threading import Thread
# create a thread
thread = Thread()
# set the name
thread.name = 'MyThread'
# report thread name
print(thread.name)
```

Thread Daemon

- A daemon thread is a background thread.
- The ideas is that backgrounds are like “*daemons*” or spirits (from the ancient Greek) that do tasks for you in the background.

```
# example of setting a thread to be a daemon via the constructor
from threading import Thread
# create a daemon thread
thread = Thread(daemon=True)
# report if the thread is a daemon
print(thread.daemon)
```

```
# example of setting a thread to be a daemon via the property
from threading import Thread
# create a thread
thread = Thread()
# configure the thread to be a daemon
thread.daemon = True
# report if the thread is a daemon
print(thread.daemon)
```

thread.is_alive(): check if a Thread is alive

```
# example of assessing whether a running thread is
alive
from time import sleep
from threading import Thread
# create the thread
thread = Thread(target=lambda:sleep(1))
# report the thread is alive
print(thread.is_alive())
# start the thread
thread.start()
# report the thread is alive
print(thread.is_alive())
# wait for the thread to finish
thread.join()
# report the thread is alive
print(thread.is_alive())
```

Thread Identifier

```
# example of reporting the thread identifier
from threading import Thread
# create the thread
thread = Thread()
# report the thread identifier
print(thread.ident)
# start the thread
thread.start()
# report the thread identifier
print(thread.ident)
```