Thread-based parallelism in Python

A multi-threaded program consists of sub-programs each of which is handled separately by different threads. Multi-threading allows for parallelism in program execution. All the active threads run concurrently, sharing the CPU resources effectively and thereby, making the program execution faster. Multi-threading is generally used when:

1. There are sub-programs whose output needs to be combined by the main program.
2. The main program contains sections of code that are relatively independent of each other.

A multi-threaded program handles different tasks at the same time, within the same process, where different threads share the data space with each other as well as the main thread.

**Starting a new thread**

The threading module in python provides function calls that is used to create new threads. The \_\_init\_\_ function is used for initializing the data associated with the new threads whereas, the run function defines the thread’s behavior as soon as the thread starts it’s execution.

In order to create a new thread, :

1. Create a sub class of the thread class.
2. Override the \_\_init\_\_ function of the thread class. This method will initialize the date specific to a thread.
3. Override the run method to define the behavior of the thread.

|  |
| --- |
| # Python program to demonstrate  # initializing a new thread  import threading    class thread(threading.Thread):      def \_\_init\_\_(self, thread\_name, thread\_ID):          threading.Thread.\_\_init\_\_(self)          self.thread\_name = thread\_name          self.thread\_ID = thread\_ID      def run(self):          print(str(self.thread\_name) +"  "+ str(self.thread\_ID));    thread1 = thread("PL", 1000)  thread2 = thread("python", 2000);    thread1.start()  thread2.start()    print("Exit") |

**Output:**

PL 1000

python 2000

Exit

**Threading module**

Threading module in python provides powerful and high level support for threads.

Threading module defines the following function calls that is used to obtain thread related data. All these functions are executed atomically.

1. **active\_count():** Return the number of Thread objects currently alive. The returned count is equal to the length of the list returned by enumerate().  
   **Syntax:**

**threading.active\_count()**

1. **current\_thread():**Return the current Thread object, corresponding to the caller’s thread of control. If the caller’s thread of control was not created through the threading module, a dummy thread object with limited functionality is returned.  
   **Syntax:**

**threading.current\_thread()**

1. **get\_ident():** Return the ‘thread identifier’ of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.  
   **Syntax:**

**threading.get\_ident()**

1. **enumerate():** Return a list of all Thread objects currently alive. The list includes daemonic threads, dummy thread objects created by current\_thread(), and the main thread. It excludes terminated threads and threads that have not yet been started.  
   **Syntax:**

**threading.enumerate()**

1. **main\_thread():**Return the main Thread object. In normal conditions, the main thread is the thread from which the Python interpreter was started.  
   **Syntax:**

**threading.main\_thread()**

1. **settrace(func):** Set a trace function for all threads started from the threading module. The func will be passed to sys.settrace() for each thread, before its run() method is called.  
   **Syntax:**

**threading.settrace(func)**

1. **setprofile(func):**Set a profile function for all threads started from the threading module. The func will be passed to sys.setprofile() for each thread, before its run() method is called.  
   **Syntax:**

**threading.setprofile(func)**

1. **stack\_size([size]):**Return the thread stack size used when creating new threads.  
   **Syntax:**

**threading.stack\_size([size])**

This module also includes the constant:

* **TIMEOUT\_MAX:** The maximum value allowed for the timeout parameter of blocking functions (Lock.acquire(), RLock.acquire(), Condition.wait(), etc.). Specifying a timeout greater than this value will raise an OverflowError.  
  **Syntax:**

**threading.TIMEOUT\_MAX**

|  |
| --- |
| # Python program to demonstrate  # threading module  import threading    def trace\_function():      print("Passing the trace function")  def profile():      print("Setting the profile of thread: " + str(threading.current\_thread().getName()))    class thread(threading.Thread):      def \_\_init\_\_(self, thread\_name, thread\_ID):          threading.Thread.\_\_init\_\_(self)          self.thread\_name = thread\_name          self.thread\_ID = thread\_ID      def run(self):          print(str(self.thread\_ID));          print("Number of active threads: "+ str(threading.active\_count()))          print("Name of current thread: " + str(threading.current\_thread().getName()))        thread1 = thread("PL", 1000)  thread2 = thread("Python", 2000);  print("Name of main thread: " + str(threading.main\_thread().getName()))  print("Identity of main thread: "+ str(threading.get\_ident()))  print("Stack size = " + str(threading.stack\_size()))  print(threading.settrace(trace\_function()))  threading.setprofile(profile())      thread1.start()  thread2.start()  print("Enumeration list: ")  print(threading.enumerate())  print("Exit") |

**Output:**

Name of main thread: MainThread

Identity of main thread: 139964150720320

Stack size = 0

Passing the trace function

None

Setting the profile of thread: MainThread

1000

Number of active threads: 2

Name of current thread: Thread-1

2000

Number of active threads: 2

Name of current thread: Thread-2

Enumeration list:

[]

Exit