

Les Threads en Python

Introduction

Les bases

INTRODUCTION

Python

Threads

Cycle de vie de threads et programmation :
création, lancement, terminaison, ...

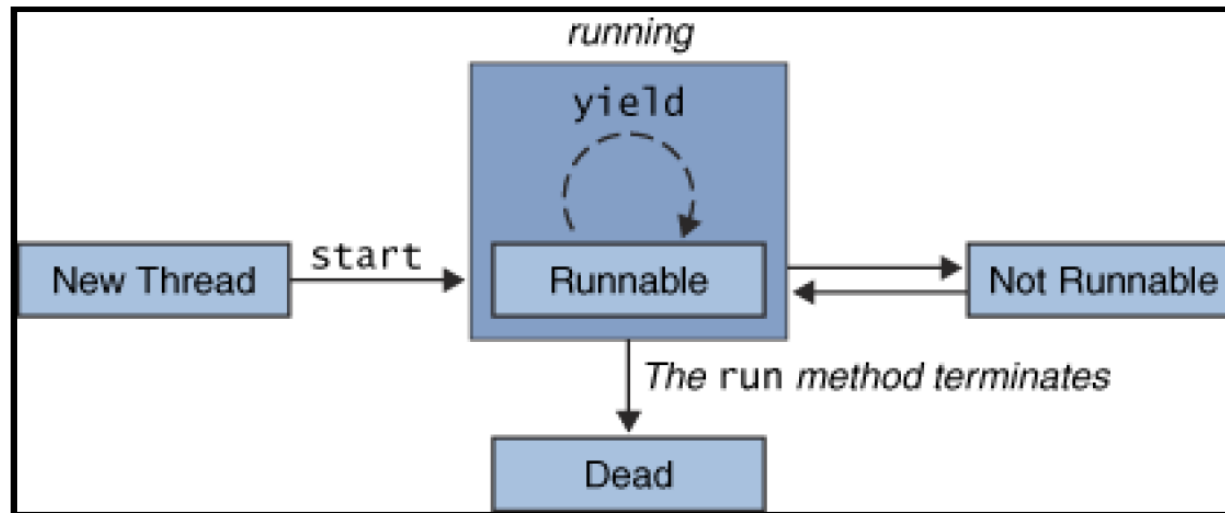
Module threading : `create()`, `start()`, `join()`,
`current_thread()`, `main_thread()`, `enumerate()`, ...

Définition locale vs classe (héritage classe
`Thread`, surcharge de `__init__()` et `run()`)



Cycle de vie de threads et programmation :
création, lancement, terminaison, ...

Cycle de vie des threads et programmation :
création, lancement, terminaison, ...



The Thread states

```
import threading
import time
```

```
# A very simple method for our thread to execute
```

```
def threadWorker():
    # it is only at the point where the thread starts executing
    # that it's state goes from 'Runnable' to a 'Running'
    # state
    print("My Thread has entered the 'Running' State")

    # If we call the time.sleep() method then our thread
    # goes into a not-runnable state. We can do no further work
    # on this particular thread
    time.sleep(10)
    # Thread then completes its tasks and terminates
    print("My Thread is terminating")
```

```
# At this point in time, the thread has no state
# it hasn't been allocated any system resources
myThread = threading.Thread(target=threadWorker)
```

```
# When we call myThread.start(), Python allocates the necessary system
# resources in order for our thread to run and then calls the thread's
# run method. It goes from 'Starting' state to 'Runnable' but not running
myThread.start()
```

```
# Here we join the thread and when this method is called
# our thread goes into a 'Dead' state. It has finished the
# job that it was intended to do.
myThread.join()
print("My Thead has entered a 'Dead' state")
```

Thread Lifecycle



tp1_01_threadLifecycle.py



tp1_01_threadLifecycle.py

Breaking it down

In this preceding code example, we define a function, `threadWorker`, which will be the invocation target of the thread that we will create. All that this `threadWorker` function does is to print out its current state, and then sleep for 10 seconds by calling `time.sleep(10)`.

After we've defined `threadWorker`, we then go on to create a New Thread object in this line:

```
myThread = threading.Thread(target=threadWorker)
```

At this point in time, our thread object is currently in the New Thread state, and hasn't yet been allocated any system resources that it needs to run. This only happens when we go on to call this function:

```
myThread.start()
```

At this point, our thread is allocated with all of its resources, and the thread's `run` function is called. The thread now enters the "Runnable" state. It goes on to print out its own state, and then proceeds to block for 10 seconds by calling `time.sleep(10)`. During the 10 seconds that this thread sleeps, the thread is considered to be in the "Not-Running" state, and other threads will be scheduled to run over this thread.

Finally, once the 10-second period has elapsed, our thread is considered to have ended and be in the "Dead" state. It no longer needs any of the resources that it was allocated, and it will be cleaned up by the garbage collector.

The Python “Thread” class

```
# Python Thread class Constructor
def __init__(self, group=None, target=None, name=None,
             args=(), kwargs=None, verbose=None):
```

- group: special parameter which is reserved for a future extension.
- **target**: the callable object to be invoked by the run() method. If not passed, this will default to None, and nothing will be started.
- **name**: the thread name.
- **args**: the argument tuple for target invocation. It defaults to ().
- kwargs: dictionary of keyword arguments to invoke the base class constructor.



Module threading : create(), start(), join(),
current_thread(), main_thread(), enumerate(), ...

Current and main Threads



tp1_02_current_and_main_threads.py

```
import threading
import time

def myChildThread():
    print("Child Thread Starting")
    time.sleep(5)
    print("Current Thread -----")
    print(threading.current_thread())
    print("-----")
    print("Main Thread -----")
    print(threading.main_thread())
    print("-----")
    print("Child Thread Ending")

child = threading.Thread(target=myChildThread)
child.start()
child.join()
```

Starting a pool of threads



tp1_03_startingPoolOfThreads.py

```
import threading
import time
import random

def executeThread(i):
    print("Thread {} started".format(i))
    sleepTime = random.randint(1,10)
    time.sleep(sleepTime)
    print("Thread {} finished executing".format(i))

for i in range(10):
    thread = threading.Thread(target=executeThread, args=(i,))
    thread.start()

    print("Active Threads:" , threading.enumerate())
```

Passage d'arguments
séparés par des virgules.
Attention à la virgule finale !

Total number of active threads



tp1_04_totalThreads.py

```
import threading
import time
import random

def myThread(i):
    print("Thread {}: started".format(i))
    time.sleep(random.randint(1,5))
    print("Thread {}: finished".format(i))

def main():
    for i in range(random.randint(2,50)):
        thread = threading.Thread(target=myThread, args=(i,))
        thread.start()

    time.sleep(4)
    print("Total Number of Active Threads: {}".format(threading.active_count()))

if __name__ == '__main__':
    main()
```

Getting current Thread in pool



tp1_05_gettingCurrentThreadInPool.py

```
import threading
import time

def threadTarget():
    print("Current Thread: {}".format(threading.current_thread()))

threads = []

for i in range(10):
    thread = threading.Thread(target=threadTarget)
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join()
```

Enumerate threads



tp1_06_enumerateThreads.py

```
import threading
import time
import random

def myThread(i):
    print("Thread {}: started".format(i))
    time.sleep(random.randint(1,5))
    print("Thread {}: finished".format(i))

def main():
    for i in range(4):
        thread = threading.Thread(target=myThread, args=(i,))
        thread.start()

    print("Enumerating: {}".format(threading.enumerate()))

if __name__ == '__main__':
    main()
```

Identifying threads by name



tp1_07_identifyingThreadsByName.py

```
import threading
import time

def myThread():
    print("Thread {} starting".format(threading.currentThread().getName()))
    time.sleep(10)
    print("Thread {} ending".format(threading.currentThread().getName()))

for i in range(4):
    threadName = "Thread-" + str(i)
    thread = threading.Thread(name=threadName, target=myThread)
    thread.start()

print("{}".format(threading.enumerate()))
```

“Killing” a Thread, really?



tp1_08_killThread.py

```
from multiprocessing import Process
import time

def myWorker():
    t1 = time.time()
    print("Process started at: {}".format(t1))
    time.sleep(20)

myProcess = Process(target=myWorker)
print("Process {}".format(myProcess))
myProcess.start()
print("Terminating Process...")
myProcess.terminate()
myProcess.join()
print("Process Terminated: {}".format(myProcess))
```

NOTE :

Erreur de ma part, j’ai mis la classe Process au lieu de Thread !
C’est quasiment la même API, même le but n’est pas le même. J’en reparlerai.

NOTE :

On ne peut pas réellement tuer un thread. Il faut passer par le module multiprocessing. J’en reparlerai.

Définition locale vs classe

- héritage de la classe Thread
- surcharge des méthodes `__init__()` et `run()`

Création d'une classe Thread custom



tp1_09_classThread.py

```
from threading import Thread

class myWorkerThread(Thread):

    def __init__(self):
        print("Hello world")
        Thread.__init__(self)

    def run(self):
        print("Thread is now running")

myThread = myWorkerThread()
print("Created my Thread Object")
myThread.start()
print("Started my thread")
myThread.join()
print("My Thread finished")
```

Héritage de la classe "Thread" :

C'est la solution la plus élégante en terme de développement.

Il s'agit de:

- 1) Hériter de la classe Thread
- 2) Surcharger le constructeur **`__init__`**
 - "***self***" est l'équivalent de this,
c'est toujours le 1er paramètre
 - Il faut toujours appeler la méthode parente :
`Thread.__init__(self)`
- 3) Surcharger la méthode **`run`**.
C'est la fonction qui sera exécutée par le thread.

Les Threads en Python

On peut faire bien plus avec les threads en Python.

On a juste vu les notions de base.