Processes and Threads All Versions

Most programs are executed line by line, only running a single process at a time. Threads allow multiple processes to flow independent of each other. Threading with multiple processors permits programs to run multiple processes simultaneously. This topic documents the implementation and usage of threads in Python.

Examples

Global Interpreter Lock

Python multithreading performance can often suffer due to the **Global Interpreter Lock** . In short, even though you can have multiple threads in a Python program, only one bytecode instruction can execute in parallel at any one time, regardless of the number of CPUs.

As such, multithreading in cases where operations are blocked by external events - like network access - can be quite effective:

```
import threading
import time


def process():
    time.sleep(2)


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))


start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.00s
# Out: Four runs took 2.00s
```

Note that even though each `process` took 2 seconds to execute, the four processes together were able to effectively run in parallel, taking 2 seconds total.

However, multithreading in cases where intensive computations are being done in Python code - such as a lot of computation - does not result in much improvement, and can even be slower than running in parallel:

```
import threading
import time


def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
        result = 0
        for i in range(100000):
            result = otherfunc(result, somefunc(i))


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))


start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.05s
# Out: Four runs took 14.42s
```

In the latter case, multiprocessing can be effective as multiple processes can, of course, execute multiple instructions simultaneously:

```
import multiprocessing
import time


def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
        result = 0
        for i in range(100000):
            result = otherfunc(result, somefunc(i))


start = time.time()
process()
print("One run took %.2fs" % (time.time() - start))


start = time.time()
processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()
print("Four runs took %.2fs" % (time.time() - start))

# Out: One run took 2.07s
# Out: Four runs took 2.30s
```

Running in Multiple Processes

Use `multiprocessing.Process` to run a function in another process. The interface is similar to `threading.Thread` :

```
import multiprocessing
import os

def process():
    print("Pid is %s" % (os.getpid(),))

processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()

# Out: Pid is 11206
# Out: Pid is 11207
# Out: Pid is 11208
# Out: Pid is 11209
```

Running in Multiple Threads

Use `threading.Thread` to run a function in another thread.

```
import threading
import os

def process():
    print("Pid is %s, thread id is %s" % (os.getpid(), threading.current_thread().name))

threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()

# Out: Pid is 11240, thread id is Thread-1
# Out: Pid is 11240, thread id is Thread-2
# Out: Pid is 11240, thread id is Thread-3
# Out: Pid is 11240, thread id is Thread-4
```

Sharing State Between Processes

Code running in different processes do not, by default, share the same data. However, the `multiprocessing` module contains primitives to help share values across multiple processes.

```
import multiprocessing

plain_num = 0
shared_num = multiprocessing.Value('d', 0)
lock = multiprocessing.Lock()

def increment():
    global plain_num
    with lock:
        # ordinary variable modifications are not visible across processes
        plain_num += 1
        # multiprocessing.Value modifications are
        shared_num.value += 1

ps = [multiprocessing.Process(target=increment) for n in range(4)]
for p in ps:
    p.start()
for p in ps:
    p.join()

print("plain_num is %d, shared_num is %d" % (plain_num, shared_num.value))

# Out: plain_num is 0, shared_num is 4
```

Sharing State Between Threads

As all threads are running in the same process, all threads have access to the same data.

However, concurrent access to shared data should be protected with a lock to avoid synchronization issues.

```
import threading

obj = {}
obj_lock = threading.Lock()

def objify(key, val):
    print("Obj has %d values" % len(obj))
    with obj_lock:
        obj[key] = val
    print("Obj now has %d values" % len(obj))

ts = [threading.Thread(target=objify, args=(str(n), n)) for n in range(4)]
for t in ts:
    t.start()
for t in ts:
    t.join()
print("Obj final result:")
import pprint; pprint.pprint(obj)

# Out: Obj has 0 values
# Out:  Obj has 0 values
# Out: Obj now has 1 values
# Out: Obj now has 2 valuesObj has 2 values
# Out: Obj now has 3 values
# Out:
# Out:  Obj has 3 values
# Out: Obj now has 4 values
# Out: Obj final result:
# Out: {'0': 0, '1': 1, '2': 2, '3': 3}
```

Syntax

Parameters

Remarks