# Module 05 – Processes

# multiprocessing

- multiprocessing module is responsible for processes creation and management
- multiprocessing API is similar to threading API and it extends the threading module functionality
- Python processes avoid the Global Interpreter Lock (GIL)and take full advantages of multiple processors on a machine
- multiprocessing module includes a lot of useful classes for — processes creation, synchronization and IPC

# The multiprocessing.Process class

- We can use the Process class to create a process object
- Process(group=None, target=None, name=None, args=())
  - target is the callable object to be invoked by the Process
  - name is the process name
  - args is the argument tuple for the target invocation.
  - group — should be always be None

# Process start and join functions

- start() - Starts the process’s activity.

- join([timeout])

  — Block the calling method until the process whose join() method is called terminates or until the optional timeout occurs.

  — If timeout parameter specified the calling method will be blocked up to timeout seconds. The exitcode will stays None until the process is actually finishes (Will be discussed later)”

# The Process example

Demo

# The Process example

```
from multiprocessing import Process, current_process

def f(val):
    cp = current_process()
    print("In child process: name={}, pid={}, val={}".format(cp.name,
cp.pid, val))

if __name__ == '__main__':
    p = Process(target=f, args=('param_val',))
    p.start()
    p.join()
```

# The Process example — cont’d

- In some platform, like windows, the child process goes through the main space before executing the target function. This is why the creation of child process must be ' f under the: if __name__==“__main__”

- Single-item tuples require a trailing comma:
  - tpl = (1,)

# join with timeout example

```
import multiprocessing as MP
import time
def func():
        for n in range(100000000):
                pass

if __name__ == "__main__":
        p = MP.Process(target=func)
        p.start()

        p.join(1)
        while p.exitcode is None:
        print("in loop")
        p.join(1)
        time.sleep(1)
```

The output:

```
In loop
In loop
In loop
In loop
```

# Exchanging data between processes

- When it comes to communicating between processes, the multiprocessing modules has two primary methods:
  - — Queues
  - — Pipes

# Exchanging data between processes — Queue

- Multiprocessing Queue main functions:
  - put
  - put_nowait
  - get
  - get_nowait
  - empty / full

# Exchanging data between processes — Queue

```
from multiprocessing import Process, Queue
def f(q):
        q.put(["one", "two",3])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get()) #prints (["one", "two",3]
    p.join()
    q.close()
```

# Exchanging data between processes -— Pipe

- Pipe() returns a pair of connection objects connected by a pipe which by default is duplex (two-way)
  - Each connection object has send() and recv() methods
- A Pipe can only have two endpoints
- A Pipe is much faster

# Exchanging data between processes -— Pipe

```
from multiprocessing import Process, Pipe

def f(conn):
        conn.send(['hello’, 'world’])
        conn.close()

if __name__ == '__main__’:
        parent_conn, child_conn = Pipe()
        p = Process(target=f, args=(child_conn,))
        p.start()
        print(parent_conn.recv()) # ['hello’, 'world’]
        p.join()
```

# Synchronization between processes

- Take a look at the following program:

```
import multiprocessing as MP
import sys

def loop():
        for i in range(400):
                sys.stdout.write(str(i)+" ")
                sys.stdout.flush()
                sys.stdout.write("in process ")
                sys.stdout.flush()
                sys.stdout.write(MP.current_process().name+ "\n")
                sys.stdout.flush()
```

# Synchronization between processes — cont’d

```
if __name__ == "__ main__":
        MP.Process(target=loop, name="child1").start()
        MP.Process(target=loop, name="child2").start()

for i in range(400):
    sys.stdout.write(str(i)+" ")
    sys.stdout.flush()
    sys.stdout.write("in process ")
    sys.stdout.flush() sys.stdout.write(MP.current_process().name+ "\n")
    sys.stdout.flush()
```

# Synchronization between processes-cont’d

- multiprocessing module has 3 classes for process synchronization:
  - Lock - non-recursive lock object
  - Rlock - recursive lock object
  - Semaphore — created with internal counter and can be acquired countertimes before released
- They all support context manages and can be used with with statement
- They all have the following function:
  - — acquire(blocking=True, timeout=-1)
    - acquire returns True if the locking were successful and False; otherwise al
  - — release()

# Multiprocessing Pool

Demo

# Multiprocessing Lock example

```
import multiprocessing as mp
import sys

def loop(lock):
    for i in range(400):
        with lock:
            sys.stdout.write(str(i) + " ")
            sys.stdout.flush()

if __name__ == "__main__":
    lock = mp.Lock()
    sys.stdout.write("in process ")
    sys.stdout.flush()
    sys.stdout.write(mp.current_process().name + "\n")
    sys.stdout.flush()
    loop(lock)
```

# Multiprocessing Lock example — cont’d

```
if __name__ == "__main__":
    lock = MP.Lock()
    MP.Process(target=loop, name="child1", args=(lock,)).start()
    MP.Process(target=loop, name="child2", args=(lock,)).start()

    for i in range(400):
        sys.stdout.write(str(i) + " ")
        sys.stdout.flush()

    sys.stdout.write("in process ")
    sys.stdout.flush()
    sys.stdout.write(MP.current_process().name + "\n")
    sys.stdout.flush()
```

# Multiprocessing Pool

- Multiprocessing module contains (among others) a Pool class that can be used for parallelize executing a function across multiple inputs.

- Using a Poo! can be a convenient approach for simple parallel processing tasks. Some of Pool tasks are:
  - pool.map
  - pool.map_unordered
  - pool.imap
  - pool.map_async
  - pool.apply
  - etc

# Multiprocessing Pool

Demo

# Multiprocessing Pool —- example

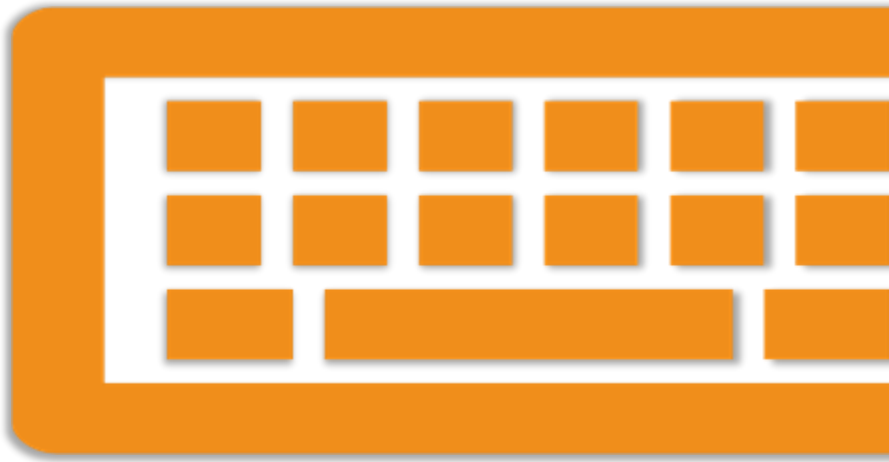
```
from multiprocessing import Pool

def increment(number):
    return number + 1

if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    pool = Pool(processes=3)
    incremented_list = pool.map(increment, numbers)
    print(incremented_list)  # [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

# Lab 01

# Lab



# Questions