MULTITHREADING

THREAD

- ⇨ Thread is a small unit of code which has its own execution part
- ⇨ Thread is a light weight device i.e., it consumes less memory
- ⇨ One thread is independent of another thread
- ⇨ Thread consists of an instructor pointer that holds the current state of the thread and controls what executes next in what order
- ⇨ A single process consists of multiple threads, each thread performs different task
- ⇨ Every process will have one thread that is always running that is “Main Thread”
- ⇨ This main thread creates the child thread objects and these child threads are initiated by the main thread
- ⇨ In python, thread is a class which have several methods:
  - ❖ start()
  - ❖ run()
  - ❖ stop()
  - ❖ resume()
  - ❖ setPriority(int)
  - ❖ getPriority() – returntype is integer
  - ❖ setName(string)
  - ❖ getName() – returntype is string
  - ❖ sleep(seconds)
  - ❖ join(milliseconds)
  - ❖ yield(milliseconds)
  - ❖ wait()

MULTITHREADING

- ⇨ multithreading means, executing multiple threads simultaneously
- ⇨ this multithreading concept is used when threads are independent of each other
- ⇨ ADVANTAGES
  - ❖ It improves the performance
  - ❖ It saves the time
  - ❖ Increased responsiveness
  - ❖ Better resource utilization
  - ❖ Enhanced user experience

CREATING THREAD

There are 3 ways to create a thread:

- ⇨ Without any class
- ⇨ By extending the thread class
- ⇨ Without extending the thread class

1. CREATING THE THREAD WITHOUT USING ANY CLASS
   a. It can be done by using threading module which is built in module

   Ex:

```
import threading
def add(a,b):
    print(a+b)
thread1=threading.Thread(target=add(10,20))
thread1.start()
```

2. CREATING THE THREAD BY EXTENDING THE THREAD CLASS

Ex:

```
from threading import Thread
class Thread1(Thread):
    def run(self):
        print("iam thread1")
thread1=Thread1()
thread1.start()
```

3. CREATING THE THREAD WITHOUT EXTENDING THE THREAD CLASS

Ex:

```
import threading
class thread1:
     def hello(self):
          print("hello iam thread1")
obj=thread1()
th1=threading.Thread(target=obj.hello())
th1.start()
```

THREADING MODULE

⇨ Threading module provides the several methods to work with threads

1. start()
    a. start() method is used to start the thread
    b. each thread should be called only once because if we start the thread twice we get runtime error
2. join(timeout=None)
    a. this method is used to stop the execution of main thread/other thread until another thread completes its execution

    ex:

```
import threading
def numbers():
    for i in range(11):
        print(i)
th1=threading.Thread(target=numbers())
th1.start()
print("main thread is waiting")
th1.join()
print("iam main thread")
```

3. threading.current_thread()
    a. it returns the reference to the current thread object
    b. it represents the thread which is currently running

    ex:

```
from threading import current_thread
print(current_thread().name)
```

4. threading.enumarate()
    a. it returns the list of all the threads that are currently running

    ex:

```
print(threading.active_count())
```

5. is_alive()
    a. it checks whether the thread is currently alive/running or not
    b. returntype is boolean

    ex:

```
print(threading.current_thread().is_alive())
```

6. getName()
    a. it returns the name of the thread
    b. it is depreciated in latest version instead of that we use “name”
    ex:

```
print(current_thread().name)
```

7. setName()
    a. it is used to change the name of the thread
    b. it is depreciated in latest version instead of that we use “name”

    ex:

```
import threading
def hello():
    print("hello ")
th1=threading.Thread(target=hello())
print(th1.name)
th1.name="hello thread"
print(th1.name)
```

8. DEAMON THREADS

⇨ The threads which are running in background is called daemon threads

❖ isDaemon()

a. it is used to check whether the thread is deamon thread or not
b. it is depreciated in latest versions but instead we use “daemon”
c. return type is Boolean

ex:

```
import threading
def numbers():
    for i in range(11):
        print(i)
th1=threading.Thread(target=numbers())
print(th1.daemon)
```

❖ setDaemon()

a. it is used to set the thread as a daemon thread

ex:

```
import threading
def numbers():
    for i in range(11):
        print(i)
th1=threading.Thread(target=numbers())
print(th1.daemon)
th1.daemon=True
print(th1.daemon)
```

MULTITHREADING EXAMPLE

```
import threading
def add(a,b):
    print(a+b)
def sub(a,b):
    print(a-b)
th1=threading.Thread(target=add(10,20))
th2=threading.Thread(target=sub(20,10))
th1.start()
th2.start()
th1.join()
th2.join()
```

THREAD SYNCRONIZATION

- ⇨ the concept of thread synchronization is at a time one thread is allowed
- ⇨ to handle race around conditions, deadlocks, other thread-based issues, we use the “Lock” object
- ⇨ if any thread wants to access a resource, then it will acquire a lock for that resource by using lock object.
- ⇨ Once a thread locks a particular resource, then no thread will not be able to access that resource until its execution is complete or lock is released by another thread/same thread
- ⇨ “Lock” object provides a way to synchronize access to shared resources by allowing only one thread to acquire the lock at a time
- ⇨ Lock is generally in two states i.e., locked or unlocked
- ⇨ It supports two methods:
  - ❖ acquire()
    - ▪ if the lock is in unlocked state, it is used to change unlocked state to locked state
    - ▪ if the lock is in locked state, the acquire() method is blocked until the state changes to unlocked state
  - ❖ release()
    - o it is used to release the lock i.e., it changes the locked state into unlocked state
    - o this method can be called by any thread not only specific by the thread that acquires the lock

ex:

```
from threading import *
balance=0
lockobj=Lock()
amount=0
def deposit(amount):
    global balance
    lockobj.acquire()
try:
    balance+=amount
finally:
    lockobj.release()
def withdraw(amount):
    global balance
    lockobj.acquire()
try:
    if balance>=amount:
          balance-=amount
    else:
       print("insufficient balance")
finally:
    lockobj.release()
thread1=Thread(target=deposit,args=(1000,))
thread2=Thread(target=withdraw,args=(500,))
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print(f"after transaction={balance}")
```