Video 26 : Threads

This video will focus 100% on threads. We'll learn about sleep(), strftime(), the Threading Module, Creating Threads, activeCount(), enumerate(), Subclassing Threads, run(), start(), is_alive(), getName(), setName(), join(), Synchronizing Threads, acquire(), release(), Lock() and more.

When you use threads it is like you are running multiple programs at once.

Threads actually take turns executing. While one executes the other sleeps until it is its turn to execute. Here is an example.

CODE

```
import threading
import time
import random

def execute_thread(i):

    # strftime or string formatted time allows you to
    # define how the time is displayed.
    # You could include the date with
    # strftime("%Y-%m-%d %H:%M:%S", gmtime())

    # Print when the thread went to sleep
    print("Thread {} sleeps at {}".format(i,
                time.strftime("%H:%M:%S", time.gmtime())))

    # Generate a random sleep period of between 1 and
    # 5 seconds
    rand_sleep_time = random.randint(1, 5)

    # Pauses execution of code in this function for
    # a few seconds
    time.sleep(rand_sleep_time)

    # Print out info after the sleep time
    print("Thread {} stops sleeping at {}".format(i,
                time.strftime("%H:%M:%S", time.gmtime())))

for i in range(10):

    # Each time through the loop a Thread object is created
    # You pass it the function to execute and any
    # arguments to pass to that method
    # The arguments passed must be a sequence which
    # is why we need the comma with 1 argument
    thread = threading.Thread(target=execute_thread, args=(i,))
    thread.start()

    # Display active threads
    # The extra 1 is this for loop executing in the main
    # thread
    print("Active Threads :", threading.activeCount())

    # Returns a list of all active thread objects
    print("Thread Objects :", threading.enumerate())
```

Subclassing Threads

You can subclass the Thread object and then define what happens each time a new thread is executed or run.

CODE

```
class CustThread(threading.Thread):

    def __init__(self, name):
        threading.Thread.__init__(self)

        self.name = name

    def run(self):

        get_time(self.name)

        print("Thread", self.name, "Execution Ends")

def get_time(name):
    print("Thread {} sleeps at {}".format(name,
                time.strftime("%H:%M:%S", time.gmtime())))

    randSleepTime = random.randint(1, 5)

    time.sleep(randSleepTime)

# Create thread objects
thread1 = CustThread("1")
thread2 = CustThread("2")

# Start thread execution of run()
thread1.start()
thread2.start()

# Check if thread is alive
print("Thread 1 Alive :", thread1.is_alive())
print("Thread 2 Alive :", thread2.is_alive())

# Get thread name
# You can change it with setName()
print("Thread 1 Name :", thread1.getName())
print("Thread 2 Name :", thread2.getName())

# Wait for threads to exit
thread1.join()
```

```
thread2.join()

print("Execution Ends")
```

Synchronizing Threads

You can lock other threads from executing. If we try to model a bank account we have to make sure the account is locked down during a transaction so that if more then 1 person tries to withdrawal money at once we don't give out more money than is in the account.

```
class BankAccount (threading.Thread):

    acct_balance = 100

    def __init__(self, name, money_request):
        threading.Thread.__init__(self)
        self.name = name
        self.money_request = money_request

    def run(self):
        # Get lock to keep other threads from accessing the account
        threadLock.acquire()

        # Call the static method
        BankAccount.get_money(self)

        # Release lock so other thread can access the account
        threadLock.release()

    @staticmethod
    def get_money(customer):
        print("{} tries to withdrawal ${} at {}".format(customer.name,
                customer.money_request,
                time.strftime("%H:%M:%S", time.gmtime())))

        if BankAccount.acct_balance - customer.money_request > 0:
            BankAccount.acct_balance -= customer.money_request
            print("New account balance is : ${}”.format(BankAccount.acct_balance))
        else:
            print("Not enough money in the account")
            print("Current balance : ${}”.format(BankAccount.acct_balance))

        time.sleep(3)

# Create a lock to be used by threads
threadLock = threading.Lock()

# Create new threads
doug = BankAccount("Doug", 1)
paul = BankAccount("Paul", 100)
sally = BankAccount("Sally", 50)

# Start new Threads
```

```
doug.start()
paul.start()
sally.start()

# Have threads wait for previous threads to terminate
doug.join()
paul.join()
sally.join()

print("Execution Ends")
```

That’s all for now. In the next video I’ll start my multipart tutorial on Regular Expressions.