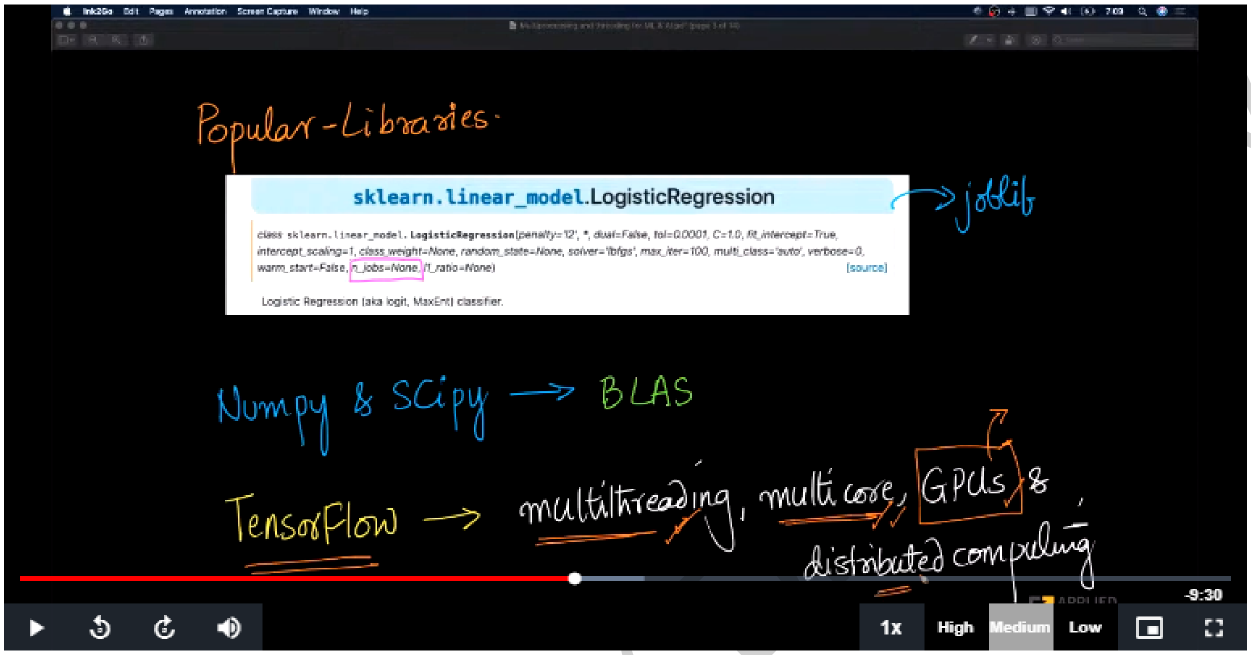
17.1 Applied Introduction and Overview



Popular-Libraries:

sklearn.linear_model.LogisticRegression

```
class sklearn.linear_model.LogisticRegression(penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True,
intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0,
warm_start=False, n_jobs=None, l1_ratio=None)
```

[source]

Logistic Regression (aka logit, MaxEnt) classifier.

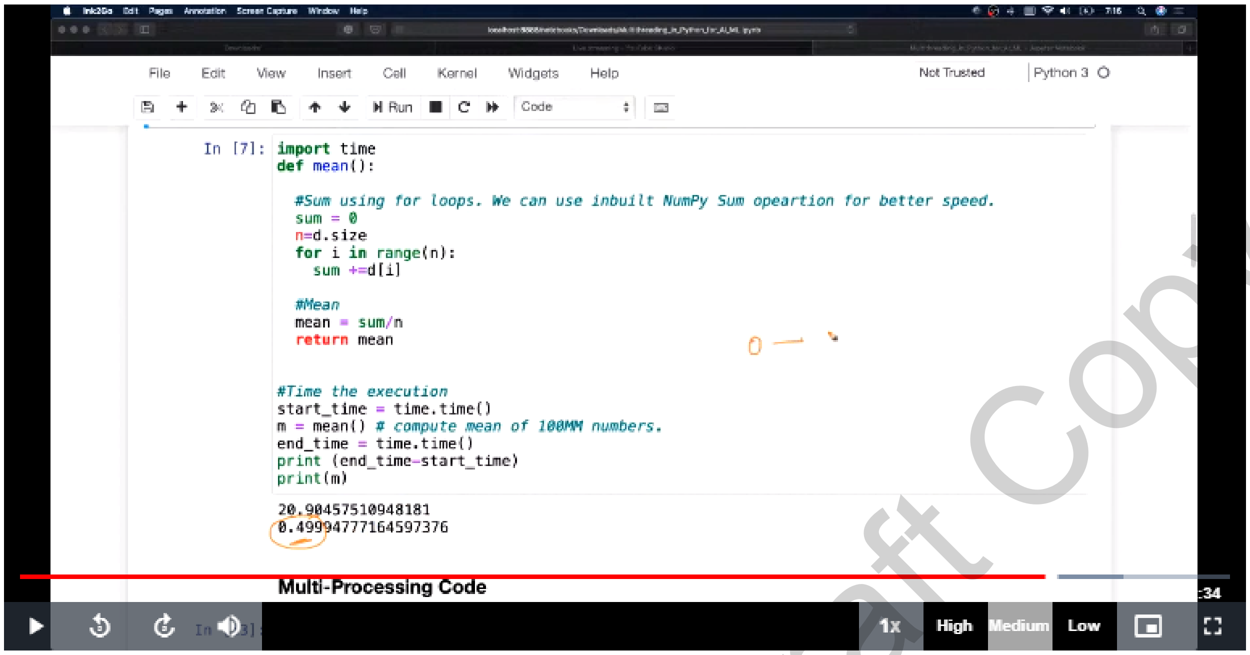
→ joblib

Numpy & SCipy → BLAS

TensorFlow → multithreading, multi core, GPUs & distributed computing

Timestamp: 8:02

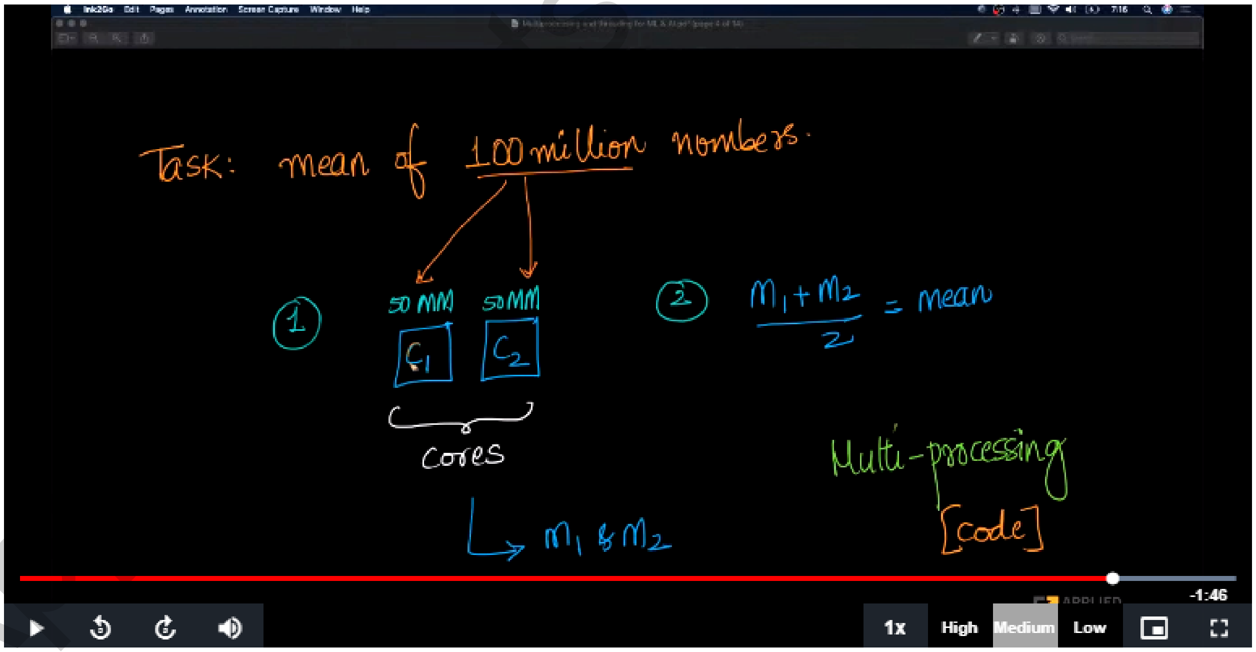
Many popular machine learning libraries like sklearn, numpy, scipy and tensorflow uses some form of parallelization libraries. Sklearn uses joblib, numpy and script internally uses BLAS. Tensorflow can perform tasks using multithreading, multicore, it can even use GPUs and distributed computing.



```
In [7]: import time
        def mean():

          #Sum using for loops. We can use inbuilt NumPy Sum opeartion for better speed.
          sum = 0
          n=d.size
          for i in range(n):
            sum +=d[i]

          #Mean
          mean = sum/n
          return mean


        #Time the execution
        start_time = time.time()
        m = mean() # compute mean of 100MM numbers.
        end_time = time.time()
        print (end_time-start_time)
        print(m)

        20.90457510948181
        0.49994777164597376
```

Timestamp: 14:58

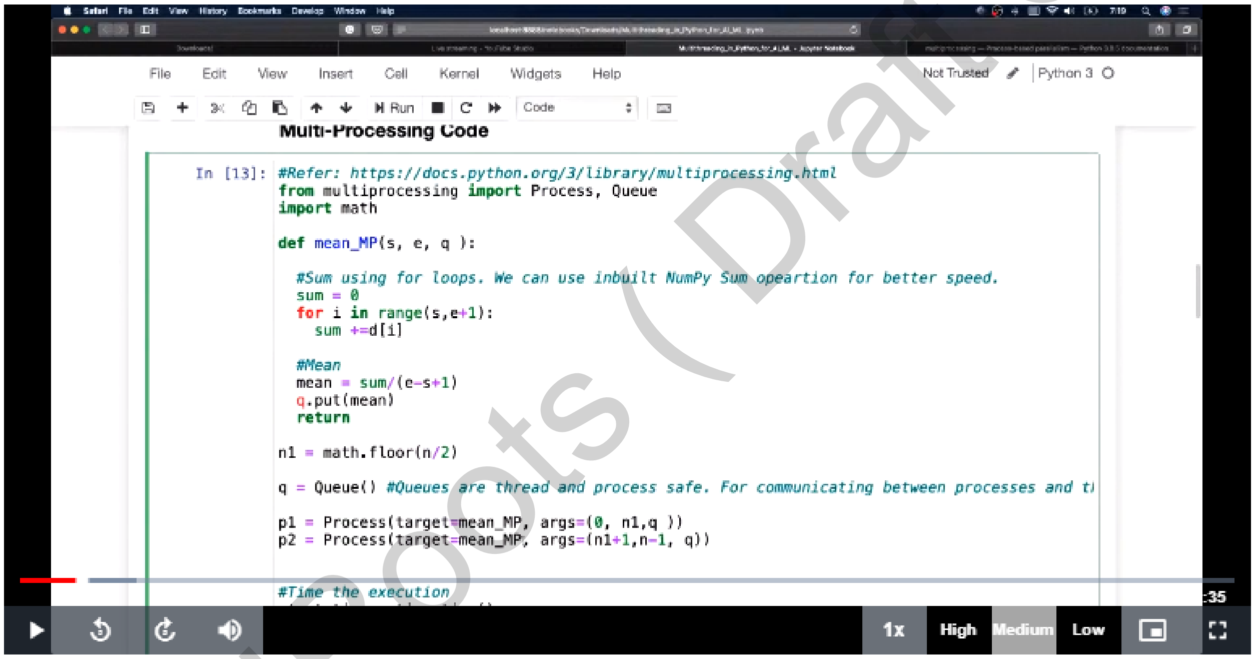
As shown in the above figure a program to calculate mean of 100 million numbers takes 20.98 sec. Note that this program runs on a single core of the processor with this code.



Timestamp: 15:46

We can parallelize the code as shown in the above figure by splitting the 100 million numbers into 50M and 50M and compute the means m1,m2 of both the parts using cpu cores c1 & c2 respectively. Finally, we can combine both the means as (m1+m2)/2. Since each core can run independently, both run parallely to calculate respective means, hence the time taken will be reduced largely.

17.2 Multiprocessing: Compute the mean of 100 Million numbers



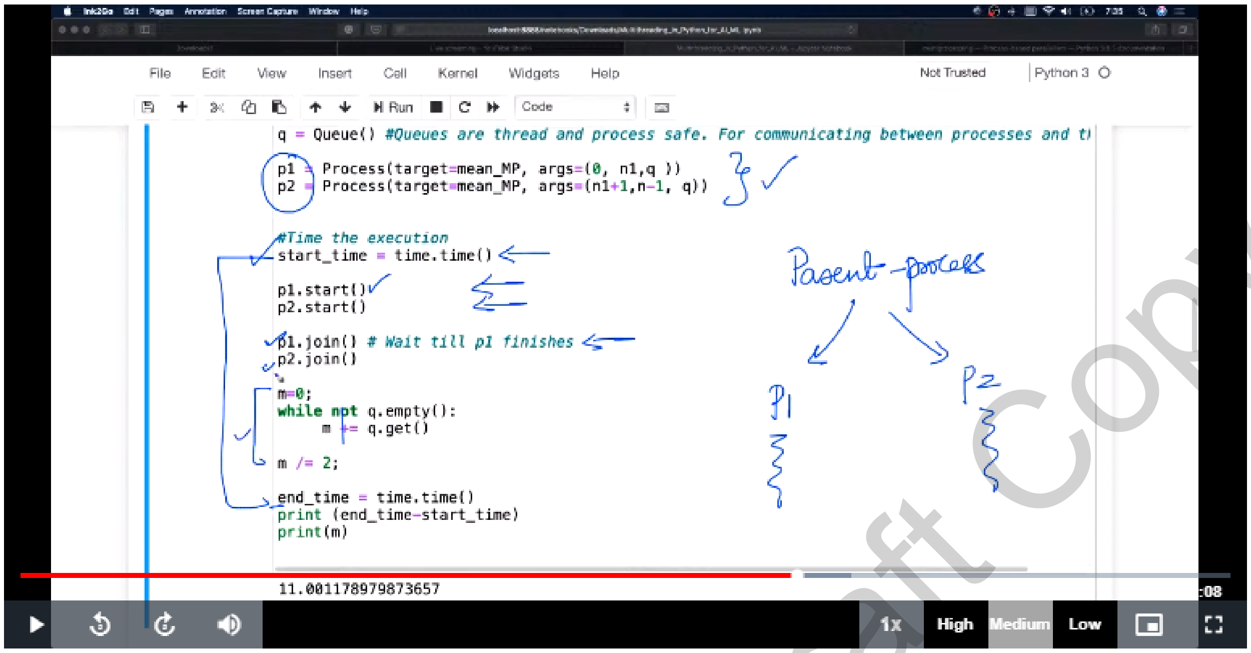
Multi-Processing Code

```
In [13]: #Refer: https://docs.python.org/3/library/multiprocessing.html
         from multiprocessing import Process, Queue
         import math

         def mean_MP(s, e, q ):

           #Sum using for loops. We can use inbuilt NumPy Sum opeartion for better speed.
           sum = 0
           for i in range(s,e+1):
             sum +=d[i]

           #Mean
           mean = sum/(e-s+1)
           q.put(mean)
           return

         n1 = math.floor(n/2)

         q = Queue() #Queues are thread and process safe. For communicating between processes and ti

         p1 = Process(target=mean_MP, args=(0, n1,q ))
         p2 = Process(target=mean_MP, args=(n1+1,n-1, q))


         #Time the execution
```

Timestamp: 1:10

As shown in the above figure, we import Process, Queue from multiprocessing library which will help us create processes and the Queue using which the processes can communicate. We define a function mean_Mp that calculate the mean of the array elements from the start index s to the index e and store the mean corresponding to the part of the array in the queue.

We define the process p1 and p2 to compute the means of the first and second part of the array d respectively.

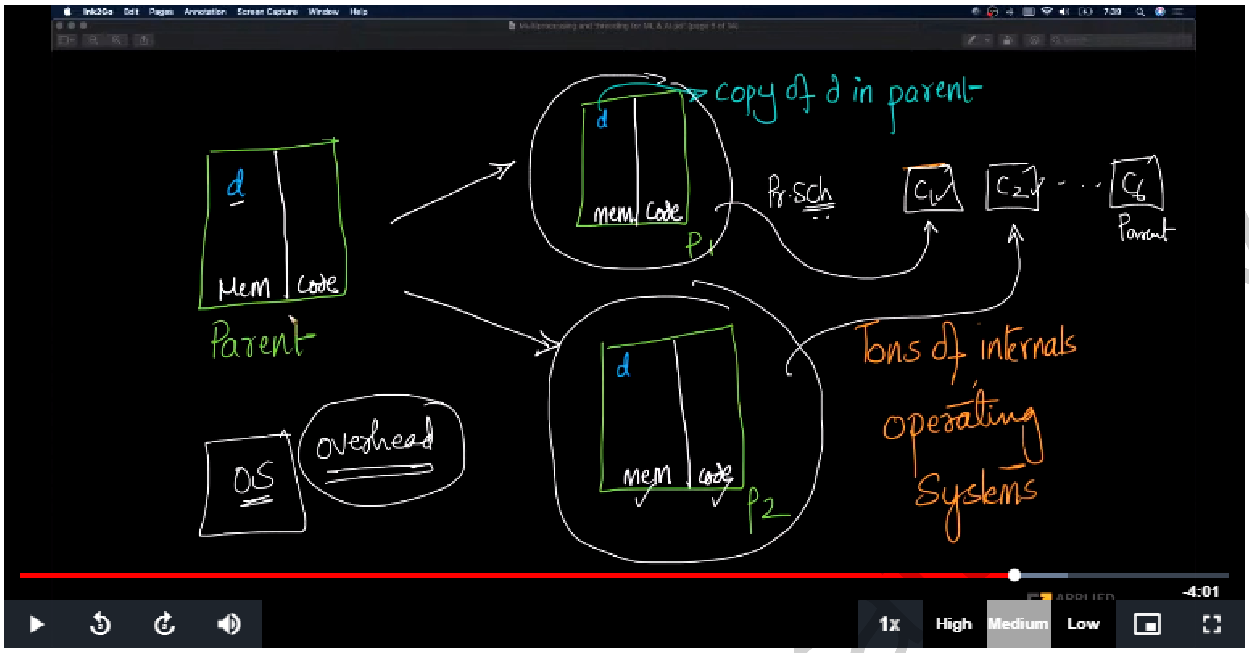


```
q = Queue() #Queues are thread and process safe. For communicating between processes and t

p1 = Process(target=mean_MP, args=(0, n1,q ))
p2 = Process(target=mean_MP, args=(n1+1,n-1, q))

#Time the execution
start_time = time.time()

p1.start()
p2.start()

p1.join() # Wait till p1 finishes
p2.join()

m=0;
while not q.empty():
    m += q.get()

m /= 2;

end_time = time.time()
print (end_time-start_time)
print(m)
```

```
11.001178979873657
```

Timestamp: 14:37

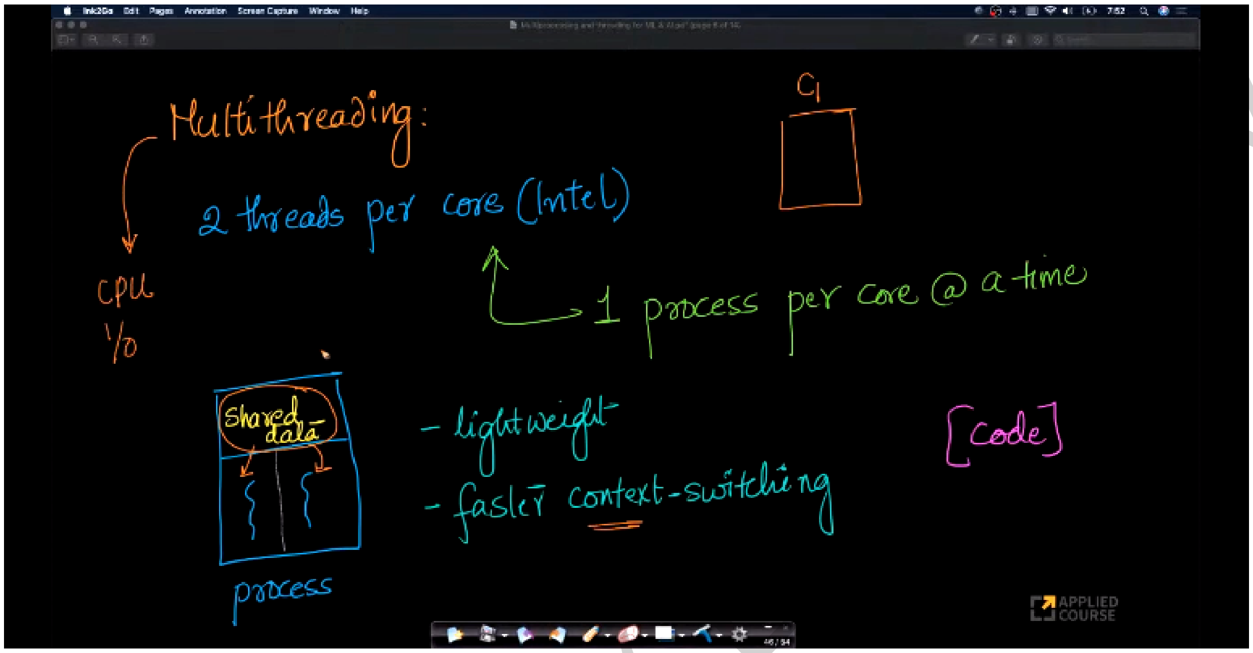
Notice in the above figure, we start the processes using start() and we combine them using join(). join() is used, so that the parent process which actually runs this code waits for the two child processes responsible for calculating the means. So once the child processes complete calculating means and storing in the queue, we then calculate the mean of the entire array in the parent process using get method of queue. Notice that the time taken for calculating the mean of the array using two process is 11.00 sec. This is a significant improvement over time taken by 1 process.



Timestamp: 18:44

Notice that while creating these child processes, the parent process copies d to both processes P1 and P2. Hence both these processes has access to d for calculating the means. If we notice, with 2 processes it took 11.00 sec instead of 20.98/2=10.47 sec (due to two cores). This additional time came from the OS overhead of creating two processes with the required code, memory and allocating them to individual cores. Note that during the entire time when both the child processes is computing the means, the parent process sleeps since it cannot proceed further unless the child processes finishes executing their code.

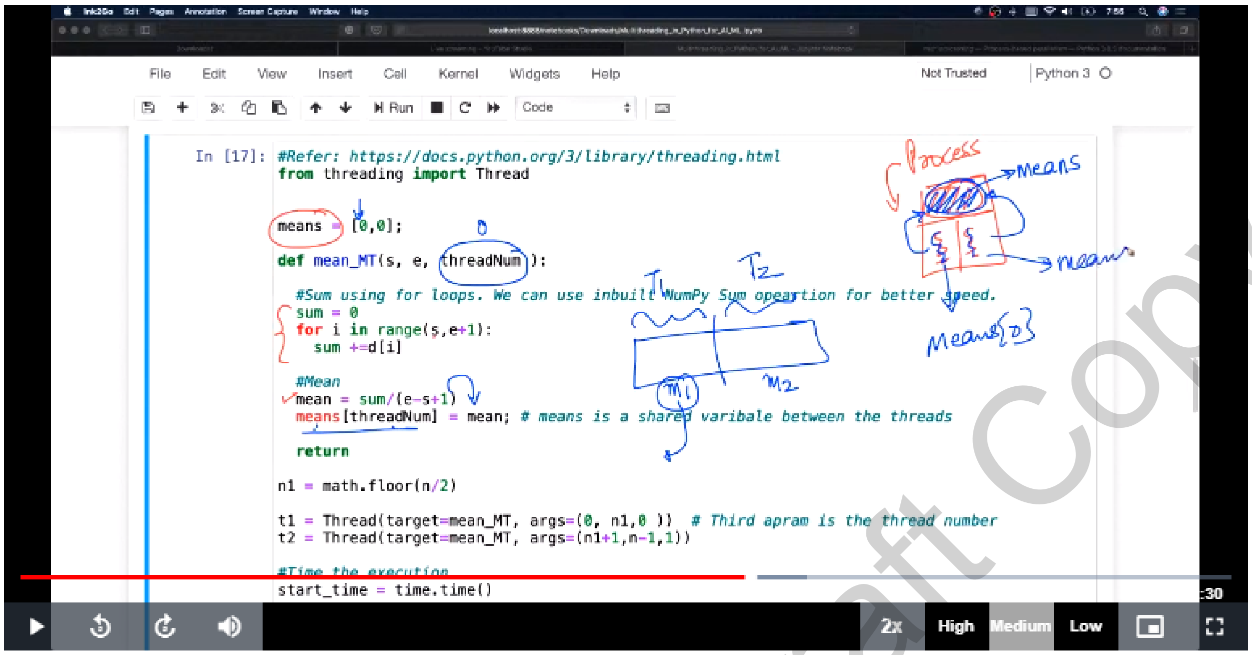
17.3 Multi-threading with code


Multithreading:
2 threads per core (Intel)
1 process per core @ a time
CPU
I/O
C1
Shared data
process
- lightweight
- faster context-switching
[code]
APPLIED COURSE

Timestamp: 9:00

Using multi-threading, we can run upto 2 threads per core. Threads are known as light weight processes with faster context switching. Context switching is the time to create two process with all the required code and memory. Since threads are executed on the same core and use same shared data they have faster context-switching times.

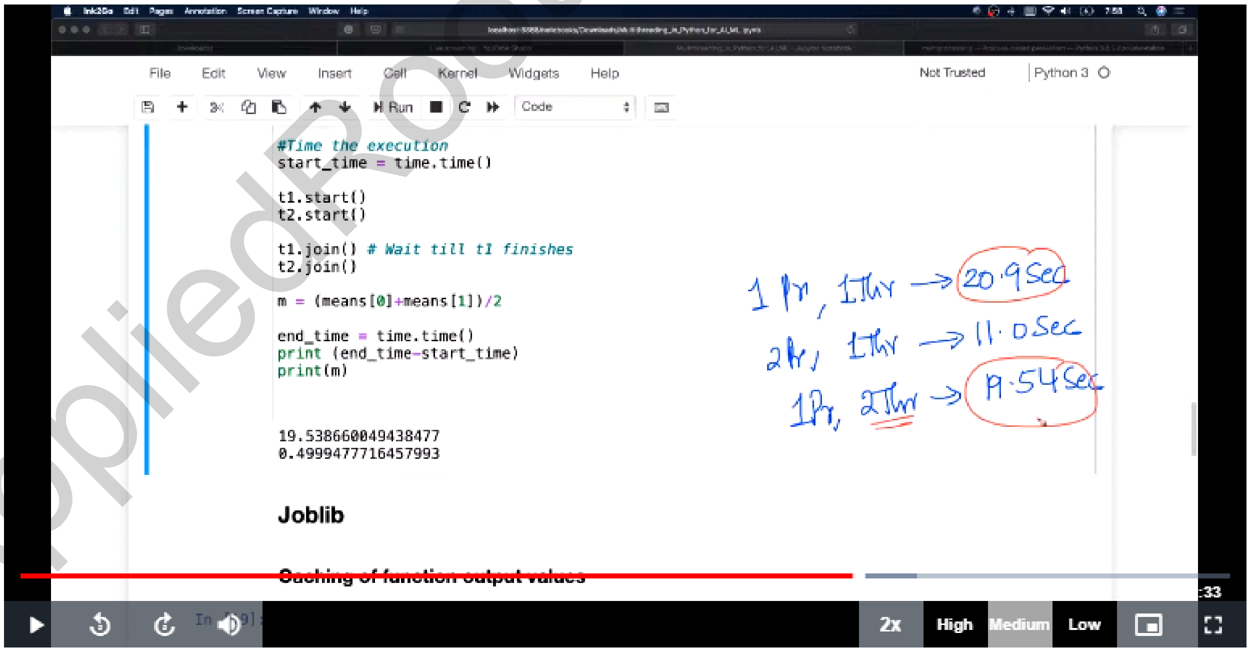
Multithreading is helpful when we have two threads both executing CPU bound and I/O bound operations, so that when one thread is performing the I/O operations, the other thread can run its code on the cpu.



```
In [17]: #Refer: https://docs.python.org/3/library/threading.html
         from threading import Thread

         means = [0,0];

         def mean_MT(s, e, threadNum):

             #Sum using for loops. We can use inbuilt NumPy Sum opeartion for better speed.
             sum = 0
             for i in range(s,e+1):
               sum +=d[i]

             #Mean
             mean = sum/(e-s+1)
             means[threadNum] = mean; # means is a shared varibale between the threads

             return

         n1 = math.floor(n/2)

         t1 = Thread(target=mean_MT, args=(0, n1,0 ))  # Third apram is the thread number
         t2 = Thread(target=mean_MT, args=(n1+1,n-1,1))

         #Time the execution
         start_time = time.time()
```

Timestamp: 12:56

Multithreading can be done in python using the threading library. Similar to how we have created processes we have created threads but in addition we give the thread number 0 and 1 to the function mean_MT and store the means in 0 and 1 st index of the means since both the threads share common memory.



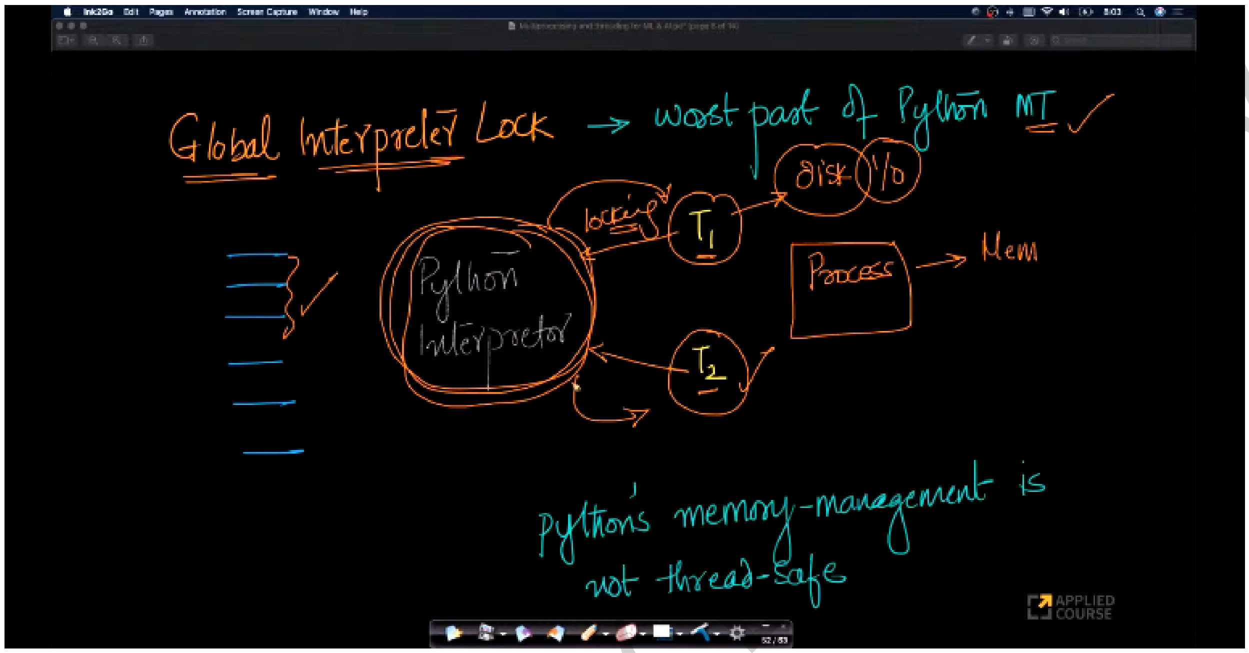
```
#Time the execution
start_time = time.time()

t1.start()
t2.start()

t1.join() # Wait till t1 finishes
t2.join()

m = (means[0]+means[1])/2

end_time = time.time()
print (end_time-start_time)
print(m)

19.538660049438477
0.4999477716457993
```

Joblib

Caching of function output values

Timestamp: 14:53

Notice that the speed from 1 thread to 2 threads for the given problem is very less. This is since both are CPU bound process there is not much parallelism going on. It is also due to a problem called GIL to be explained below.

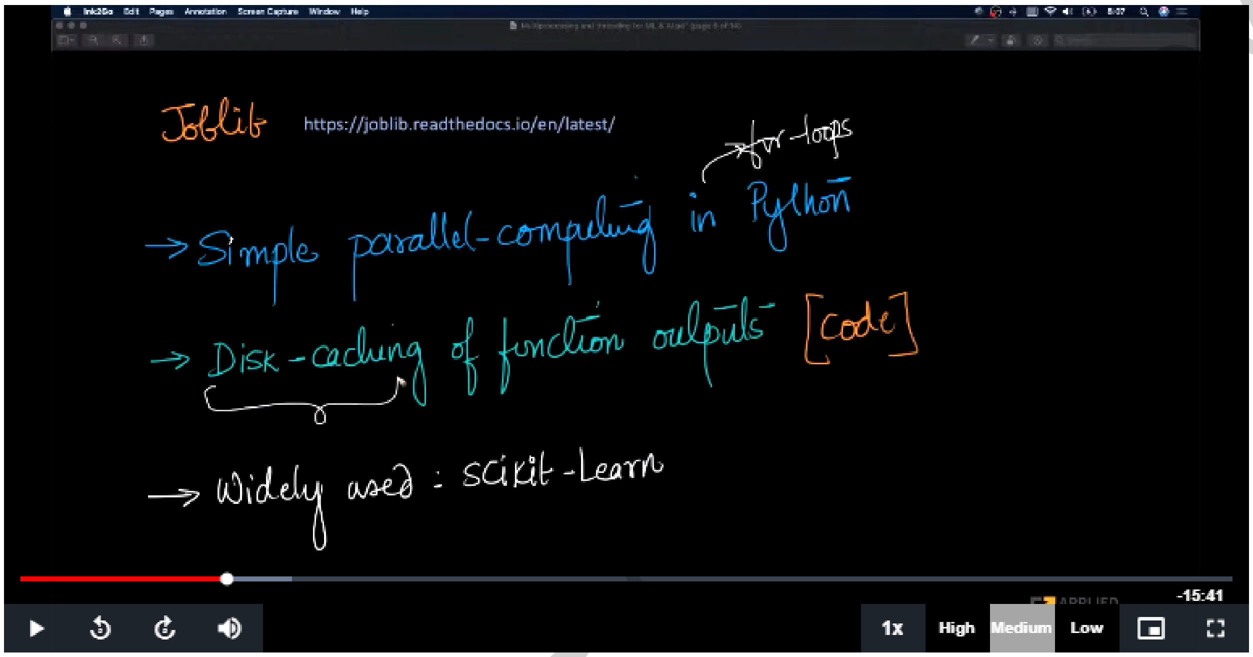

Global Interpreter Lock → worst part of Python MT ✓
disk I/O
locking
T1
T2
Python Interpretor
Process → Mem
python's memory-management is not thread-safe
APPLIED COURSE

Timestamp: 20:14

While using multithreading in python, both the threads need access to the python interpreter to execute their code. But in python only one thread can access the python interpreter and the other thread is left waiting for the other thread to release the python interpreter. This is called global interpreter lock.This releasing happens when the thread is waiting for some I/O or disk operations.

Note that python's memory management is not thread safe meaning if different threads write to a single memory then it can get messed up.

17.4 Joblib: Simplified Parallel Programming in python


Joblib https://joblib.readthedocs.io/en/latest/

for-loops

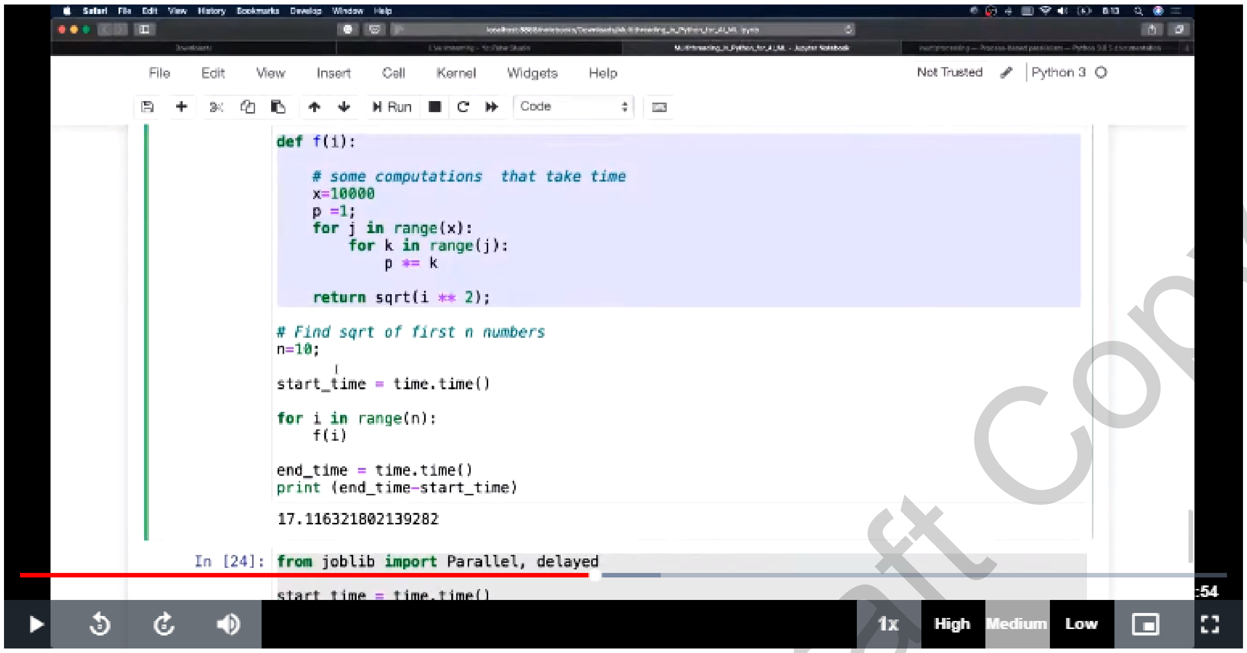
→ Simple parallel-computing in Python

→ Disk-caching of function outputs [code]

→ Widely used : scikit-Learn

Timestamp: 3:14

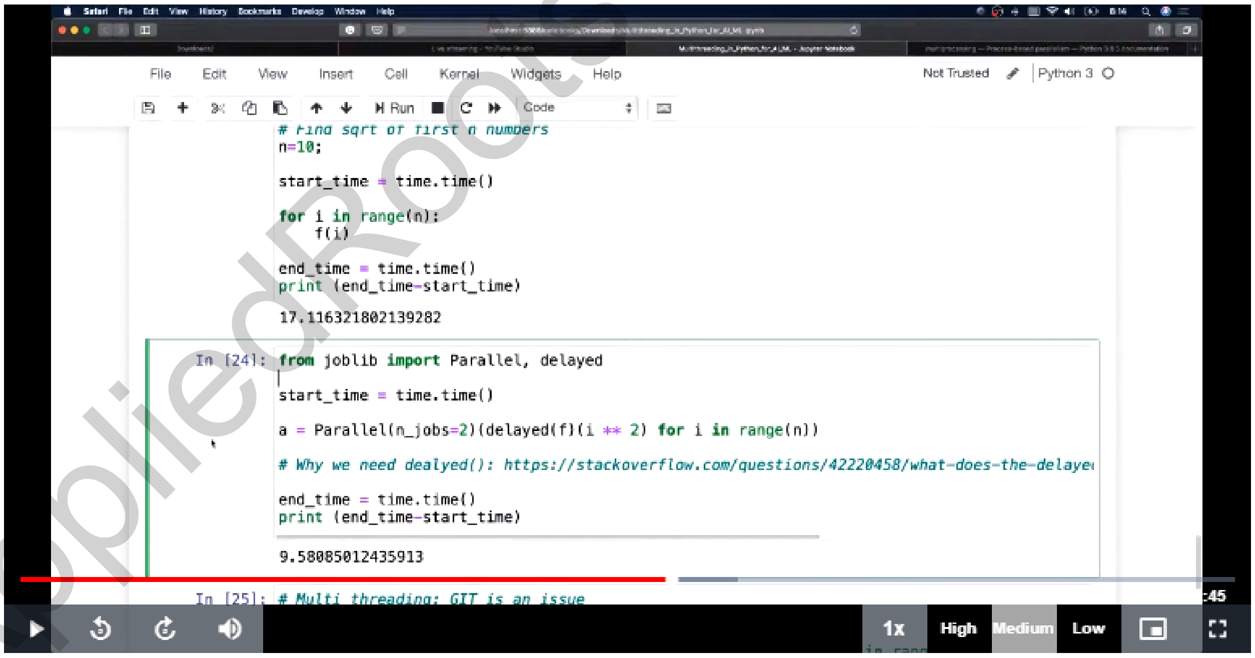
Instead of using the raw multithreading and multiprocessing libraries we can work with joblib which is a simple parallel computing library in python. It can highly optimize for loops and it can use disk caching of function outputs.
So when we pass an input a to a function f and get output o, it store that information that a function f when given input a gives output o. So the next time when someone calls the same function with the same input a, it searches whether it has seen the same input for the same function, if found in cache it returns the corresponding output o without computing it again.
It is widely used in scikit learn.



```
def f(i):

    # some computations  that take time
    x=10000
    p =1;
    for j in range(x):
        for k in range(j):
            p *= k

    return sqrt(i ** 2);

# Find sqrt of first n numbers
n=10;

start_time = time.time()

for i in range(n):
    f(i)

end_time = time.time()
print (end_time-start_time)

17.116321802139282

In [24]: from joblib import Parallel, delayed
```

Timestamp: 9:01

Notice in the above code with no parallelization, it took 17.11 seconds to run the function f(i) for i in range(n).

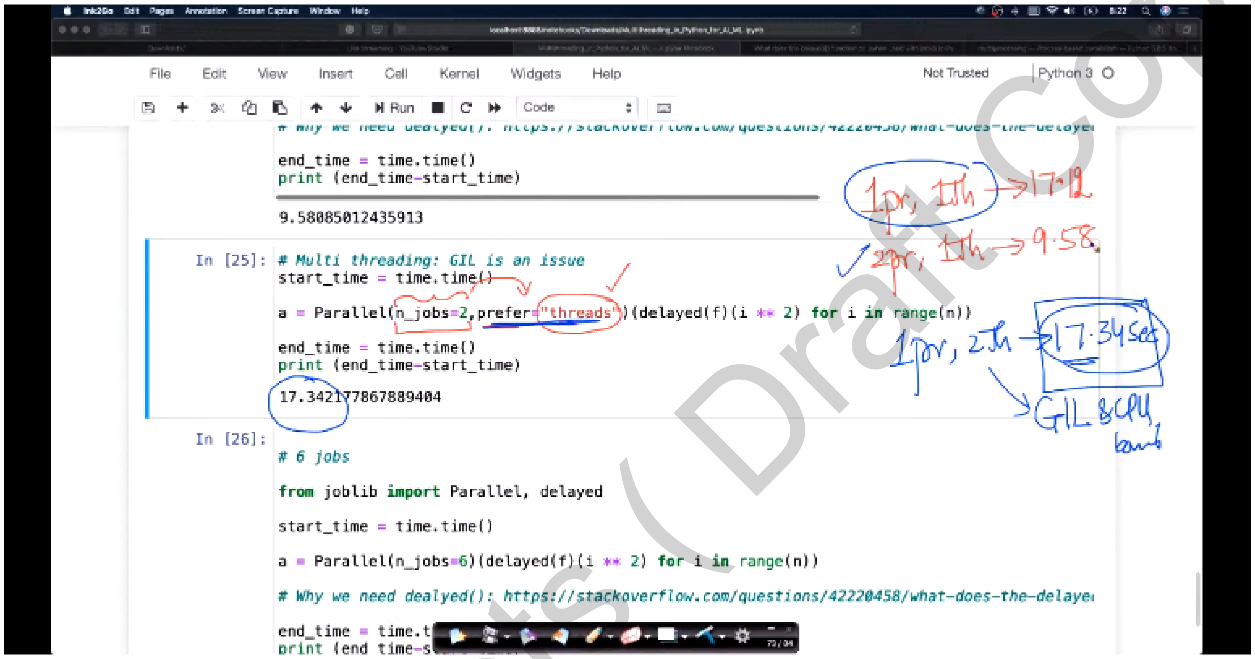


```
# Find sqrt of first n numbers
n=10;

start_time = time.time()

for i in range(n):
    f(i)

end_time = time.time()
print (end_time-start_time)

17.116321802139282

In [24]: from joblib import Parallel, delayed

start_time = time.time()

a = Parallel(n_jobs=2)(delayed(f)(i ** 2) for i in range(n))

# Why we need dealyed(): https://stackoverflow.com/questions/42220458/what-does-the-delaye

end_time = time.time()
print (end_time-start_time)

9.58085012435913

In [25]: # Multi threading: GIT is an issue
```

Timestamp: 10:10

Notice that using joblib with 2 cores specified by n_jobs it took only 9.58 sec. Please refer the below link to learn about delayed. But what it does is to say the function we want to run and the inputs to run it with and let all the inputs be ready before we parallelize.

https://stackoverflow.com/questions/42220458/what-does-the-delayed-function-do-when-used-with-joblib-in-python#:~:text=Delayed%20is%20a%20decorator%20that,and%20popped%20out%20as%20needed.



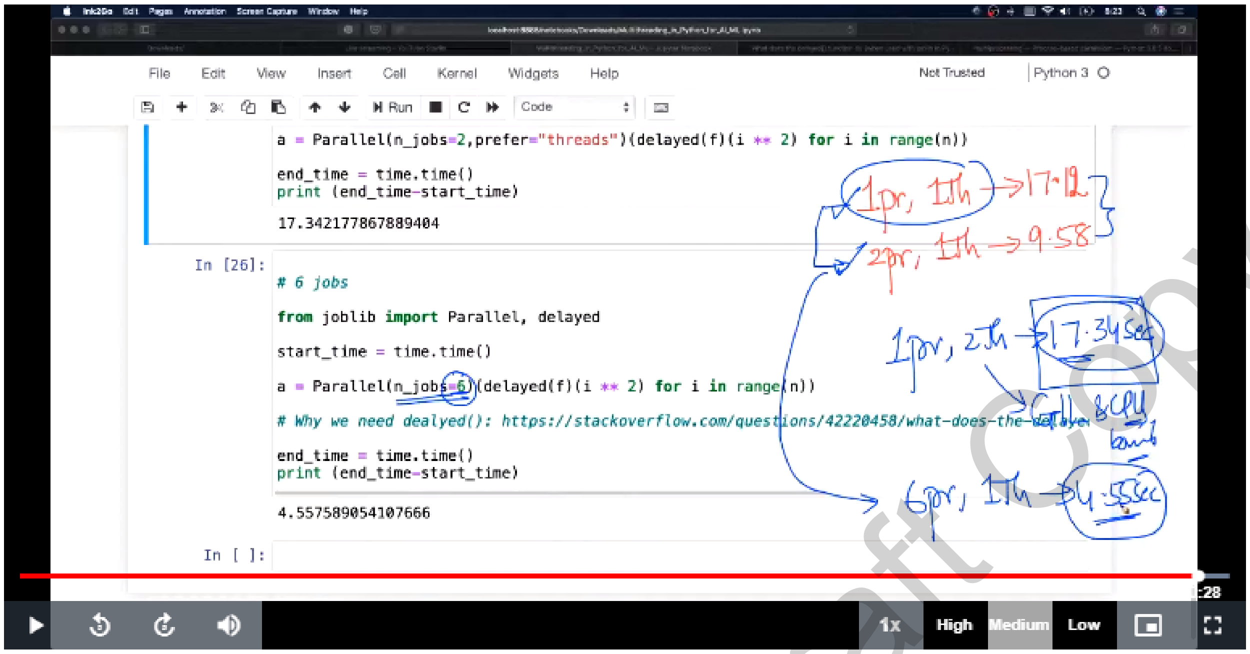
```
end_time = time.time()
print (end_time-start_time)

9.58085012435913
```

```
In [25]: # Multi threading: GIL is an issue
         start_time = time.time()

         a = Parallel(n_jobs=2,prefer="threads")(delayed(f)(i ** 2) for i in range(n))

         end_time = time.time()
         print (end_time-start_time)

         17.342177867889404
```

```
In [26]:
         # 6 jobs

         from joblib import Parallel, delayed

         start_time = time.time()

         a = Parallel(n_jobs=6)(delayed(f)(i ** 2) for i in range(n))

         # Why we need dealyed(): https://stackoverflow.com/questions/42220458/what-does-the-delaye

         end_time = time.t
         print (end_time-s
```

Timestamp: 17:41

We can also do multithreading using joblib as shown in the above figure. Note that the time taken for 2 threads is longer than the time taken with 1 thread. This is due to GIL and since both the threads are cpu bound.



```
a = Parallel(n_jobs=2,prefer="threads")(delayed(f)(i ** 2) for i in range(n))

end_time = time.time()
print (end_time-start_time)

17.342177867889404
```

```
In [26]:
# 6 jobs

from joblib import Parallel, delayed

start_time = time.time()

a = Parallel(n_jobs=6)(delayed(f)(i ** 2) for i in range(n))

# Why we need dealyed(): https://stackoverflow.com/questions/42220458/what-does-the-delaye

end_time = time.time()
print (end_time-start_time)

4.557589054107666

In [ ]:
```

Timestamp: 18:35

Notice that with 6 processes it took 4.55 which is longer than 17.12/6=2.85. This is because of the overhead of creating these 6 process and tracking them and content switch.