# Workshop on AI/ML using Python

## Topics Covered:

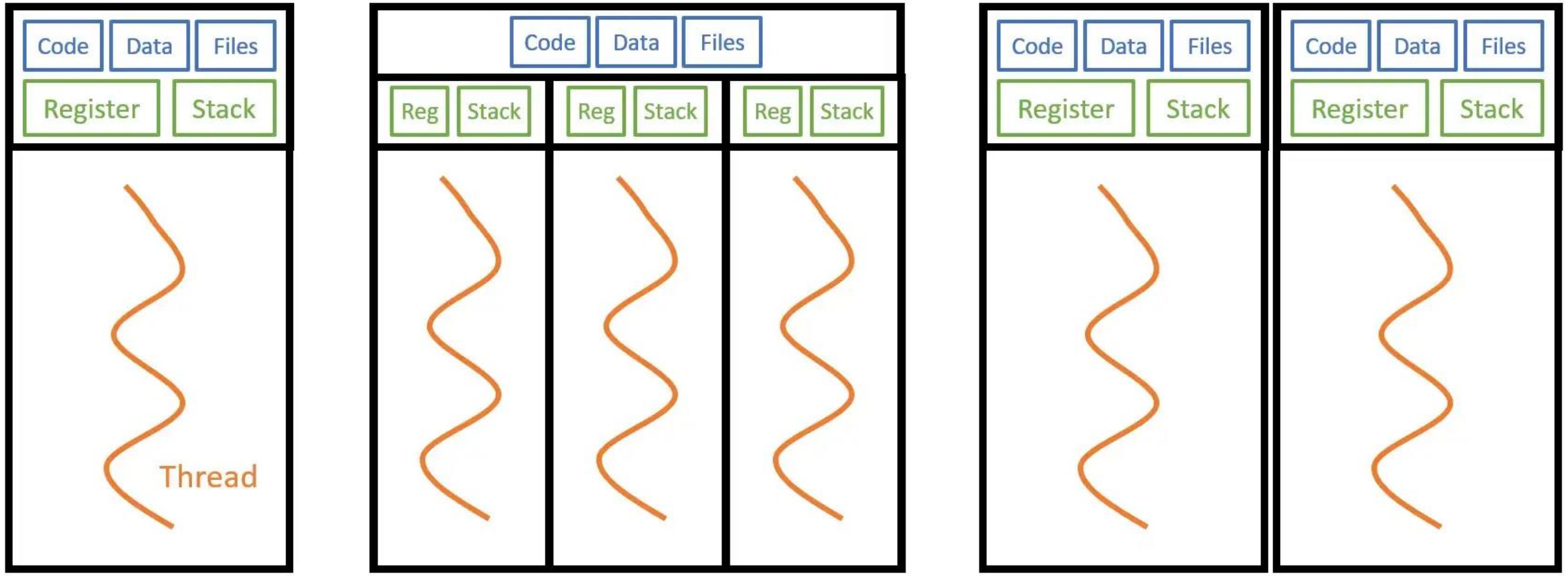
1. Concurrency in Python-Multithreading and Multiprocessing
2. Collections
3. Memory Mapped Files
4. Working with binary data

Hemant Lalwani
HRDPD/SIPG

# Concurrency in Python Multithreading and Multiprocessing

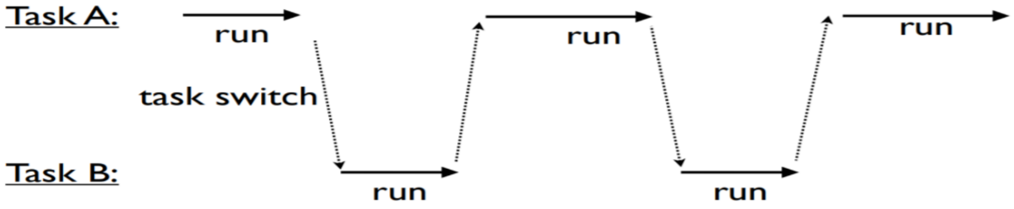
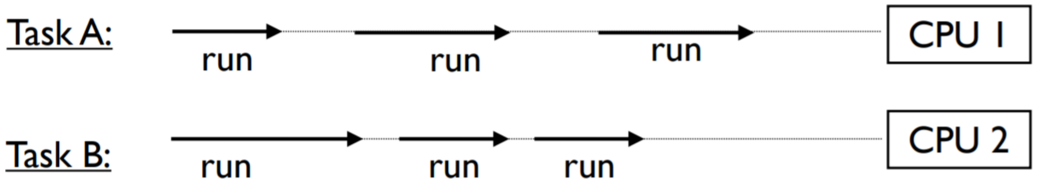
# Introduction – Basic Terminology

- **Program** : Is an executable file consisting of a set of instructions to perform some task.
- **Process** : Is a program in execution (Loaded into main memory along with all the resources it needs to operate).
- **Thread** : Is a unit of execution within a process or a lightweight process . (A process can have anywhere from one thread to many).




Single Processor Single Thread | Single Processor Multithread | Multiprocessing

# Concurrency vs Parallelism

| Concurrent execution or concurrency | Parallel execution or parallelism |
|---|---|
| • A condition when two or more tasks can start, execute and complete (making progress) in overlapping time periods.<br>• If only one CPU is available, the only way it can run multiple tasks is by rapidly switching between them. | • A condition when two or more tasks are executing simultaneously ( run at the same time on multi-core processor). |
|   |   |

# Multitasking

- Ability of operating system to perform multiple tasks (processes, programs, threads)at the same time.
- utilize the CPU to reduce response time and improves performance.

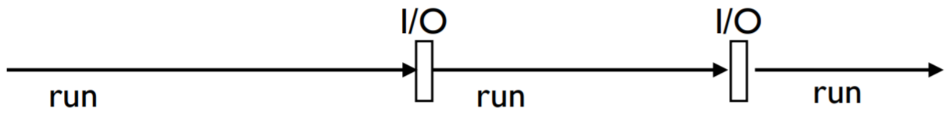
| Process-Based Multitasking(Multiprocessing) | Thread-Based Multitasking(Multithreading) |
|---|---|
| • Executing multiple tasks simultaneously, where each task is separate independent process is called as process based multitasking.<br>• **Example** | • Executing multiple tasks simultaneously, where each task is separate independent part of process (or) program.<br>• Multi-threading allows single process to have multiple code segments (threads) running concurrently within the context of process.<br>• **Example** |

# Task Execution – CPU Bound and I/O Bound Tasks

- All tasks execute by alternating between CPU processing and I/O handling
- For I/O, tasks must wait (sleep)
- Behind the scenes, the underlying system will carry out the I/O operation and wake the task when it's finished.

## CPU Bound Tasks

- A task is "CPU Bound" if it spends most of its time doing computation with little I/O

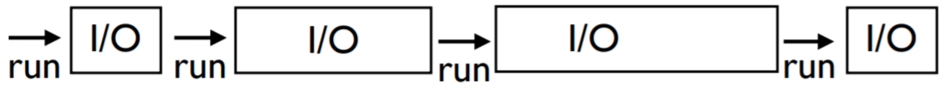



- **Examples:**

➢ Matrix multiplication
➢ Video Compression
➢ Image Processing

## I/O Bound Tasks

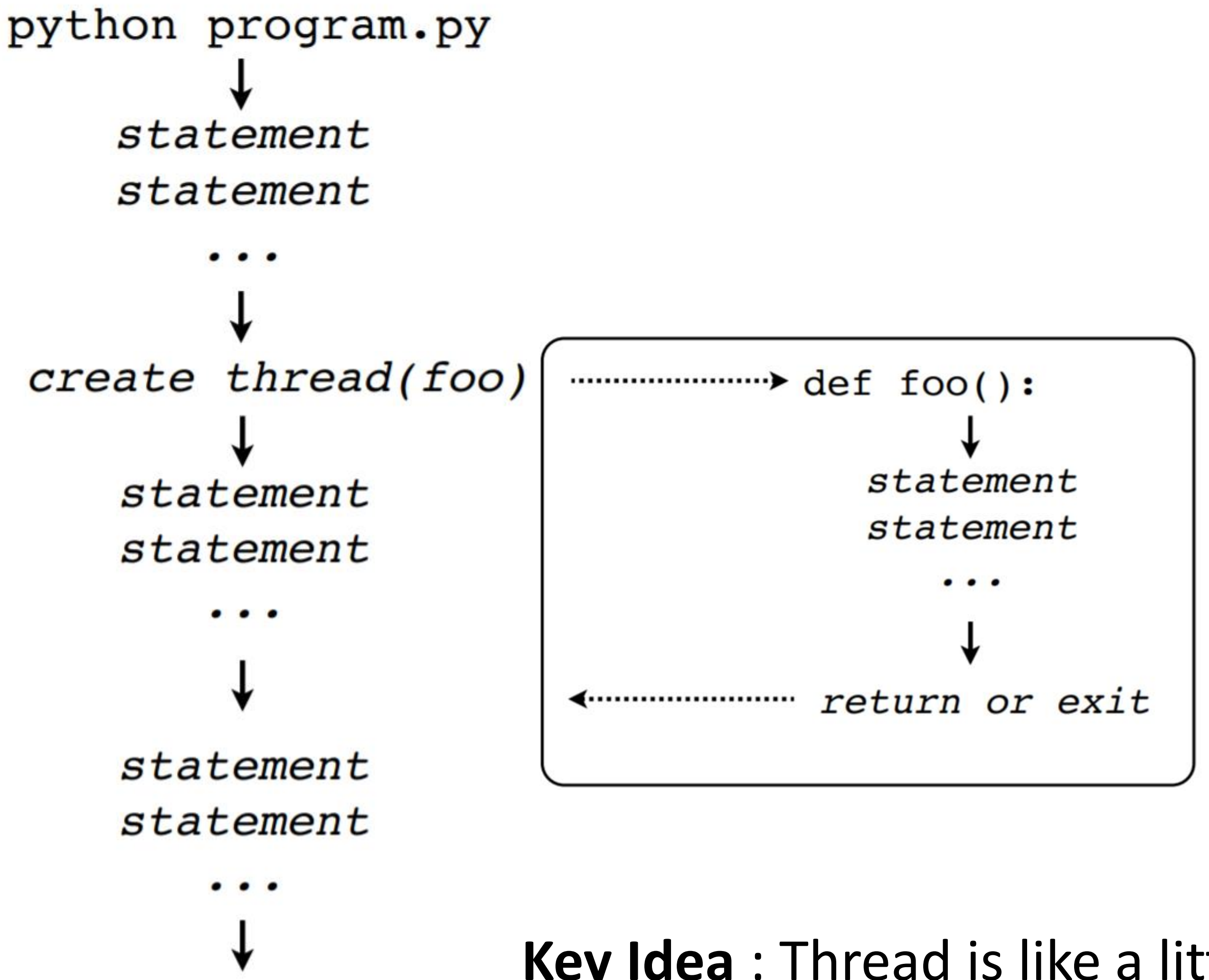
- A task is “I/O Bound" if it spends most of its time doing I/O than doing computation.




- **Examples:**

➢ Reading input from user
➢ File processing
➢ Database connections, N/W Requests

# Thread Basics




## Steps

- Program Launch. Python loads a program and starts executing statements.
- Creation of a thread. Launching of function
- Concurrent execution of statements
- Thread terminates on return or exit

**Key Idea** : Thread is like a little “task” that independently runs inside program.

# How to create threads in Python ?

**Threading Module :**

This Module provides Thread class, and this Thread class provide following methods

- start() − starts a thread by calling the run method.
- join()− waits for threads to terminate.
- isAlive() − checks whether a thread is still executing.
- getName() − returns the name of a thread.
- setName() − sets the name of a thread.

Creating Thread Using Threading Module

*threading.Thread (target=None, name=None, args=())*

- target is the callable function to be invoked by the run() method.
- name is the thread name.
- args is the argument tuple for the function invocation.

## By Extending Thread Class

```python
import threading
#inherit Thread class and override init()
# & run() function:
class mythread(threading.Thread):
    def __init__(self,msg):
        super(mythread,self).__init__()
        self.msg=msg
    def run(self):
        for i in range(5):
            print(self.msg)
#Create an object of Thread class to create new thread.
t1 = mythread("Thread1")
t2 = mythread("Thread2")
#Call start method of Thread class to start thread
t1.start()
t2.start()
#Once the threads start, the current program also
#keeps on executing. In order to stop execution of
#current program until a thread is complete,call join.
t1.join()
t2.join()
# The current program will first wait for the completion
#of t1 and then t2. Once, they are finished, the remaining
#statements of current program are executed.
print("Done!")
```

## Without Extending Thread Class

```python
import threading
def display(msg):
    for i in range(5):
        print(msg)
#Create an object of Thread class to create new thread.
t1 = threading.Thread(target=display, args=("Thread1",))
t2 = threading.Thread(target=display, args=("Thread2",))
# To start a thread, we use start method of Thread class.
t1.start()
t2.start()
#Once the threads start, the current program also
#keeps on executing. In order to stop execution of
#current program until a thread is complete,call join.
t1.join()
t2.join()
# The current program will first wait for the completion
#of t1 and then t2. Once, they are finished, the remaining
#statements of current program are executed.
print("Done!")
```

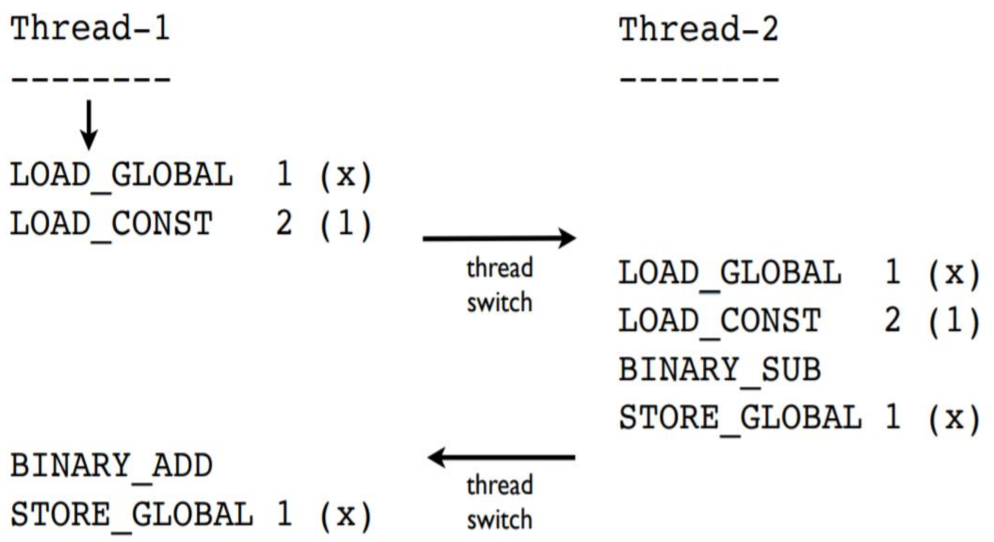
# Shared Data Access

- Thread share all of the data in program.
- Thread scheduling is non-deterministic. Access to any kind of shared data is also non-deterministic.

```python
import threading
x = 0
def foo():
    global x
    for i in range(100000): x += 1

def bar():
    global x
    for i in range(100000): x -= 1

t1 = threading.Thread(target=foo)
t2 = threading.Thread(target=bar)
t1.start(); t2.start()
t1.join(); t2.join()
print(x)    # Expected result is 0
```




- Whenever two or more processes/threads manipulate a shared resource concurrently and the outcome depends on particular order in which access takes place ,a "**Race Condition**" occurs.
- **Thread Synchronization** (Only one thread/process can make modification to shared data at any given time) - Using Lock , Semaphore

# Whether Multithreading is useful or not ?

## A Performance Test

```
import datetime
from threading import *
def countdown(n):
    while n > 0 :
        n = n - 1
count = 10000000
# Sequential Execution
start = datetime.datetime.now()
countdown(count)
end = datetime.datetime.now()
print(end-start)
# Threaded Execution
start = datetime.datetime.now()
t1 = Thread(target= countdown,args=(count/2,))
t2 = Thread(target= countdown,args=(count/2,))
t1.start(); t2.start();
t1.join();t2.join();
end = datetime.datetime.now()
print(end-start)
```

```
0:00:00.652738
0:00:01.104947
```

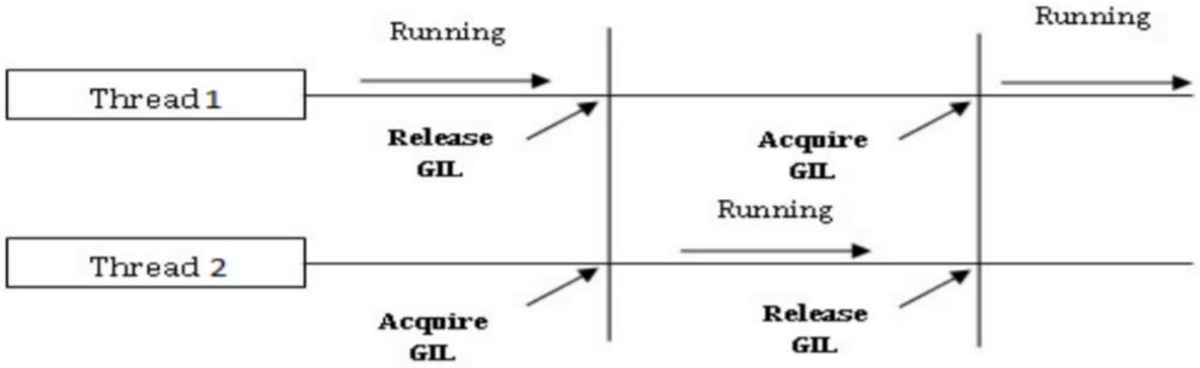
- Now, you might expect two threads to run twice as fast on multiple CPU cores.

- Sequential : 0.65 s
- Threaded(2 threads) : 1.10 s
- Threaded(4 threads) : 1.43 s

Why this happened ?

# Global Interpreter Lock(GIL)

- Python does not have a thread scheduler. Fully managed by the host operating system(All scheduling/thread switching)
- Execution of Python code is controlled by the Python Virtual Machine.
- Multiple threads may be "running" within the Python interpreter, only one thread is being executed by the interpreter at any given time.
- Access to the Python Virtual Machine is controlled by the global interpreter lock (GIL). This lock ensures that exactly one thread is running in the interpreter at once.
- Simplifies many low-level details (memory management, callouts to C extensions, etc.)
- When a thread starts running, it acquires GIL and when it waits for I/O, it releases the GIL, so that other threads of that process can run.



# Multiprocessing

- An alternative to threads is to run multiple independent copies of the Python interpreter in separate processes. Each instance of Python is independent.
- With multiprocessing, there are no shared memory address space. Every process is completely isolated/independent.
- Different interpreters cooperate by sending messages to each other using Pipes/Sockets/Queues.

## How to create processes in Python ?

**Multiprocessing Module :**

- A module for writing concurrent Python programs based on communicating processes.
- useful for concurrent CPU-bound processing.

Creating Process Using mulitprocessing Module

*mulitprocessing.process(target=None, name=None, args=())*

Note : Always launch process in main()

## By Extending Process Class

```python
import multiprocessing
# inherit Process class and override
#init() & run() function:
class countDown(multiprocessing.Process):
    def __init__(self,count):
        super(countDown,self).__init__()
        self.count=count
    def run(self):
        while self.count > 0:
            print(self.count)
            self.count -=1
#To create a new process, we create an
#object of Process class.
if __name__=='__main__':
    p1 = countDown (10)
    p2 = countDown (5)
    p1.start() ;p2.start()
    p1.join();p2.join()
print("Done!")
```

## Without Extending Process Class

```python
from multiprocessing import *
def countDown (count):
    while count > 0:
        print(count)
        count -=1
#To create a new process, we create an
#object of Process class.
if __name__=='__main__':
    p1 = Process(target=countDown, args=(10,))
    p2 = Process(target=countDown, args=(5,))
    p1.start() ;p2.start()
    p1.join();p2.join()
    print("Done!")
```

# Whether Multiprocessing is useful or not ?

## A Performance Test

```python
import datetime
from multiprocessing import *
def countdown(n):
    while n > 0 :
        n = n - 1
count = 10000000
# Sequential Execution
start = datetime.datetime.now()
countdown(count)
end = datetime.datetime.now()
print(end-start)
# Threaded Execution
start = datetime.datetime.now()
t1 = Process(target= countdown,args=(count/2,))
t2 = Process(target= countdown,args=(count/2,))
t1.start(); t2.start();
t1.join();t2.join();
end = datetime.datetime.now()
print(end-start)
```

- Sequential : 0.65 s
- Parallel (2 processes) : 0.44 s
- Parallel (4 processes) : 0.30 s

# Process Communication
# Pipes

- The Pipe() function returns a pair of connection objects connected by a pipe.
- Each connection object has send() and recv() methods to send and receive messages

```python
from multiprocessing import *
def process1_send_function(conn, events):
    for event in events:
        conn.send(event)
        print("Sent:",event)
def process2_recv_function(conn):
    while True:
        event = conn.recv()
        if event == "eod":
            print("Event Received: End of Day")
            return
        print(f"Event Received: {event}")
if __name__ == "__main__":
    events = ["get up", "brush your teeth", "shower", "work", "eod"]
    conn1, conn2 = Pipe()
    process_1 = Process(target=process1_send_function, args=(conn1, events))
    process_2 = Process(target=process2_recv_function, args=(conn2,))
    process_1.start(); process_2.start()
    process_1.join(); process_2.join()
```

# Process Communication
# Queues

A simple way to communicate between process with multiprocessing is to use a Queue to pass messages back and forth.

```python
import multiprocessing
def process1_send_function(queue, events):
    for event in events:
        queue.put(event)
        print(f"Event Sent: {event}")
def process2_recv_function(queue):
    while True:
        event = queue.get()
        if event == "eod":
            print("Event Received: End of Day")
            return
        print(f"Event Received: {event}")
if __name__ == "__main__":
    events = ["get up", "brush your teeth", "shower", "work", "eod"]
    queue = multiprocessing.Queue()
    process_1 = multiprocessing.Process(target=process1_send_function, args=(queue, events))
    process_2 = multiprocessing.Process(target=process2_recv_function, args=(queue,))
    process_1.start(); process_2.start()
    process_1.join(); process_2.join()
```

# Process Synchronization
# Locks

Synchronization ensures that two or more concurrent processes/threads do not simultaneously execute some particular program segment where the shared resources are accessed known as critical section.

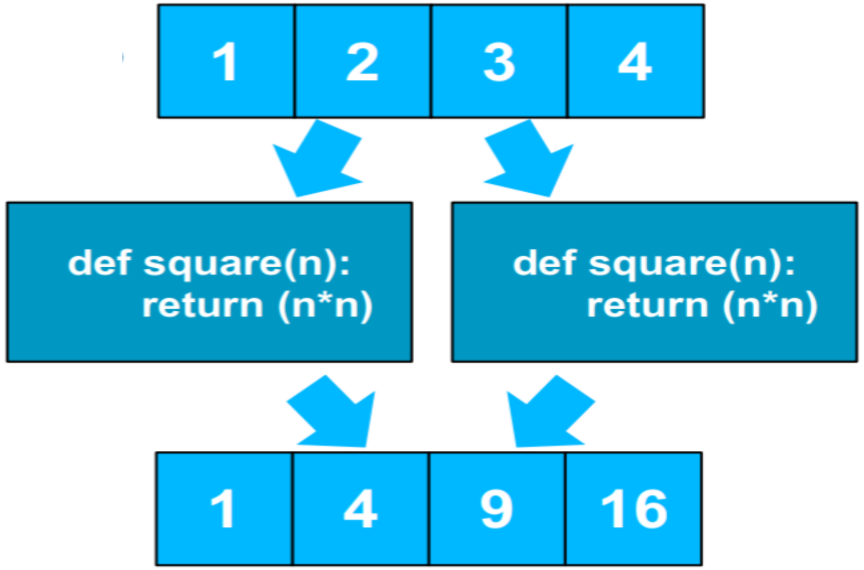
```python
from multiprocessing import *
# function to withdraw from account
def withdraw(balance):
    for _ in range(10000):
        balance.value = balance.value - 1
# function to deposit to account
def deposit(balance):
    for _ in range(10000):
        balance.value = balance.value + 1
if  __name__=='__main__':
    # initial balance (in shared memory)
    balance = multiprocessing.Value('i', 100)
    # creating new processes
    p1 = Process(target=withdraw, args=(balance,))
    p2 = Process(target=deposit, args=(balance,))
    p1.start(); p2.start();
    p1.join(); p2.join();
    print(balance.value)
```

```python
from multiprocessing import *
# function to withdraw from account
def withdraw(balance,lock):
    for _ in range(10000):
        lock.acquire()
        balance.value = balance.value - 1
        lock.release()
# function to deposit to account
def deposit(balance,lock):
    for _ in range(10000):
        lock.acquire()
        balance.value = balance.value + 1
        lock.release()
if  __name__=='__main__':
    # initial balance (in shared memory)
    balance = multiprocessing.Value('i', 100)
    # creating a lock object
    lock = multiprocessing.Lock()
    # creating new processes
    p1 = Process(target=withdraw, args=(balance,lock))
    p2 = Process(target=deposit, args=(balance,lock))
    p1.start(); p2.start();
    p1.join(); p2.join();
    print(balance.value)
```

# Work Distribution
# Pool

- multiprocessing module provides a Pool class that represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes.

```python
import multiprocessing
def square(n):
    return n*n
if __name__ == "__main__":
    mylist = [1,2,3,4]
    # creating a pool object
    p = multiprocessing.Pool()
    # map list to target function
    result = p.map(square, mylist)
```



# Pool Methods

When choosing one, you have to take multi-args, blocking, and ordering into account:

| | Multi-args | Blocking | Ordered-results |
|---|---|---|---|
| pool.map | no | yes | yes |
| pool.map_async | no | no | yes |
| pool.apply | yes | yes | yes |
| Pool.apply_async | yes | no | yes |

TASK: Image Augmentation with multiprocessing.

Steps : 1. Read an Image ,

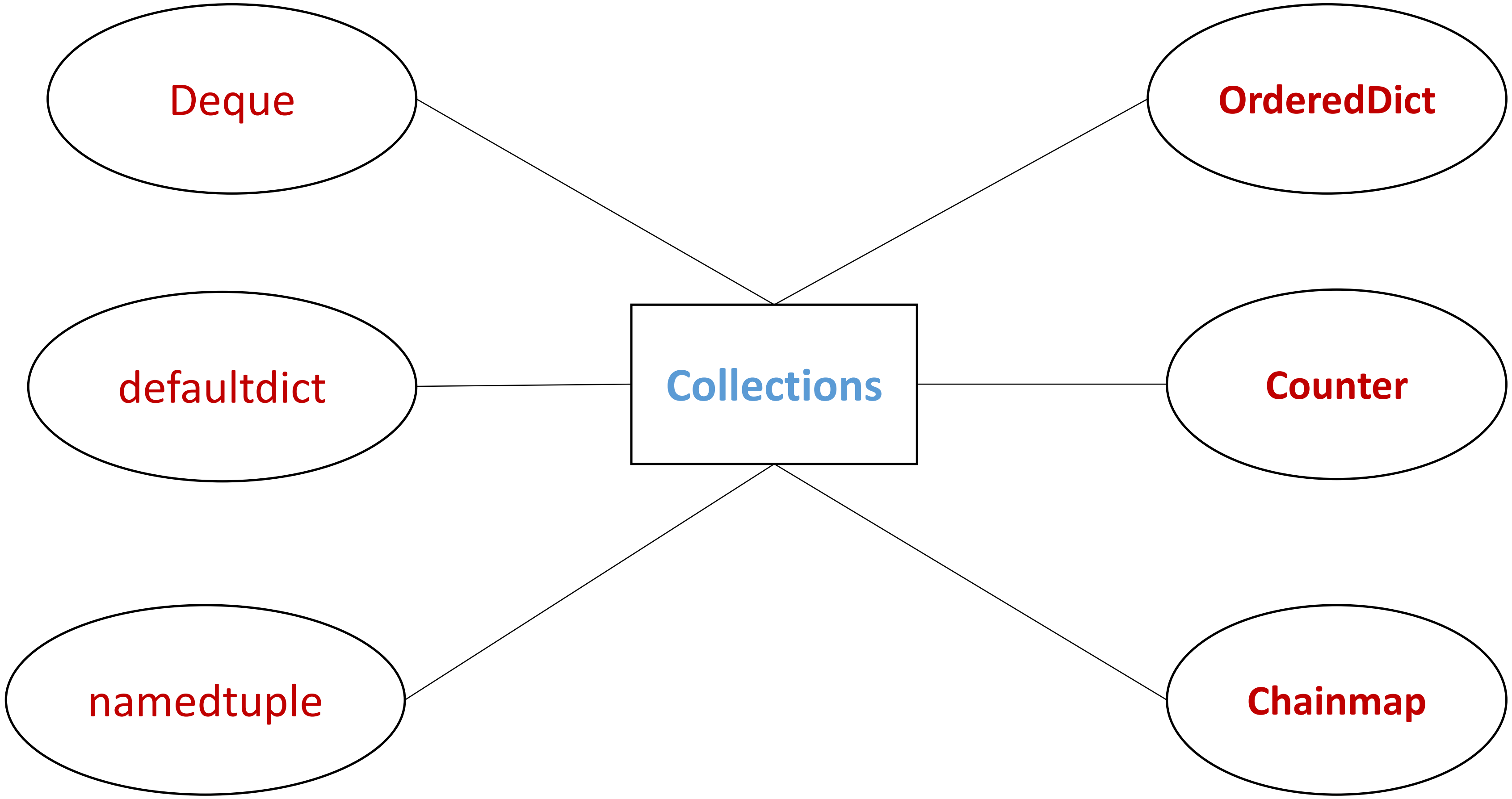
2. Segment image into small size chips,

3. perform flipping, rotation on chips using multiprocessing.

# Collections

**Container** : A Container is a type of object that can be used to hold multiple items while simultaneously providing a way to access and iterate over them, such as a Tuple, list, dictionary.

## Collections Module



# Namedtuple : Improving Code Readability

A function for creating subclasses of tuple that provides named fields that allow accessing items by name while keeping the ability to access items by index .

```
# The field values are passed as a string seperated by ' '
from collections import namedtuple
movie = namedtuple('movie','genre rating lead_actor')

#create instances of movie
ironman = movie(genre='action',rating=8.5,lead_actor='robert downey')
titanic = movie(genre='romance',rating=8,lead_actor='leonardo')
seven = movie(genre='crime',rating=9,lead_actor='Brad Pitt')

#Access the fields
print(titanic.genre)
print(seven.lead_actor)
print(ironman.rating)
```

# Deque : Building Efficient Queues and Stacks

A sequence-like collection that supports efficient addition and removal of items from either end of the sequence.

```
# Importing the deque
from collections import deque

# Initialization
l = ['Hi', 'This', 'is', 'Python', 'Training', 'Class']
myDeq = deque(l)

# Printing the deque
print(myDeq)
```

```
deque(['Hi', 'This', 'is', 'Python', 'Training', 'Class'])
```

```
# Inserting at both the ends
myDeq.append("!!")
myDeq.appendleft("!!")
print(myDeq)
```

```
deque(['!!', 'Hi', 'This', 'is', 'Python', 'Training', 'Class', '!!'])
```

```
# Removing from both the ends
myDeq.pop()
myDeq.popleft()
print(myDeq)
```

```
deque(['Hi', 'This', 'is', 'Python', 'Training', 'Class'])
```

```
# Insertion at position 1
myDeq.insert(1, "User")
print(myDeq)
```

```
deque(['Hi', 'User', 'This', 'is', 'Python', 'Training', 'Class'])
```

```
# Insertion of Value User
myDeq.remove("User")
print(myDeq)
```

```
deque(['Hi', 'This', 'is', 'Python', 'Training', 'Class'])
```

```
print("count of Python in deque is : ", myDeq.count("Python"))
```

```
count of Python in deque is :  1
```

```
# Deque Reversal
myDeq.reverse()
print(myDeq)
```

```
deque(['Class', 'Training', 'Python', 'is', 'This', 'Hi'])
```

```
# Deque Rotation
myDeq.rotate(1)
print(myDeq)
```

```
deque(['Hi', 'Class', 'Training', 'Python', 'is', 'This'])
```

# defaultdict : Handling Missing Keys

A dictionary subclass for constructing default values for missing keys and automatically adding them to the dictionary.

```
favorites = {"pet": "dog", "color": "blue", "language": "Python"}

favorites["fruit"]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'fruit'
```

```
#Importing the dictionary
from collections import defaultdict

# Initializing the dictionary
dic = defaultdict(int)

# Adding the values
dic[1] = 'a'
dic[2] = 'b'
dic[3] = 'c'

# Printing the dictionary
print(dic[4])
```

```
0
```

# OrderedDict : Keeping Your Dictionaries Ordered

A dictionary subclass that keeps the key-value pairs ordered according to when the keys are inserted.

```
from collections import OrderedDict

letters = OrderedDict(b=2, d=4, a=1, c=3)
print(letters)
```

```
OrderedDict([('b', 2), ('d', 4), ('a', 1), ('c', 3)])
```

```
# Move b to the right end
letters.move_to_end("b")
print(letters)
```

```
OrderedDict([('d', 4), ('a', 1), ('c', 3), ('b', 2)])
```

```
# Move b to the left end
letters.move_to_end("b", last=False)
print(letters)
```

```
OrderedDict([('b', 2), ('d', 4), ('a', 1), ('c', 3)])
```

```
# Sort letters by key
for key in sorted(letters):
    letters.move_to_end(key)
print(letters)
```

```
OrderedDict([('a', 1), ('b', 2), ('c', 3), ('d', 4)])
```

```
from collections import OrderedDict

# Regular dictionaries compare the content only
letters_0 = dict(a=1, b=2, c=3, d=4)
letters_1 = dict(b=2, a=1, d=4, c=3)
print(letters_0 == letters_1)
```

```
True
```

```
# Ordered dictionaries compare content and order
letters_0 = OrderedDict(a=1, b=2, c=3, d=4)
letters_1 = OrderedDict(b=2, a=1, d=4, c=3)
print(letters_0 == letters_1)
```

```
False
```

```
letters_2 = OrderedDict(a=1, b=2, c=3, d=4)
print(letters_0 == letters_2)
```

```
True
```

# Counter: Counting Objects in One Go

A dictionary subclass that supports convenient counting of unique items in a sequence or iterable.

```python
from collections import Counter

letters = Counter("mississippi")
print(letters)
```

```
Counter({'i': 4, 's': 4, 'p': 2, 'm': 1})
```

```python
# Update the counts of m and i
letters.update(m=3, i=4)
print(letters)
```

```
Counter({'i': 8, 'm': 4, 's': 4, 'p': 2})
```

```python
# Add a new key-count pair
letters.update({"a": 2})
print(letters)
```

```
Counter({'i': 8, 'm': 4, 's': 4, 'p': 2, 'a': 2})
```

```python
# Update with another counter
letters.update(Counter(["s", "s", "p"]))
print(letters)
```

```
Counter({'i': 8, 's': 6, 'm': 4, 'p': 3, 'a': 2})
```

```python
from collections import Counter

inventory = Counter(dogs=23, cats=14, pythons=7)
adopted = Counter(dogs=2, cats=5, pythons=1)
inventory.subtract(adopted)
print(inventory)
```

```
Counter({'dogs': 21, 'cats': 9, 'pythons': 6})
```

```python
new_pets = {"dogs": 4, "cats": 1}
inventory.update(new_pets)
print(inventory)
```

```
Counter({'dogs': 25, 'cats': 10, 'pythons': 6})
```

```python
inventory = inventory-Counter(dogs=2,cats=3,pythons=1)
print(inventory)
```

```
Counter({'dogs': 23, 'cats': 7, 'pythons': 5})
```

```python
new_pets = {"dogs": 4, "pythons": 2}
inventory += new_pets
print(inventory)
```

```
Counter({'dogs': 27, 'cats': 7, 'pythons': 7})
```

# ChainMap: Chaining Dictionaries Together

A dictionary-like class that groups multiple dictionaries and other mappings together to create a single object. In other words, it takes several mappings and makes them logically appear as one.

```
from collections import ChainMap

numbers = {"1": 1, "2": 2}
letters = {"a": "A", "b": "B"}

alpha_nums = ChainMap(numbers, letters)
alpha_nums
```

```
ChainMap({'1': 1, '2': 2}, {'a': 'A', 'b': 'B'})
```

```
alpha_nums.maps
```

```
[{'1': 1, '2': 2}, {'a': 'A', 'b': 'B'}]
```

```
#Add a new key pair in first dict
alpha_nums["3"] = 3
alpha_nums
```

```
ChainMap({'1': 1, '2': 2, '3': 3}, {'a': 'A', 'b': 'B'})
```

```
#To add a new key-pair in second dict
alpha_nums.maps[1]["c"] = "C"
alpha_nums
```

```
ChainMap({'1': 1, '2': 2, '3': 3}, {'a': 'A', 'b': 'B', 'c': 'C'})
```

```
#Pop key that exists in first dict
alpha_nums.pop("2")
alpha_nums
```

```
ChainMap({'1': 1, '3': 3}, {'a': 'A', 'b': 'B', 'c': 'C'})
```

```
del alpha_nums.maps[1]["a"]
alpha_nums
```

```
ChainMap({'1': 1, '3': 3}, {'b': 'B', 'c': 'C'})
```

```
#Clear the dictionary
alpha_nums.clear()
alpha_nums
```

```
ChainMap({}, {'b': 'B', 'c': 'C'})
```

# Memory Mapped Files

# Regular File I/O

```
def regular_io(filename):
    with open(filename, mode="r", encoding="utf8") as file_obj:
        text = file_obj.read()
        print(text)
```

Reads file into physical memory and prints it to the screen

**1. Transferring** control to the kernel operating system code with system calls(System calls are the API that the operating system provides to allow your program to go from user space to kernel space)

**2. Interacting** with the physical disk where the file resides

**3. Copying** the data into different buffers between user space and kernel space

**4**. All these layers add **latency** and can slow down your program.

**Solution** : Memory Mapped Files

# Memory Mapped Files

- Memory-mapping typically improves I/O performance because it uses lower-level operating system APIs to store file contents directly in physical memory. It does not involve a separate system call for each access and does not require copying data between buffers – the memory is accessed directly.

- Memory-mapping uses the concept of virtual memory to make it appear to the program that a large file has been loaded to main memory.

- But in reality the file is only present on the disk. The operating system just maps the address of the file into the program’s address space so that program can access the file.

# Python mmap: Improved File I/O With Memory Mapping

mmap Object Creation Syntax :

*mmap.mmap(file_obj.fileno(), length=0, access=mmap.ACCESS_READ)*

1. File discriptor, which comes from the fileno() method of a regular file object

2. length=0. This is the length in bytes of the memory map. 0 is a special value indicating that the system should create a memory map large enough to hold the entire file.

3. The access argument tells the operating system how you're going to interact with the mapped memory. The options are ACCESS_READ, ACCESS_WRITE, ACCESS_COPY.

- **ACCESS_READ** creates a read-only memory map.
- **ACCESS_WRITE** specifies write-through semantics, meaning the data will be written through memory and persisted on disk.
- **ACCESS_COPY** does not write the changes to disk.

# Python mmap: Improved File I/O With Memory Mapping

```python
import os
filename = "/data/hemant/abc_log.txt"
isFile = os.path.isfile(filename)
def regular_io(filename):
    with open(filename, mode="r", encoding="utf8") as file_obj:
        text = file_obj.read()
```

```python
import mmap

def mmap_io(filename):
    with open(filename, mode="r", encoding="utf8") as file_obj:
        with mmap.mmap(file_obj.fileno(), length=0, access=mmap.ACCESS_READ) as mmap_obj:
            text = mmap_obj.read()
```

```python
import timeit
timeit.repeat("regular_io(filename)",repeat=3,number=1,
        setup="from __main__ import regular_io, filename")
```

```
[0.3878343170072185, 0.2810663979907986, 0.23127491801278666]
```

```python
timeit.repeat("mmap_io(filename)",repeat=3,number=1,
...     setup="from __main__ import mmap_io, filename")
```

```
[0.16993256099522114, 0.1392672919901088, 0.13084593899839092]
```

# mmap Objects as Strings
# Search a Memory-Mapped File

Memory mapping transparently loads the file contents into memory as a string. So string operations can be performed on mmap objects like slicing , searching etc.

```
import mmap

def regular_io_find(filename):
    with open(filename, mode="r", encoding="utf-8") as file_obj:
        text = file_obj.read()
        text.find(" the ")

def mmap_io_find(filename):
    with open(filename, mode="r", encoding="utf-8") as file_obj:
        with mmap.mmap(file_obj.fileno(), length=0, access=mmap.ACCESS_READ) as mmap_obj:
            mmap_obj.find(b" the ")
```

```
import timeit
timeit.repeat("regular_io_find(filename)",repeat=3,number=1,
        setup="from __main__ import regular_io_find, filename")
```

```
[0.3952411329955794, 0.2799205269984668, 0.23502482900221366]
```

```
timeit.repeat("mmap_io_find(filename)",repeat=3,number=1,
        setup="from __main__ import mmap_io_find, filename")
```

```
[0.0007837999874027446, 0.0004058490012539507, 0.00045438400411512 7]
```

# Memory-Mapped Objects as Files

A memory-mapped file is part string and part file, so mmap also allows you to perform common file operations like seek(), tell(), and readline().

```
import mmap
def regular_io_find_and_seek(filename):
    with open(filename, mode="r", encoding="utf-8") as file_obj:
        file_obj.seek(10000)
        text = file_obj.read()
        text.find(" the ")
```

```
def mmap_io_find_and_seek(filename):
    with open(filename, mode="r", encoding="utf-8") as file_obj:
        with mmap.mmap(file_obj.fileno(), length=0, access=mmap.ACCESS_READ) as mmap_obj:
            mmap_obj.seek(10000)
            mmap_obj.find(b" the ")
```

```
import timeit
timeit.repeat("regular_io_find_and_seek(filename)",repeat=3,number=1,
        setup="from __main__ import regular_io_find_and_seek, filename")
```

```
[1.0732763529958902, 0.8469445810042089, 0.8576581630040891]
```

```
timeit.repeat("mmap_io_find_and_seek(filename)",repeat=3,number=1,
        setup="from __main__ import mmap_io_find_and_seek, filename")
```

```
[0.00016790399968158454, 0.00021273999300319701, 0.0004815139982383698]
```

# Writing a Memory-Mapped File With Python's mmap

Memory mapping is most useful for reading files, but you can also use it to write files. The mmap API for writing files is very similar to regular file I/O except for a few differences.

```
import mmap
def mmap_io_write(filename, text):
    with open(filename, mode="w", encoding="utf-8") as file_obj:
        with mmap.mmap(file_obj.fileno(), length=0, access=mmap.ACCESS_WRITE) as mmap_obj:
            mmap_obj.write(text)
```

```
import mmap
def mmap_io_write(filename):
    with open(filename, mode="r+") as file_obj:
        with mmap.mmap(file_obj.fileno(), length=0, access=mmap.ACCESS_WRITE) as mmap_obj:
            mmap_obj[10:16] = b"python"
            mmap_obj.flush()
```

# Search and Replace Text

Memory mapped file data is a string of mutable bytes. So it is much more straightforward and efficient to write code that searches and replaces data in a file.

```python
import mmap
import os
import shutil

def regular_io_find_and_replace(filename):
    with open(filename, "r", encoding="utf-8") as orig_file_obj:
        with open("tmp.txt", "w", encoding="utf-8") as new_file_obj:
            orig_text = orig_file_obj.read()
            new_text = orig_text.replace(" the ", " eht ")
            new_file_obj.write(new_text)

    shutil.copyfile("tmp.txt", filename)
    os.remove("tmp.txt")

def mmap_io_find_and_replace(filename):
    with open(filename, mode="r+", encoding="utf-8") as file_obj:
        with mmap.mmap(file_obj.fileno(), length=0, access=mmap.ACCESS_WRITE) as mmap_obj:
            orig_text = mmap_obj.read()
            new_text = orig_text.replace(b" eht ", b" the ")
            mmap_obj[:] = new_text
            mmap_obj.flush()
```

```python
import timeit
timeit.repeat("regular_io_find_and_replace(filename)",repeat=3,number=1,
        setup="from __main__ import regular_io_find_and_replace, filename")
```

```
[4.253578268006095, 3.1810569380031666, 3.3544113550015027]
```

```python
timeit.repeat("mmap_io_find_and_replace(filename)",repeat=3,number=1,
        setup="from __main__ import mmap_io_find_and_replace, filename")
```

```
[2.264397455001017, 1.5541641849995358, 1.598239985993132]
```

# Working with Binary Data

# Working with binary data

- Strings represent text, bytes objects represent binary data (i.e. images, video, or anything else you could represent on a computer).
- The **bytes** type in Python is immutable and stores a sequence of values ranging from 0-255 (8-bits). You can get the value of a single byte by using an index like an array, but the values can not be modified.
- To create a mutable object you need to use the **bytearray** type ( To modify a set of bytes).

```
text ="hello"
list(text)
```

```
['h', 'e', 'l', 'l', 'o']
```

```
data=b"hello"
list(data)
```

```
[104, 101, 108, 108, 111]
```

```
bytestr=bytes(b'abc')
# initializing a string with b
# makes it a binary string
bytestr
```

```
b'abc'
```

```
bytestr[0]
```

```
97
```

```
bytestr[0] = 98
```

```
TypeError: 'bytes' object does not support item assignment
```

```
# Cast bytes to bytearray
mutable_bytes = bytearray(b'\x00\x0F')

# Bytearray allows modification
mutable_bytes[0] = 255
mutable_bytes.append(255)
print(mutable_bytes)

# Cast bytearray back to bytes
immutable_bytes = bytes(mutable_bytes)
print(immutable_bytes)
```

```
bytearray(b'\xff\x0f\xff')
b'\xff\x0f\xff'
```

# Writing Bytes to a file

In Python, files are opened in text mode by default. To open files in binary mode, when specifying a mode, add 'b' to it.

```
# Pass "wb" to write a new file, or "ab" to append
with open("test.txt", "wb") as binary_file:
    # Write text or bytes to the file
    binary_file.write("Write text by encoding\n".encode('utf8'))
    num_bytes_written = binary_file.write(b'\xDE\xAD\xBE\xEF')
    print("Wrote %d bytes." % num_bytes_written)
```

## Reading Bytes from a file

```
with open("test.txt", "rb") as binary_file:
    # Read the whole file at once
    data = binary_file.read()
    print(data)
```

```
b'Write text by encoding\n\xde\xad\xbe\xef'
```

## Reading file line by line

```
with open("test.txt", "rb") as text_file:
    # One option is to call readline() explicitly
    # single_line = text_file.readline()

    # It is easier to use a for loop to iterate each line
    for line in text_file:
        print(line)
```

```
b'Write text by encoding\n'
b'\xde\xad\xbe\xef'
```

# Seeking a specific position in a file

```
# Seek can be called one of two ways:
#   x.seek(offset)
#   x.seek(offset, starting_point)

# starting_point can be 0, 1, or 2
# 0 - Default. Offset relative to beginning of file
# 1 - Start from the current position in the file
# 2 - Start from the end of a file (will require a negative offset)

with open("test.txt", "rb") as binary_file:
    # Seek a specific position in the file and read N bytes
    binary_file.seek(0, 0)  # Go to beginning of the file
    couple_bytes = binary_file.read(2)
    print(couple_bytes)
```

```
b'Wr'
```

# Getting the size of a file

```
import os
file_length_in_bytes = os.path.getsize("test.txt")
print(file_length_in_bytes)
```

```
27
```

# Getting system byte order

```
# Find out what byte order your system uses
import sys
print("Native byteorder: ", sys.byteorder)
```

```
Native byteorder:  little
```

# Integer to Bytes

```
i = 16
# Create two byte from the integer 16
two_byte = i.to_bytes(2, byteorder='big', signed=False)
print(two_byte)
```

```
b'\x00\x10'
```

```
i = 16
# Create two bytes from the integer
two_byte = i.to_bytes(2, byteorder='little', signed=False)
print(two_byte)
```

```
b'\x10\x00'
```

```
i = -16
# Create two bytes from the integer
two_byte = i.to_bytes(2, byteorder='little', signed=True)
print(two_byte)
```

```
b'\xf0\xff'
```

```
# Create bytes from a list of integers with values from 0-255
bytes_from_list = bytes([255, 254, 253, 252])
print(bytes_from_list)
```

```
b'\xff\xfe\xfd\xfc'
```

```
# Create a byte from a base 2 integer
one_byte = int('11110000', 2)
print(one_byte)
```

# Bytes to Integer

```
# Create an int from bytes. Default is unsigned.
some_bytes = b'\x00\xF0'
i = int.from_bytes(some_bytes, byteorder='big')
print(i)
```

```
240
```

```
# Create a signed int
i = int.from_bytes(b'\x00\x0F', byteorder='big', signed=True)
print(i)
```

```
15
```

**to_bytes** – Returns an array of bytes representing an integer.

**from_bytes** – Returns integer represented by given array of bytes.

**Byte Order** : “**big**” – most significant byte at the beginning of the byte array.

“**little**” - most significant byte at the end of the byte array.

**Signed** – indicates whether 2’s complement is used to represent the integer.

# Character Encoding

Character Encodings are a way to assign values to bytes that represent a certain character in that scheme. Some encodings are ASCII(probably the oldest), Latin, and UTF-8.

```
binary_data = bytes([65, 66, 67])
# ASCII values for A, B, C
text = binary_data.decode('utf-8')
print(text)
```

```
ABC
```

```
# Text to Binary
message = "Hello"   # str
binary_message = message.encode('utf-8')
print(type(binary_message))  # bytes
```

```
<class 'bytes'>
```

# Format Strings

Format strings can be helpful to visualize or output byte values. Format strings require an integer value so the byte will have to be converted to an integer first.

```
a_byte = b'\xff'  # 255
i = ord(a_byte)   # Get the integer value of the byte

bin = "{0:b}".format(i) # binary: 11111111
hex = "{0:x}".format(i) # hexadecimal: ff
oct = "{0:o}".format(i) # octal: 377

print(i)
print(bin)
print(hex)
print(oct)
```

# Bitwise Operations

In Python, bitwise operators are used to perform bitwise calculations on integers. The integers are first converted into binary and then operations are performed on bit by bit, hence the name bitwise operators.

```
byte1 = int('11110000', 2)  # 240
byte2 = int('00001111', 2)  # 15
byte3 = int('01010101', 2)  # 85

# AND
print(byte1 & byte2)
```

```
0
```

```
# OR
print(byte1 | byte2)
```

```
255
```

```
# XOR
print(byte1 ^ byte3)
```

```
165
```

```
# Shifting right will lose the right-most bit
print(byte2 >> 3)
```

```
1
```

```
# Shifting left will add a 0 bit on the right side
print(byte2 << 1)
```

```
30
```

```
# See if a single bit is set
bit_mask = int('00000001', 2)  # Bit 1
print(bit_mask & byte1)  # Is bit set in byte1?
print(bit_mask & byte2)  # Is bit set in byte2?
```

```
0
1
```

# Struct Module

**Struct** module is used to convert the native data types of *Python* into **string of bytes** and vice versa. This module performs conversions between Python values and C structs represented as Python bytes objects.

The module’s functions and objects can be used for two largely distinct applications, data exchange with external sources (files or network connections), or data transfer between the Python application and the C layer.

```python
import struct
# struct.pack () - Packing values to Python byte-string (byte object)
# The first parameter is the format string. Here it specifies the data is structured
# with a single four-byte integer followed by two characters.
# The rest of the parameters are the values for each item in order
binary_data = struct.pack("icc", 8499000, b'A', b'Z')
print(binary_data)
```

```
b'8\xaf\x81\x00AZ'
```

```python
# When unpacking, you receive a tuple of all data in the same order
tuple_of_data = struct.unpack("icc", binary_data)
print(tuple_of_data)
```

```
(8499000, b'A', b'Z')
```

# Struct Module

**struct.calcsize() :**Return the size of the struct corresponding to the given format.

```
import struct
print("The size of 3 integer is :", struct.calcsize('iii'))
print("The size of 5 char is :", struct.calcsize('ccccc'))
print("The total size is :", struct.calcsize('ffiicc'))
```

```
The size of 3 integer is : 12
The size of 5 char is : 5
The total size is : 18
```

```
# diff.py - Do two files match?
import sys

with open(sys.argv[1], 'rb') as file1, \
        open(sys.argv[2], 'rb') as file2:
    data1 = file1.read()
    data2 = file2.read()

if data1 != data2:
    print("Files do not match.")
else:
    print("Files match.")
```

```
#is_jpeg.py - Does the file have a JPEG binary signature?
import sys
import binascii

jpeg_signatures = [
    binascii.unhexlify(b'FFD8FFD8'),
    binascii.unhexlify(b'FFD8FFE0'),
    binascii.unhexlify(b'FFD8FFE1')
]

with open(sys.argv[1], 'rb') as file:
    first_four_bytes = file.read(4)

    if first_four_bytes in jpeg_signatures:
        print("JPEG detected.")
    else:
        print("File does not look like a JPEG.")
```

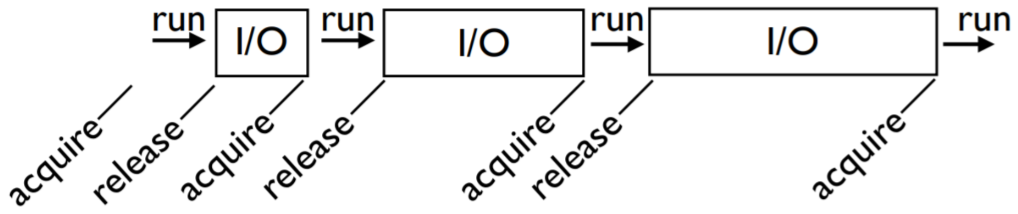
# Thank You

# Global Interpreter Lock(GIL)

| I/O Bound Processing | CPU Bound Processing |
|---|---|

## I/O Bound Processing

- Whenever a thread runs, it holds the GIL.However, the GIL is released on blocking I/O




- So, any time a thread is forced to wait, other "ready" threads get their chance to run
- Threads are useful for I/O-bound processing or Limit CPU-bound processing to C extensions (that release the GIL)

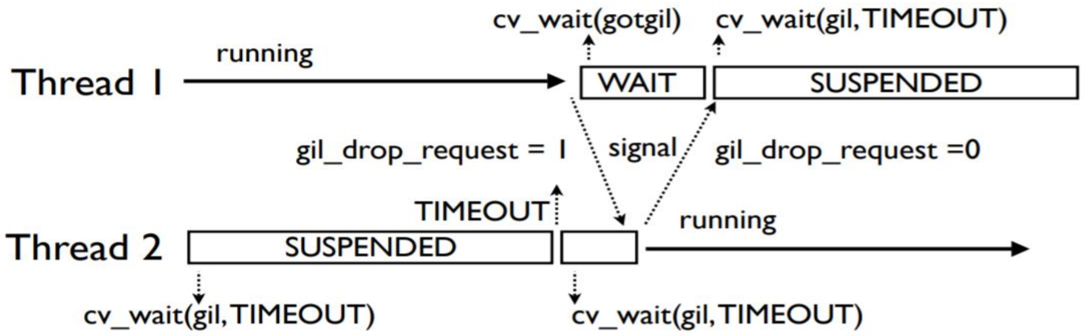
## CPU Bound Processing

- there is a global variable

  */* Python/ceval.c */*

  ...

  *static volatile int gil_drop_request = 0;*
- A thread runs until the value gets set to 1. At which point, the thread must drop the GIL.




- So, the timeout sequence happens over and over again as CPU-bound threads execute

# Thread Synchronization
# Mutex Locks

Synchronization ensures that two or more concurrent processes/threads do not simultaneously execute some particular program segment where the shared resources are accessed known as critical section.

## Mutex Locks

- Mutual Exclusion Lock

  ***m = threading.Lock()***

- Primarily used to synchronize threads so that only one thread can make modifications to shared data at any given time
- There are two basic operations

  ***m.acquire() # Acquire the lock***

  ***m.release() # Release the lock***

- Only one thread can successfully acquire the lock at any given time
- If another thread tries to acquire the lock when its already in use, it gets blocked until the lock is released

## Use of Mutex Locks

- Commonly used to enclose critical sections.

```
x = 0
x_lock = threading.Lock()

Thread-1                    Thread-2
--------                    --------
...                         ...
x_lock.acquire()            x_lock.acquire()

x = x + 1                   x = x - 1

x_lock.release()            x_lock.release()
...                         ...
```

Critical Section

# Thread Synchronization
# Semaphores

- A counter-based synchronization primitive

    *sema= threading.Semaphore(n) # Create a semaphore*

    *sema.acquire() # Acquire*

    *sema.release() # Release*

• acquire() - Waits if the count is 0, otherwise decrements the count and continues

• release() - Increments the count and signals waiting threads (if any)

Use of Semaphores

- Using a semaphore to limit resources

```
sema = threading.Semaphore(5) # Max: 5-threads
def fetch_page(url):
    sema.acquire()
    try:
        u = urllib.urlopen(url)
        return u.read()
    finally:
        sema.release()
```

- • In this example, only 5 threads can be executing the function at once (if there are more, they will have to wait)