Aprameyah
Technologies Pvt. Ltd.

More Python

List Comprehensions

- List comprehensions are pythons way of writing list processing code with minimal nesting

```
S = [x**2 for x in range(10)]
V = [2**i for i in range(13)]
M = [x for x in S if x % 2 == 0]

noprimes = []
primes = []
for i in range(2, 8):
    for j in range(i*2, 50, i):
        noprimes.append(j)

noprimes = [j for i in range(2, 8) for j in range(i*2, 50, i)]
```

Aprameyah
Technologies Pvt. Ltd.

List Comprehensions

```
for x in range(2,50):
    if x not in noprimes:
        primes.append(x)

primes = [x for x in range(2, 50) if x not in noprimes]
```

```
words = 'The quick brown fox jumps over the lazy dog'.split()
stuff = []
for word in words:
    temp = [word.upper(), word.lower(), len(word)]
    stuff.append(temp)
print(stuff)

stuff = [[w.upper(), w.lower(), len(w)] for w in words]
print(stuff)
```

Aprameyah
Technologies Pvt. Ltd.

Lambda Operator

- Used to define expressions as function references

```
f = lambda x, y : x + y

f(1,1)
```

- Creates an anonymous function

Map, Filter, Reduce

- Map means transformation: seq2 = map(func, sequence)

```
temp = [32,35,39,27]
farh = []
for T in temp:
    f = ((float(9)/5)*T + 32)
    farh.append(f)
print(farh)

---------------------------------
farh = map(lambda T: ((float(9)/5)*T + 32), temp)
for f in farh:
    print(f)
---------------------------------
def farhaniet(T):
    return ((float(9)/5)*T + 32)

farh = map(farhaniet, temp)
for f in farh:
    print(f)
```

Filter

- Filter eliminates part of the list

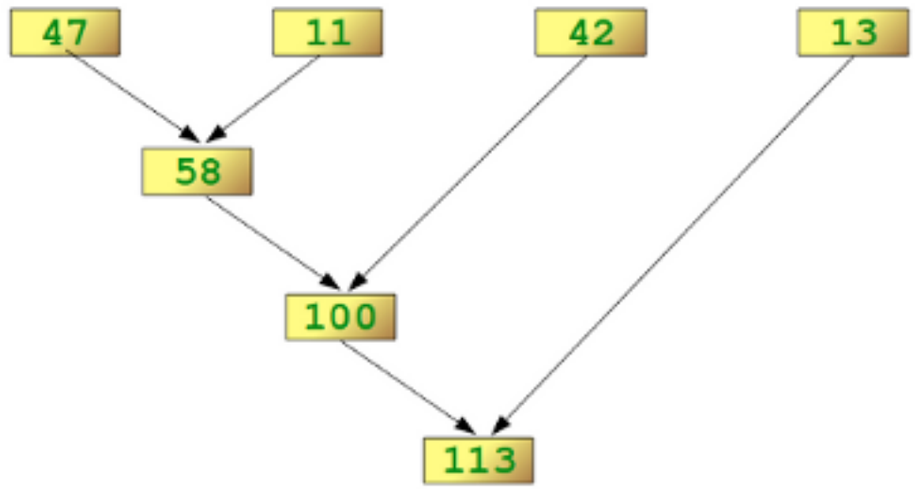
```
even = []
for i in range(100):
    if not i % 2:
        even.append(i)
print(even)
------------------------------
even = filter(lambda x: not x%2, range(100))
print(even)
for e in even:
    print (e,end=' ‘)
------------------------------
def even_filter(x):
   return not x%2
even = filter(even_filter, range(100))
print(even)
for e in even:
    print (e,end=' ')
```

Reduce

- Reduce function produces a single value out of a sequence

```
sum_=0
for i in range(100):
    sum_+=i
print (sum_)
```

```
from _functools import reduce
sum_ = reduce(lambda x,y:x+y,range(100))
print("Sum2: "+str(sum_))
```



Aprameyah Technologies Pvt. Ltd.

Lets Try It (19)

- Transform this logic to use map/filter/reduce

```
class GzipRegularCommand(Command):
    """Implements the a command to gzip one or more files without using map/reduce/filter"""

    def __init__(self, command):
        #assuming second word would be the path
        if len(command.split(sep=" ")) >2 :
            self.source_pattern = command.split(sep=" ")[1]
            self.dest_folder = command.split(sep=" ")[2]
        else:
            raise CommandError("invalid gzip syntax. Only works in current directory. Source file pattern
can be regex. Usage: gzip sourceFilePattern destFolder")
```

```
    def execute(self):
        files = os.listdir()
        process_count =0
        for file_name in files:
            # --- Filter operation ---
            if os.path.isdir(file_name):
                continue
            match_obj = re.match(self.source_pattern,file_name)
            if not match_obj:
                continue
            # --- Map operation ---
            dest_file_name = os.getcwd()+"/temp/"+file_name+".gz"
            print("Compressing file: {} to {}".format(file_name,dest_file_name))
            with open(file_name, mode='rb') as source_file, gzip.open(dest_file_name, 'wb') as dest_file :
                dest_file.writelines(source_file)
                # --- Reduce operation ---
                process_count+=1
        print("{} files processed.".format(process_count))
```

What are Threads

- Threads are an independent path of execution in a single process.

- Threads can be started by other threads to trigger another parallel execution path (fork)

- Python threads are mapped to an operating system thread

Simple Thread

```
import threading

class MyThread (threading.Thread):
    def run(self):
        for i in range(100000):
            print("Value of i ",i)

thread1 = MyThread()
thread1.start()
print("Started thread")
for i in range(100000):
    print("Value of i ********** ",str(i))
```

Daemon Threads

- Threads that are meant to run background processess that support the main process
- Main process threads (user threads) keep the interpreter alive. Whereas daemon threads do not
- Try an example to demonstrate this concept

Timeit Module

- Timeit module is used to do time measurements for code. Concept is to do average time measurement over a number of execution

- Timer function needs a function to time and any setup needed

```
from timeit import Timer
t = Timer("sin_compute()", "from __main__ import sin_compute")
print("Sin computation: ",t.timeit(5))
```

Thread Racing

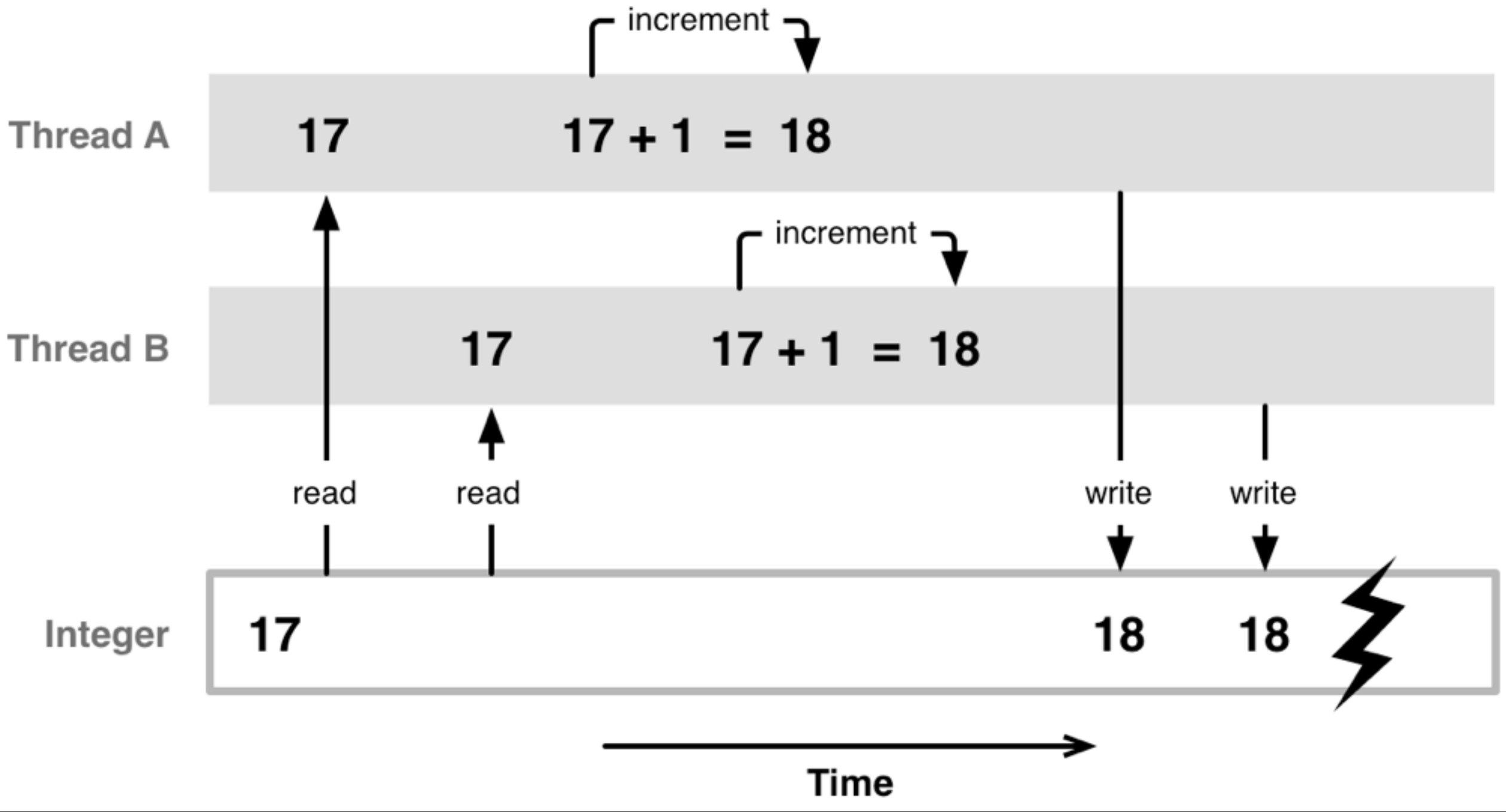
```
import threading
k=[]
k.append(0)

class MyThread (threading.Thread):
    def run(self):
        for i in range(100000):
            k[0] = i
            print("Value of i = ",k)
            if k[0]!=i:
                print("This cant print#################################")

thread1 = MyThread()
thread1.start()
print("Started thread");
for j in range(100000):
    k[0] = j
    print("********* Value of j = ",j)
    if k[0]!=j:
        print("This cant print#################################”)
```

repeat the above code for k=0 (an int instead of an array)

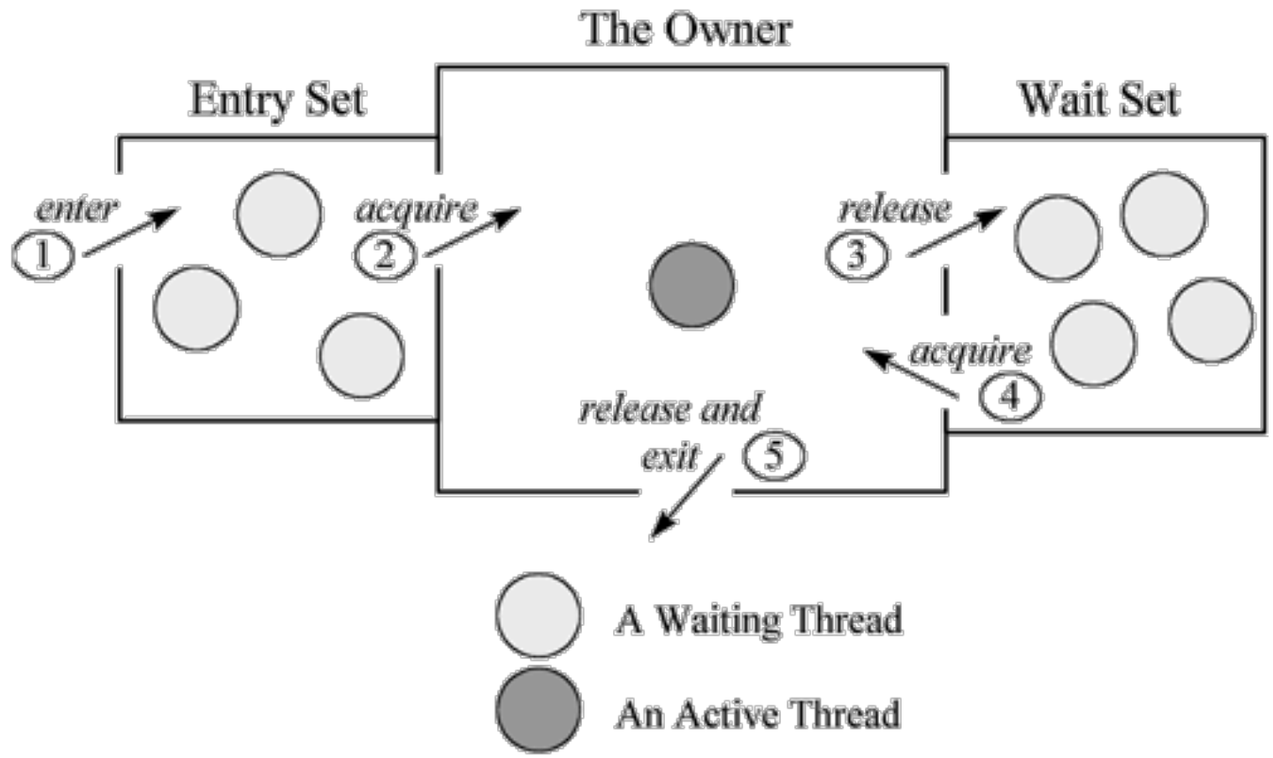
Increments

```
import threading
k=0

class MyThread (threading.Thread):
    def run(self):
        global k
        for i in range(1000000):
            k+=1

thread1 = MyThread()
thread1.start()
print("Started thread");
for j in range(1000000):
    k+=1
thread1.join()
print("Value of k %d" % k)
```

Thread Racing on Increments



Thread Sync


The Owner
Entry Set
Wait Set
enter
1
acquire
2
release
3
acquire
4
release and
exit
5
A Waiting Thread
An Active Thread

Conditions

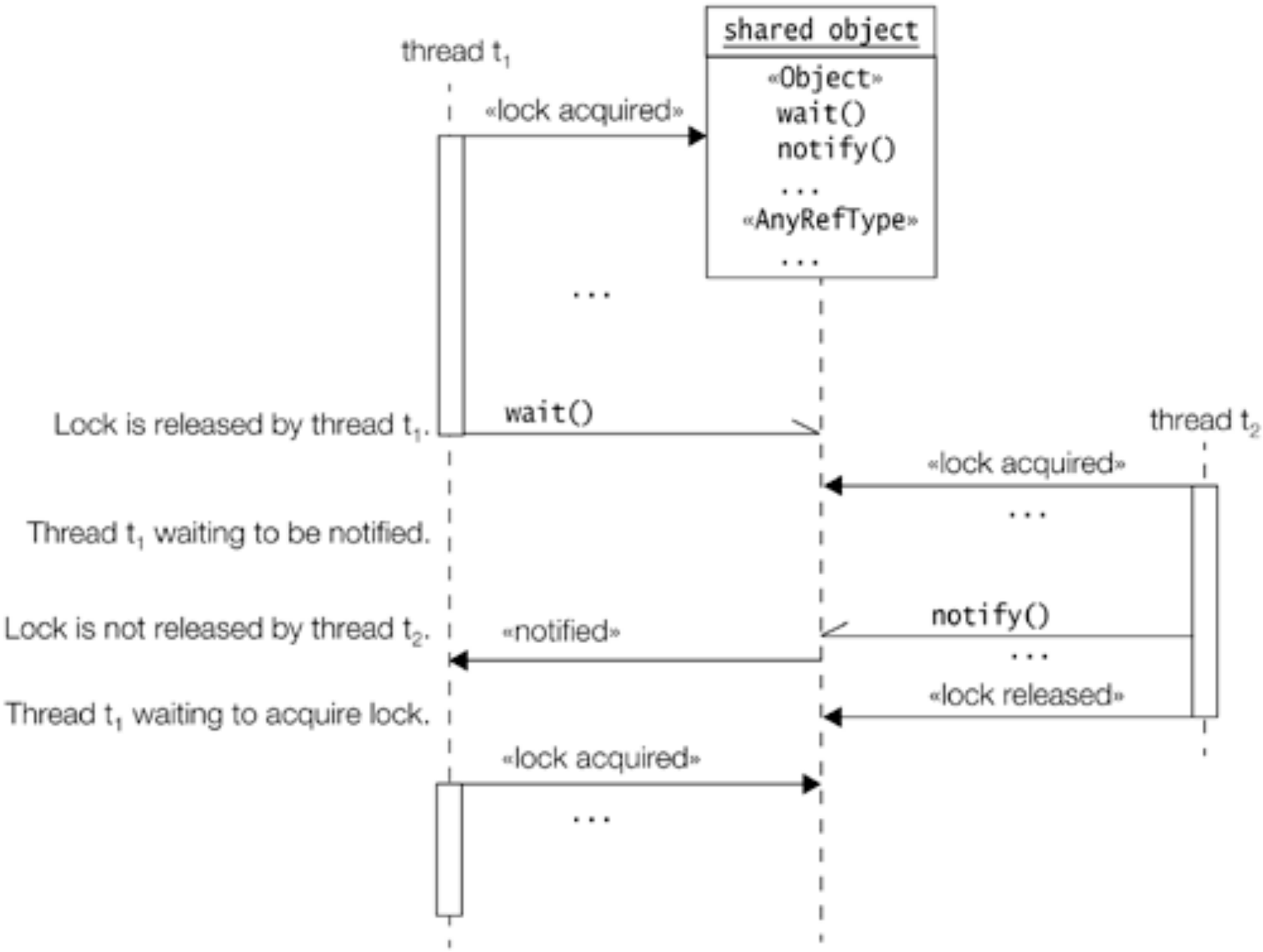
- Conditions object can be used to acquire and release a lock

```
from threading import Condition
import threading
k=[]
k.append(0)
# confirms to the context managemnt protocol
cv = Condition()

class MyThread (threading.Thread):
    def run(self):
        for i in range(100000):
            cv.acquire()
            k[0] = i
            print("Value of i = ",k)
            if k[0]!=i:
                print("This cant print####################################")
            cv.release()

thread1 = MyThread()
thread1.start()
print("Started thread");
for j in range(100000):
    cv.acquire()
    k[0] = j
    print("********* Value of j = ",j)
    if k[0]!=j:
        print("This cant print####################################")
    cv.release()
```

Typical Producer Consumer

```
from threading import Condition
import threading

cv = Condition()
data_list = []
class MyProducerThread (threading.Thread):
    def run(self):
        cv.acquire()
        while True:
            text = input("Enter text: ")
            data_list.append(text)
            cv.notify()
            print("Notified...")
        cv.release()

class MyConsumerThread (threading.Thread):
    def run(self):
        cv.acquire()
        while True:
            while len(data_list)==0 :
                print("Waiting...")
                cv.wait()
            print("Processing...")
            data = data_list[0]
            del data_list[0:len(data_list)]
            print("Consumer got data: ",data)
            cv.notify()
        cv.release()

thread1 = MyConsumerThread()
thread1.start()
thread2 = MyProducerThread()
thread2.start()
```

Wait-Notify


thread t_1
shared object
«Object»
wait()
notify()
...
«AnyRefType»
...
«lock acquired»
...
wait()
Lock is released by thread t_1.
thread t_2
«lock acquired»
...
Thread t_1 waiting to be notified.
notify()
Lock is not released by thread t_2.
«notified»
...
«lock released»
Thread t_1 waiting to acquire lock.
«lock acquired»
...

Wait-Notify Guarantees

- When a thread is notified it is not guaranteed to wake up AS SOON AS the lock is released
  - Any other thread could get the lock
- notifyAll() can be used to wake up more than one thread but no guarantee about the order in which it wakes threads up
- Some threads can get starved for a long time

Aprameyah Technologies Pvt. Ltd.

Thread Pools

- Thread pools limit the maximum number of threads allowed in a system

```
class ThreadPool:
    def __init__(self,N) :
        self.workerQueue = Queue()
        self.workerThreads = [None] * N
        #Start N Threads and keep them running
        for i in range(N) :
            self.workerThreads[i] = self.Worker(self.workerQueue)
            self.workerThreads[i].start()

    def addTask(self,runner):
        try :
            self.workerQueue.put(runner)
        except:
            pass

    class Worker(Thread):
        def __init__(self, workerQueue):
            Thread.__init__(self)
            self.workerQueue = workerQueue

        def run(self):
            while True :
                runner = self.workerQueue.get()
                runner()
```

Divide Work

- Compute sin() of all numbers from 1 to 10000000 and add it all up in two separate threads.. compare the time it takes to do the same in a single thread.

- Explore the concept of Join

Aprameyah
Technologies Pvt. Ltd.

GIL - Global Interpreter Lock

- is a mutex that prevents multiple native threads from executing Python bytecodes at once.

- This lock is necessary mainly because CPython's memory management is not thread-safe.

- Note that potentially blocking or long-running operations, such as I/O, image processing, and NumPy number crunching, happen outside the GIL

Multi Processing

- processes are spawned by creating a Process object and then calling its start() method. Process follows the API of threading.Thread.

- Not limited by GIL

```
from multiprocessing import Process

def f(name):
    print('hello', name)

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

Process Pool

- The Pool class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways.

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))
```

Aprameyah
Technologies Pvt. Ltd.

Process Pool Job Submissions

```
with Pool(processes=4) as pool:

    # print same numbers in arbitrary order
    for i in pool.imap_unordered(f, range(10)):
        print(i)

    # evaluate "f(10)" asynchronously
    res = pool.apply_async(f, [10])
    print(res.get(timeout=1))
```

Modules

- Modules help us organise our code into standard structure so that distribution utilities can be used to publish these modules to the world

- A module is simply a test file that contains python code.

  - Name of the file should end with .py

  - Python Package Index (PyPI) provides a central repository of third party python modules on the internet

Modules

- In order to publish the modules, we need to prepare a distribution (A collection of files)

- Interpreter looks for python modules in a bunch of locations set by environment variable PYTHONPATH

```
import sys

for s in sys.path:
    print(s)
```

Steps To Create Module

- Create a folder for the module & copy mymodule.py into it

- Create a file called setup.py that contains meta data about the distribution

```
from distutils.core import setup

setup(name='MyModule',
      version='1.0',
      description='My Module',
      author='Maruthi',
      author_email='maruthi@leviossa.com',
      url='http://www.leviossa.com/',
      py_modules=['mymodule']
     )
```

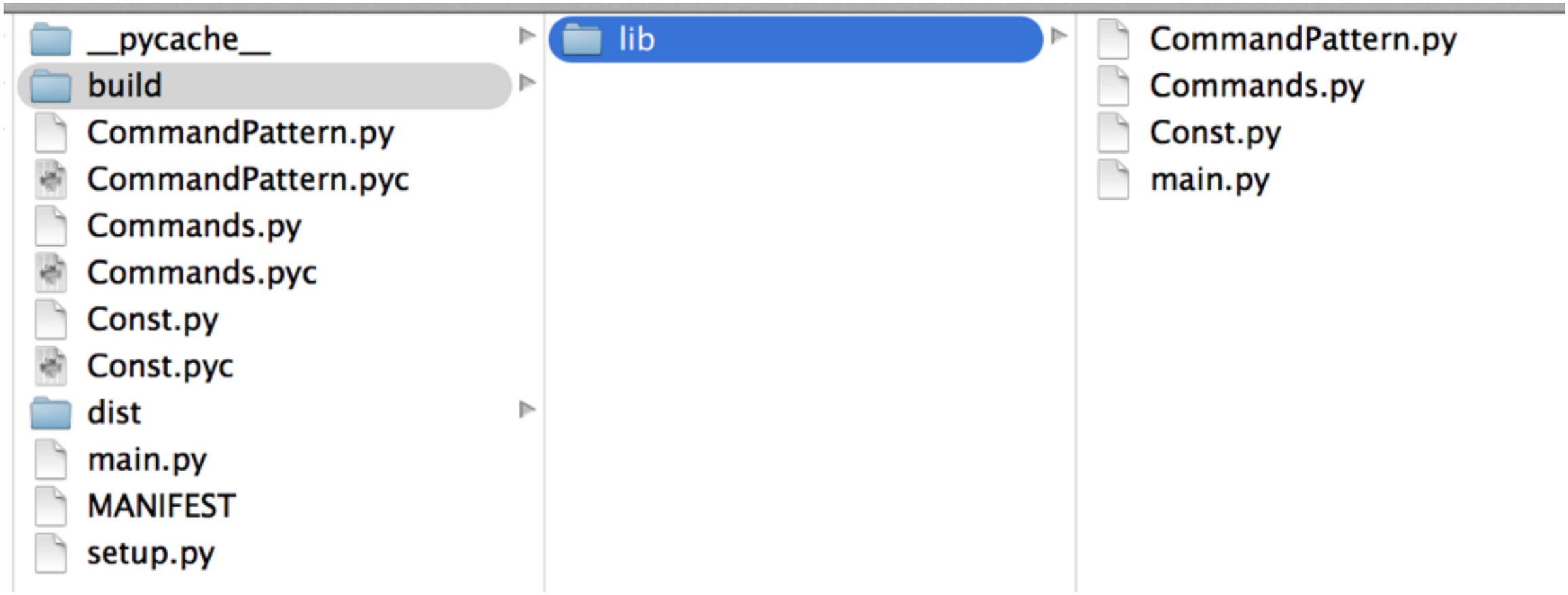
Steps To Create Module

- Build a distribution
  - python3 setup.py sdist
- Install into local
  - sudo python3 setup.py install
- Import it in any other python file to use it
  - import mymodule

Publishing Modules

- python3 setup.py register
    - Command line registration (You can do this online at pypi.python.org
- python3 setup.py sdist upload

Folders created by Build



Namespaces

- All code in python is associated with a namespace
- When we put code into a module, python automatically creates a namespace with the same name as the module
- so all functions in our module should be invoked as mymodule.myfunction (after “import mymodule") or as myfunction (after “from mymodule import myfunction”)
- BIFs belong to __builtins__ namespace and automatically imported into __main__ namespace