1)What does declarative programming paradigm highlight?
Declarative programming focuses on the end result,

- Declarative programming is a programming paradigm that expresses the **logic of a computation** without describing itscontrol flow.
- This paradigmoften considers programs as theories of a **formal logic**, and computations as deductions in that logic space.
- Declarative programming is often defined as any style of programming that is **not imperative**.
- Common declarative languages include those of database **query languages** (SQL), **logic programming, functional programming**, etc.

---

3)Give the code for ordered and unordered list in HTML

An ordered list starts with the `<ol>` tag. Each list item starts with the `<li>` tag.

The list items will be marked with numbers by default:

Example

```
<ol>
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ol>
```

Unordered HTML List

An unordered list starts with the `<ul>` tag. Each list item starts with the `<li>` tag.

The list items will be marked with bullets (small black circles) by default:

Example

```
<ul>
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ul>
```

9)List out the advantages of event driven Programming?

Flexibility

Programmers that use event-driven can be altered easily if the programmer wants something to be changed. This paradigm allows the programmer to produce a form of their requirements.

Suitability for Graphical Interfaces

Event-driven allows the user to select different tools from the toolbar to directly create what they need such as buttons, radio buttons, etc.

Simplicity of Programming

Event-driven can make programming easier for some by being able to directly edit the object you want the code for.

Easy to Find Natural Dividing Lines

it is easy to find natural dividing lines for unit testing infrastructure.

Highly Compostable

It is highly compostable.

Simple and Understandable

It allows for a very simple and understandable model for both sides of the DevOps Bridge.

Purely Procedural and Purely Imperative

Both purely procedural and purely imperative approaches get brittle as they grow in length and complexity.

A good way to Model Systems

It is one good way to model systems that need to be both asynchronous and reactive.

Allows for more Interactive Programs

It allows for more interactive programs. Almost all modern GUI programs use event-driven programming.

7)Differentiate Functional and Procedural Programming Paradigm

Functional Vs Procedural		
S.No	**Functional Paradigms**	**Procedural Paradigm**
1	Treats computation as the evaluation of mathematical functions avoiding state and mutable data	Derived from structured programming, based on the concept of modular programming or the *procedure call*
2	Main traits are Lambda calculus, formula, recursion, referential transparency	Main traits are Local variables, sequence, selection, iteration, and modularization
3	**Functional** programming focuses on **expressions**	**Procedural** programming focuses on **statements**
4	Often recursive. Always returns the same output for a given input.	The output of a routine does not always have a direct correlation with the input.
5	Order of evaluation is usually undefined.	Everything is done in a specific order.
6	Must be stateless. i.e. No operation can have side effects.	Execution of a routine may have side effects.
7	Good fit for parallel execution, Tends to emphasize a divide and conquer approach.	Tends to emphasize implementing solutions in a linear fashion.

8)Explain the following with suitable example
a. Map()
b. Reduce()
c. Filter ()
8) map() function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)
Syntax :
map(fun, iter)

```
# Python program to demonstrate working
# of map.

# Return double of n
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

Output :
[2, 4, 6, 8]

reduce

The reduce(fun,seq) function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along.

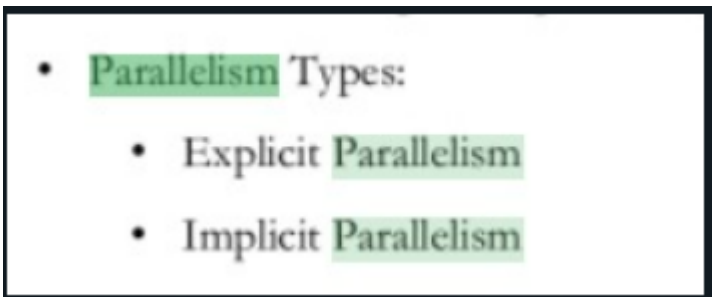
```
from functools import reduce

nums = [1, 2, 3, 4]
ans = reduce(lambda x, y: x + y, nums)
print(ans)


print(functools.reduce(lambda a, b: a if a > b else b, lis))
```

filter

The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.
syntax:
filter(function, sequence)

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# returns True if number is even
def check_even(number):
    if number % 2 == 0:
        return True

    return False

# Extract elements from the numbers list for which check_even() returns True
even_numbers_iterator = filter(check_even, numbers)

# converting to list
even_numbers = list(even_numbers_iterator)

print(even_numbers)

# Output: [2, 4, 6, 8, 10]
```

10)Explain types of Parallelism and also methods of parallelism in detail



Explicit parallelism

- Explicit Parallelism is characterized by the presence of explicit constructs in the programming language, aimed at describing (to a certain degree of detail) the way in which the parallel computation will take place.
- A wide range of solutions exists within this framework. One extreme is represented by the ``ancient'' use of basic, low level mechanisms to deal with parallelism--like fork/join primitives, semaphores, etc--eventually added to existing programming languages. Although this allows the highest degree of flexibility (any form of parallel control can be implemented in terms of the basic low level primitives gif), it leaves the additional layer of complexity completely on the shoulders of the programmer, making his task extremely complicate.

Implicit Parallelism

- Allows programmers to write their programs without any concern about the exploitation of parallelism. Exploitation of parallelism is instead automatically performed by the compiler and/or the runtime system. In this way the parallelism is transparent to the programmer maintaining the complexity of software development at the same level of standard sequential programming.
- Extracting parallelism implicitly is not an easy task. For imperative programming languages, the complexity of the problem is almost prohibitively and allows positive results only for restricted sets of applications (e.g., applications which perform intensive operations on arrays.
- Declarative Programming languages, and in particular Functional and Logic languages, are characterized by a very high level of abstraction, allowing the programmer to focus on what the problem is and leaving implicit many details of how the problem should be solved.
- Declarative languages have opened new doors to automatic exploitation of parallelism. Their focusing on a high level description of the problem and their mathematical nature turned into positive properties for implicit exploitation of parallelism.

Methods for parallelism

There are many methods of programming parallel computers. Two of the most common are message passing and data parallel.

1. Message Passing - the user makes calls to libraries to explicitly share information between processors.
2. Data Parallel - data partitioning determines parallelism
3. Shared Memory - multiple processes sharing common memory space
4. Remote Memory Operation - set of processes in which a process can access the memory of another process without its participation
5. Threads - a single process having multiple (concurrent) execution paths
6. Combined Models - composed of two or more of the above.

Methods for parallelism

Message Passing:

- Each Processor has direct access only to its local memory
- Processors are connected via high-speed interconnect
- Data exchange is done via explicit processor-to-processor communication i.e processes communicate by sending and receiving messages : send/receive messages
- Data transfer requires cooperative operations to be performed by each process (a send operation must have matching receive)

Data Parallel:

- Each process works on a different part of the same data structure
- Processors have direct access to global memory and I/O through bus or fast switching network
- Each processor also has its own memory (cache)
- Data structures are shared in global address space
- Concurrent access to shared memory must be coordinate
- All message passing is done invisibly to the programmer

11)What are the various issues in concurrent programming? Explain producer consumer problem

with python program.

Issues Concurrent Programming Paradigm

Concurrent programming is programming with multiple tasks. The major issues of concurrent programming are:

- Sharing computational resources between the tasks;
- Interaction of the tasks.

Objects shared by multiple tasks have to be safe for concurrent access. Such objects are called protected. Tasks accessing such an object interact with each other indirectly through the object.

An access to the protected object can be:

- Lock-free, when the task accessing the object is not blocked for a considerable time;
- Blocking, otherwise.

Blocking objects can be used for task synchronization. To the examples of such objects belong:

- Events;
- Mutexes and semaphores;
- Waitable timers;
- Queues

Producer and Consumer problem using thread

```
import threading,time,Queue
items = Queue.Queue()
# A producer thread
def producer():
    print "I'm the producer"
    for i in range(30):
        items.put(i)
        time.sleep(1)
# A consumer thread
def consumer():
    print "I'm a consumer", threading.currentThread().name
    while True:
        x = items.get()
        print threading.currentThread().name,"got", x
        time.sleep(5)
# Launch a bunch of consumers
cons = [threading.Thread(target=consumer)
        for i in range(10)]
for c in cons:
    c.setDaemon(True)
    c.start()
# Run the producer
producer()
```

12)Explain with suitable example
a. How to customize turtle
b. Drawing geometric shapes

Import Turtle
t=turtle.Turtle()

1)To change background colour of the screen- turtle.bgcolour(“blue”)
2)to change the title of the screen-turtle.title(“my turtle screen”)
3)changing turtle size-t.shapesize(1,5,6)
4)to change pensize- t.pensize(5)

5)to change turtle colour- t.fillcolor(“red”)
6)to change pen color or outline of the colour-t.pencolor(“blue”)
7)to change turtle shape into turtle itself - t.shape(“turtle”)
Arrow shape-t.shape(“arrow”)
Circle shape=t.shape(“circle”)
8)to change pen speed = t.speed(5)

To draw geometric shapes
1)square

```
Import Turtle
t=turtle.Turtle()
t.forward(100)
t.right(90)
t.forward(100)
t.right(90)
t.forward(100)
t.right(90)
t.forward(100)
t.right(90)
```

2)circle

```
Import Turtle
t=turtle.Turtle()
t.circle(60)
```

3)rectangle

```
Import Turtle
t=turtle.Turtle()
t.fd(100)
t.rt(90)
t.fd(200)
t.rt(90)
t.fd(100)
t.rt(90)
t.fd(200)
t.rt(90)
```

4)triangle

```
Import Turtle
t=turtle.Turtle()
import turtle
t= turtle.Turtle()
t.forward(100)
t.left(120)
t.forward(100)
t.left(120)
```

t.forward(100)

13)Explain in detail
a. Multiprocessing
b. Multithreading

Multiprocessing refers to the ability of a system to support more than one processor at the same time. Applications in a multiprocessing system are broken to smaller routines that run independently. The operating system allocates these threads to the processors improving performance of the system.

```
import multiprocessing
def worker(num):
    print('Worker:', num)
    for i in range(num):
        print(i)
    return

jobs = []
for i in range(1,5):
    p = multiprocessing.Process(target=worker, args=(i+10,))
    jobs.append(p)
    p.start()
```

- A thread is basically an independent flow of execution. A single process can consist of multiple threads. Each thread in a program performs a particular task. For Example, when you are playing a game say FIFA on your PC, the game as a whole is a single process, but it consists of several threads responsible for playing the music, taking input from the user, running the opponent synchronously, etc.
- Threading is that it allows a user to run different parts of the program in a concurrent manner and make the design of the program simpler.
- Multithreading in Python can be achieved by importing the threading module.

Example:

```
import threading
from threading import *
```

Example:

```
import threading
from threading import *
```

Parallel program using Threads in Python

```
# simplest way to use a Thread is to instantiate it with a target function
and call start() to let it begin working.
from threading import Thread,current_thread
print(current_thread().getName())
def mt():
   print("Child Thread")
   for i in range(11,20):
        print(i*2)
def disp():
  for i in range(10):
        print(i*2)
child=Thread(target=mt)
child.start()
disp()
print("Executing thread name :",current_thread().getName())
```

```
from threading import Thread,current_thread
class mythread(Thread):
   def run(self):
      for x in range(7):
         print("Hi from child")
a = mythread()
a.start()
a.join()
print("Bye from",current_thread().getName())
```

14)Explain with suitable example about Locks and Rlocks

Locks:

Locks are perhaps the simplest synchronization primitives in Python. A Lock has only two states—locked and unlocked (surprise). is created in the unlocked state and has two principal methods—acquire() and release(). The acquire() method locks the Lock and bloc execution until the release() method in some other co-routine sets it to unlocked.

R-Locks:

R-Lock class is a version of simple locking that only blocks if the lock is held by another thread. While simple locks will block if same thread attempts to acquire the same lock twice, a re-entrant lock only blocks if another thread currently holds the lock.

Synchronization in Python using Lock

```
import threading
x = 0    # A shared value
COUNT = 100
lock = threading.Lock()

def incr():
    global x
    lock.acquire()
    print("thread locked for increment cur x=",x)
    for i in range(COUNT):
        x += 1
        print(x)
    lock.release()
    print("thread release from increment cur x=",x)

def decr():
    global x
    lock.acquire()
    print("thread locked for decrement cur x=",x)
    for i in range(COUNT):
        x -= 1
        print(x)
    lock.release()
    print("thread release from decrement cur x=",x)
t1 = threading.Thread(target=incr)
t2 = threading.Thread(target=decr)
t1.start()
t2.start()
t1.join()
t2.join()
```

Synchronization in Python using RLock

```
import threading

class Foo(object):
    lock = threading.RLock()
    def __init__(self):
        self.x = 0
    def add(self,n):
        with Foo.lock:
            self.x += n
    def incr(self):
        with Foo.lock:
            self.add(1)
    def decr(self):
        with Foo.lock:
            self.add(-1)


def adder(f,count):
    while count > 0:
        f.incr()
        count -= 1

def subber(f,count):
    while count > 0:
        f.decr()
        count -= 1

# Create some threads and make sure it works
COUNT = 10
f = Foo()
t1 = threading.Thread(target=adder,args=(f,COUNT))
t2 = threading.Thread(target=subber,args=(f,COUNT))
t1.start()
t2.start()
t1.join()
t2.join()
print(f.x)
```

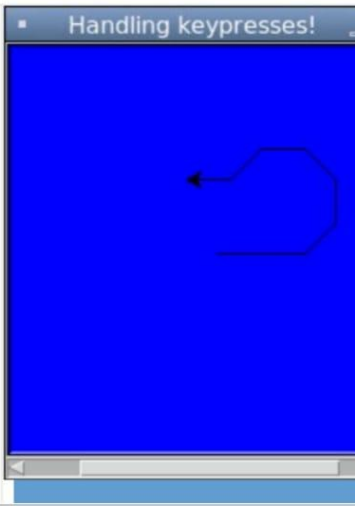
15)Explain with suitable example about Keypress events and mouse events

Keypress events, Mouse events

The turtle module allows us to detect when the user has hit certain keys on the keyboard or moved/clicked the mouse. Whenever the user performs an action as such it is called an event. We can listen for events and trigger functions to run if we "hear" the event

Keypress events

```
1  import turtle
2  turtle.setup(400,500)               # Determine the window size
3  wn = turtle.Screen()                # Get a reference to the window
4  wn.title("Handling keypresses!")    # Change the window title
5  wn.bgcolor("blue")              # Set the background color
6  tess = turtle.Turtle()              # Create our favorite turtle
7
8  #  "event handlers".
9  def h1():
10   tess.forward(30)
11 def h2():
12   tess.left(45)
13 def h3():
14   tess.right(45)
15 def h4():
16   wn.bye()
17
18 wn.onkey(h1, "Up")
19 wn.onkey(h2, "Left")
20 wn.onkey(h3, "Right")
21 wn.onkey(h4, "q")
22
23 wn.listen()
24 wn.mainloop()
```



Points to Remember

Function	Description
Wn.listen()	Window's listen method is called , otherwise it won't notice our keypresses
h1(), h2(), h3(), h4()	Handler functions. The handlers can be arbitrarily complex functions that call other functions, etc. q key closes the turtle window
Some of the symbolic names to try are Cancel (the Break key), BackSpace, Tab, Return(the Enter key), Shift_L (any Shift key), Control_L (any Control key), Alt_L (any Alt key), Pause, Caps_Lock, Escape, Prior (Page Up), Next (Page Down), End, Home, Left, Up, Right, Down, Print, Insert, Delete, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, Num_Lock, and Scroll_Lock.	

Mouse events

```
1   import turtle
2
3   turtle.setup(300,300)
4   wn = turtle.Screen()
5   wn.title("How to handle mouse clicks on the window!")
6   wn.bgcolor("Yellow")
7
8   tess = turtle.Turtle()
9   tess.color("brown")
10  tess.pensize(3)
11  tess.shape("circle")
12
13  def h1(x, y):
14    tess.goto(x, y)
15
16  wn.onclick(h1)  # Wire up a click on the window.
17  wn.mainloop()
```