UNIT 3 QUESTION BANK ANSWERS

1) The function multiplies y by x and decreases x by 1 in each recursive call.
The calls proceed as: fun(4,2) → fun(3,8) → fun(2,24) → fun(1,48) → fun(0,48).
When x becomes 0, the function returns y.
Output: 48

2)

```
def average(lst, n):
    if n == 0:
        return 0
    return (lst[n-1] + average(lst, n-1))

lst = [10, 20, 30, 40]
avg = average(lst, len(lst)) / len(lst)
print(avg)
```

3)

```
def sanitize(lst, index=0):
    if index == len(lst):
        return lst
    if lst[index] < 0:
        lst[index] = 0
    return sanitize(lst, index + 1)

lst = [5, -3, 7, -1, 9]
print(sanitize(lst))
```

4) This function uses nested recursion. When the number becomes greater than 100, it returns num - 10.
For fun(75), the recursive calls eventually reach fun(101), which returns 91.
The recursion then resolves back to the initial call.
Output: 91

5)

```python
def isPowerOfTwo(n):
    if n == 1:
        return True
    if n <= 0 or n % 2 != 0:
        return False
    return isPowerOfTwo(n // 2)

num = 16
print(isPowerOfTwo(num))
```

6)

```python
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))
```

7)

```python
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(6))
```

8)

```
def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a % b)

print(gcd(48, 18))
```

9)

```
def reverse_string(s):
    if s == "":
        return s
    return reverse_string(s[1:]) + s[0]

print(reverse_string("python"))
```

10)

```
def sum_of_digits(n):
    if n == 0:
        return 0
    return (n % 10) + sum_of_digits(n // 10)

print(sum_of_digits(1234))
```

11)

```
def power(x, n):
    if n == 0:
        return 1
    return x * power(x, n - 1)

print(power(2, 5))
```

12) a. **True** – Each recursive call uses its own copy of variables.
b. **True** – Each recursive call creates a new stack frame in memory.
c. **True** – A recursive function must return a value to terminate properly.
d. **True** – Any loop can be rewritten using recursion.
e. **True** – Missing or incorrect base case causes infinite recursion.
f. **False** – Recursive functions are often harder to understand and maintain than loops.

13)

```
def unique_words(line):
    for word in line.split():
        yield word

line = input("Enter a line: ")
print(set(unique_words(line)))
```

14)

```
def students(data):
    for student in data:
        yield student

records = [("Asha", 85), ("Ravi", 92), ("Meena", 88)]
print(max(students(records), key=lambda x: x[1]))
```

15)

```python
def is_palindrome(s):
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    return is_palindrome(s[1:-1])

print(is_palindrome("madam"))
```

16)

```python
def reverse_chars(s):
    for i in range(len(s)-1, -1, -1):
        yield s[i]

for ch in reverse_chars("python"):
    print(ch, end="")
```

17) The first statement uses a **list comprehension**, which creates a full list in memory before summing.
The second statement uses a **generator expression**, which generates values one at a time.
Generator expressions are **more memory efficient**.
Both produce the same result, but differ in memory usage.

18)

```
def concatenate_lists(list1):
    def inner(list2):
        return [a + b for a, b in zip(list1, list2)]
    return inner

l1 = ["Hi ", "Good ", "Nice ", "Happy ", "Best "]
l2 = ["Day", "Morning", "Try", "Life", "Luck"]

result = concatenate_lists(l1)
print(result(l2))
```

19) A closure is a function that remembers values from its enclosing scope even after the outer function has finished execution.

```
def outer(x):
    def inner():
        return x + 5
    return inner

f = outer(10)
print(f())
```

20) A regular function does not remember variables after execution.
A closure **retains access to free variables** from its enclosing scope.
Closures enable **data hiding** and persistent state.

21) A free variable is a variable used inside a function but not defined there.
It is defined in the enclosing scope and remembered by the closure.
Free variables allow closures to retain state.

22) Closures restrict direct access to variables, allowing controlled modification.

```
def counter():
    count = 0
    def increment():
        nonlocal count
        count += 1
        return count
    return increment

c = counter()
print(c())
print(c())
```

23) Closures are useful in function factories, callbacks, and decorators.
They allow functions to remember configuration settings.
Example: maintaining a running count without using global variables.

24) Lexical scoping means variables are resolved based on their position in the source code.
Inner functions can access variables of outer functions.
Closures rely on lexical scoping to retain values.

25)

```
def multiply(x):
    def inner(y, z):
        return x * y * z
    return inner

f = multiply(2)
print(f(3, 4))
```

26) In a regular function, variables exist only during function execution.
In a closure, variables persist as long as the inner function exists.
This allows state retention across function calls.

27) a. **True** – Decorators add functionality to existing functions.
b. **False** – A decorator can be applied to multiple functions.
c. **False** – Decorated functions can accept arguments.
d. **False** – Decorated functions can return values.

28) A callback is a function passed as an argument to another function and executed later.

```
def greet(name):
    print("Hello", name)

def process(func):
    func("Python")

process(greet)
```

29) In GUI programs, a callback executes when an event occurs.
def on_click(event):
print("Button clicked")
The function is automatically called when the button is clicked.

30) A decorator modifies the behavior of a function without changing its code.

```
def my_decorator(func):
    def wrapper():
        print("Before function")
        func()
        print("After function")
    return wrapper

@my_decorator
def say_hi():
    print("Hi")

say_hi()
```

31) Nested decorators apply multiple decorators to a single function.

```
def d1(func):
    def w():
        print("D1")
        func()
    return w

def d2(func):
    def w():
        print("D2")
        func()
    return w

@d1
@d2
def hello():
    print("Hello")

hello()
```

32) Generators produce values one at a time using yield.

```
def numbers():
    for i in range(3):
        yield i

for n in numbers():
    print(n)
```

33) Generators use less memory.
They produce values lazily.
Suitable for large datasets and infinite sequences.

34) squares = (x*x for x in range(5))

print(list(squares))

35) A generator uses yield and returns an iterator.
A regular function uses return and exits completely.
Generators maintain state between calls.

36) next() retrieves the next value from a generator.
It raises StopIteration when values are exhausted.

37)

```
def fib(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a+b

for x in fib(5):
    print(x)
```

38)

```
import wx

class SimpleCalc(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Add & Subtract", size=(250, 200))
        panel = wx.Panel(self)

        self.num1 = wx.TextCtrl(panel, pos=(20, 20))
        self.num2 = wx.TextCtrl(panel, pos=(20, 60))
        self.result = wx.StaticText(panel, label="Result:", pos=(20, 140))

        addBtn = wx.Button(panel, label="Add", pos=(150, 20))
        subBtn = wx.Button(panel, label="Subtract", pos=(150, 60))

        addBtn.Bind(wx.EVT_BUTTON, self.add)
        subBtn.Bind(wx.EVT_BUTTON, self.sub)

        self.Show()

    def add(self, event):
        a = int(self.num1.GetValue())
        b = int(self.num2.GetValue())
        self.result.SetLabel("Result: " + str(a + b))

    def sub(self, event):
        a = int(self.num1.GetValue())
        b = int(self.num2.GetValue())
        self.result.SetLabel("Result: " + str(a - b))

app = wx.App()
SimpleCalc()
app.MainLoop()
```

39) A closure is a function that remembers variables from its enclosing scope.

Criteria for closure:

i. Must have a nested function
ii. Inner function must use a variable from outer function
iii. Outer function must return the inner function

40) Python supports multiple ways to import modules:

a. import module
b. from module import function
c. from module import *
d. import module as alias

Each method controls namespace access differently.

41)

```python
def divisible_by_5(n):
    for i in range(0, n+1):
        if i % 5 == 0:
            yield i

for num in divisible_by_5(30):
    print(num)
```

42)

```python
def c_to_f(c):
    return (c * 9/5) + 32

def f_to_c(f):
    return (f - 32) * 5/9
```

43) import module

a. from module import function
b. from module import *
c. import module as alias
d. Each method affects how identifiers are accessed.

44) When `a.py` runs, it prints its own statements first.
Then `abc.py` is imported, so its print statements execute.
Output:

```
this is with in a.py
this is with in abc.py
abc abc
a __main__
```

45)

a. True
b. False
c. True
d. True
e. True
f. False
g. True

h. False
i. False
j. True
k. True
l. True
m. True

46) (a) import driving_conversions – access via module name
(b) from driving_conversions import milesPerHr
(c) from driving_conversions import *
(d) from driving_conversions import milesPerHr as mph
(e) Each import creates a different namespace scope.

47)

a. `__doc__`
b. `__init__.py`
c. Private members

48) When a module is imported, Python executes the file and creates a module object in memory.
49) sys.path is modified at runtime, while PYTHONPATH is an environment variable set before execution.
50) It indicates the file is being executed directly, not imported.
51) They indicate internal or private use within a module.
52) It marks a directory as a Python package.
53) Use `from package import module` or aliasing.
54) All directories intended to be treated as Python packages.
55) When importing entire packages or avoiding namespace conflicts.
56) Python supports debugging using the `pdb` module and IDE debuggers.
Yes, stepping through code is possible.
57) Commands: `n`, `s`, `c`, `q`, `l`, `b`
Debug command: `python -m pdb filename.py`
58) A package is a collection of related Python modules stored in a directory.
59) from datetime import datetime

print(datetime.now())

60) __init__.py

61)

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        super().__init__(None, title="Color Change", size=(300,200))
        panel = wx.Panel(self)

        b1 = wx.Button(panel, label="Red")
        b2 = wx.Button(panel, label="Green")
        b3 = wx.Button(panel, label="Blue")

        b1.Bind(wx.EVT_BUTTON, lambda e: panel.SetBackgroundColour("RED"))
        b2.Bind(wx.EVT_BUTTON, lambda e: panel.SetBackgroundColour("GREEN"))
        b3.Bind(wx.EVT_BUTTON, lambda e: panel.SetBackgroundColour("BLUE"))

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(b1)
        sizer.Add(b2)
        sizer.Add(b3)

        panel.SetSizer(sizer)
        self.Show()

app = wx.App()
MyFrame()
app.MainLoop()
```

62)

Steps to create widgets in wxPython:

i. Create `wx.App`
ii. Create `wx.Frame`
iii. Create `wx.Panel`
iv. Add widgets
v. Bind events

```python
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        super().__init__(None, title="Label Display", size=(300,200))
        panel = wx.Panel(self)

        btn = wx.Button(panel, label="Click Me")
        self.label = wx.StaticText(panel, label="")

        btn.Bind(wx.EVT_BUTTON, self.show_text)

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(btn)
        sizer.Add(self.label)

        panel.SetSizer(sizer)
        self.Show()

    def show_text(self, event):
        self.label.SetLabel("Button Clicked")

app = wx.App()
MyFrame()
app.MainLoop()
```

63)

```python
import wx

class Calc(wx.Frame):
    def __init__(self):
        super().__init__(None, title="Calculator", size=(300,200))
        panel = wx.Panel(self)

        self.t1 = wx.TextCtrl(panel)
        self.t2 = wx.TextCtrl(panel)
        self.result = wx.StaticText(panel, label="Result")

        btn = wx.Button(panel, label="Add")
        btn.Bind(wx.EVT_BUTTON, self.add)

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.t1)
        sizer.Add(self.t2)
        sizer.Add(btn)
        sizer.Add(self.result)

        panel.SetSizer(sizer)
        self.Show()

    def add(self, event):
        res = int(self.t1.GetValue()) + int(self.t2.GetValue())
        self.result.SetLabel(str(res))

app = wx.App()
Calc()
app.MainLoop()
```

64) Sizers control layout automatically in wxPython.
BoxSizer arranges widgets vertically or horizontally.
GridSizer arranges widgets in rows and columns.
Sizers make GUIs responsive and platform-independent.

65)

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        super().__init__(None, title="Message Box", size=(200,300))
        panel = wx.Panel(self)

        btn = wx.Button(panel, label="Click Here", pos=(50,70))
        btn.Bind(wx.EVT_BUTTON, self.show_msg)

        self.Show()

    def show_msg(self, event):
        wx.MessageBox("Button clicked", "Info")

app = wx.App()
MyFrame()
app.MainLoop()
```

66) MenuBar is displayed at the top of a window.
Menu contains menu items such as File, Edit, Exit.
Menus allow user interaction through commands.

67) `Button` triggers an action when clicked.
`RadioButton` allows selection of only one option from a group.
Both support event handling using `Bind()`.

68) Sizers adjust widgets automatically based on window size.
Absolute positioning places widgets at fixed coordinates.
Sizers are preferred for flexible and responsive layouts.

69)

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        super().__init__(None, title="Show/Hide", size=(300,200))
        panel = wx.Panel(self)

        self.btn2 = wx.Button(panel, label="Button 2")
        btn1 = wx.Button(panel, label="Toggle")

        btn1.Bind(wx.EVT_BUTTON, self.toggle)

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(btn1)
        sizer.Add(self.btn2)

        panel.SetSizer(sizer)
        self.Show()

    def toggle(self, event):
        self.btn2.Show(not self.btn2.IsShown())
        self.Layout()

app = wx.App()
MyFrame()
app.MainLoop()
```

70)

```
import wx

class MyFrame(wx.Frame):
    def __init__(self):
        super().__init__(None, title="Event Handling", size=(300,200))
        panel = wx.Panel(self)

        btn = wx.Button(panel, label="Click Me")
        btn.Bind(wx.EVT_BUTTON, self.on_click)

        self.Show()

    def on_click(self, event):
        print("Button clicked")

app = wx.App()
MyFrame()
app.MainLoop()
```