## **PRACTICE SHEET SOLUTION**

Q1) b) Passing unlimited positional arguments

Explanation: \*args in Python allows a function to accept **any number of positional arguments** (stored as a tuple).

Q2) 8

**Explanation:**

1. **Function Logic:**
   * **Base Case:** If n == 0, return 1.
   * **Even**n**:** Multiply n with twisted\_fact(n-1).
   * **Odd**n**:** Skip multiplication and return twisted\_fact(n-1) directly.
2. **Evaluation for**twisted\_fact(5)**:**
   * **5 (Odd):** Returns twisted\_fact(4)  
     → No multiplication, recursion continues.
   * **4 (Even):** Returns 4 \* twisted\_fact(3)  
     → Multiplies by 4, then recurses.
   * **3 (Odd):** Returns twisted\_fact(2)  
     → Skips multiplication, continues.
   * **2 (Even):** Returns 2 \* twisted\_fact(1)  
     → Multiplies by 2, then recurses.
   * **1 (Odd):** Returns twisted\_fact(0)  
     → Skips multiplication, continues.
   * **0 (Base Case):** Returns 1.
3. **Unwinding the Recursion:**
   * twisted\_fact(1) → 1
   * twisted\_fact(2) → 2 \* 1 = 2
   * twisted\_fact(3) → 2 (skipped multiplication)
   * twisted\_fact(4) → 4 \* 2 = 8
   * twisted\_fact(5) → 8 (skipped multiplication)
4. **Final Output:**  
   The function effectively **multiplies only even numbers** from n down to 2.  
   For n=5:  
   4×2 = 8

Q3)

Inner: 6

Outer: 10

Global: 6

**Explanation:**

1. **Global**x**(Line 1):**
   * x = 5 is a **global variable**.
2. outer()**Function (Line 2-9):**
   * **Local**x = 10 is defined inside outer() (shadows the global x).
   * Calls inner().
3. inner()**Function (Line 4-7):**
   * global x declares that x refers to the **global**x**(not**outer**’s**x**)**.
   * x += 1 modifies the **global**x**(now**5 → 6**)**.
   * Prints "Inner: 6".
4. **Back to**outer()**:**
   * Prints "Outer: 10" (local x in outer() is unchanged).
5. **After**outer()**:**
   * Prints "Global: 6" (global x was modified by inner()).

Q4) The output is **6**, which is the number of unique paths from (0,0) to (2,2) using only right/down moves, computed recursively as combinations.

Q5) Output: 91

This is the **McCarthy 91 function**, a recursive function designed to always return 91 for any integer input n ≤ 100.

**Step-by-Step for**nested(95)**:**

1. **First Call:** nested(95) → 95 ≤ 100 → Calls nested(nested(95 + 11)) = nested(nested(106)).
2. **Second Call:** nested(106) → 106 > 100 → Returns 106 - 10 = 96.
3. **Third Call:** nested(96) → 96 ≤ 100 → Calls nested(nested(96 + 11)) = nested(nested(107)).
4. **Fourth Call:** nested(107) → 107 > 100 → Returns 107 - 10 = 97.
5. **Fifth Call:** nested(97) → 97 ≤ 100 → Calls nested(nested(108)) → Returns 98.
   * This pattern continues until nested(101) → Returns 91.
6. **Final Unwinding:** All nested calls resolve to 91.

Q6)

def getCostOfCoffee(numberOfCoffees, pricePerCoffee):

# Calculate the number of free coffees we get in this order:

numberOfFreeCoffees = numberOfCoffees // 9

# Calculate the number of coffees we will have to pay for in this order: numberOfPaidCoffees = numberOfCoffees - numberOfFreeCoffees

# Calculate and return the price:

return numberOfPaidCoffees \* pricePerCoffee

Q7) Output :

Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C

**Total Moves Made:** **7** (for n = 3 disks).

The minimal number of moves needed is **always** 2n−1. So for:

* **3 disks:** 23−1=7 moves.
* **64 disks:** 264−1 moves

Q8) Output: Inner x: 10  
Modified:

x = 5

def outer():

x = 10

def inner():

print("Inner x:", globals()['x'])

inner()

Q9)

def sort\_digits(n):

# Base case: single digit number is already sorted

if n < 10:

return n

# Convert number to string to process digits

s = str(n)

# Find the smallest digit

min\_digit = min(s)

# Remove first occurrence of smallest digit

new\_s = s.replace(min\_digit, '', 1)

# Recurse with remaining digits and build result

return int(min\_digit + str(sort\_digits(int(new\_s) if new\_s else 0)))

print(sort\_digits(4312)) # Output: 1234

print(sort\_digits(989)) # Output: 899

print(sort\_digits(0)) # Output: 0

Q10)

def isLeapYear(year):

# Years divisible by 400 are leap years:

if year % 400 == 0:

return True

# Otherwise, years divisible by 100 are not leap years:

elif year % 100 == 0:

return False

# Otherwise, years divisible by 4 are leap years:

elif year % 4 == 0:

return True

# Otherwise, every other year is not a leap year:

else:

return False

def isValidDate(year, month, day):

# If month is outside the bounds of 1 to 12, return False:

if not (1 <= month <= 12):

return False

# If the year is a leap year and the date is Feb 29th, it is valid:

if leapyear.isLeapYear(year) and month == 2 and day == 29:

return True

# Check for invalid dates in 31-day months:

if month in (1, 3, 5, 7, 8, 10, 12) and not (1 <= day <= 31):

return False

# Check for invalid dates in 30-day months:

elif month in (4, 6, 9, 11) and not (1 <= day <= 30):

return False

# Check for invalid dates in February:

elif month == 2 and not (1 <= day <= 28):

return False

# Date passes all checks and is valid, so return True:

return True

Q11)

def supply\_water(water, buildings=0):

if water < 20:

return buildings

return supply\_water(water \* 0.9, buildings + 1)

def city\_report(city\_name, \*args, required=20):

sector = 1

for water in args:

b = supply\_water(water)

print(f"Sector {sector} of {city\_name}: {b} buildings received water")

sector += 1

Q12)

import random

import string

def generate\_captcha(length=5):

characters = string.ascii\_letters + string.digits

captcha = ''.join(random.choice(characters) for \_ in range(length))

print("CAPTCHA:", captcha)

generate\_captcha()

Q13)

def custom\_bool(x):

if x:

return True

else:

return False

print(custom\_bool(""))

print(custom\_bool("0"))

print(custom\_bool(0))

Q14)

def climbStairs(n):

if n == 1:

return 1

if n == 2:

return 2

return climbStairs(n - 1) + climbStairs(n - 2)

print(climbStairs(5)) # Output: 8 (1+1+1+1+1, 2+1+1+1, 1+2+1+1, etc.)

or

def climbStairs2(n):

if n == 0:

return 0

if n == 1:

return 1

a, b = 1, 2

for \_ in range(2, n):

a, b = b, a + b

return b

print(climbStairs2(5)) # Output: 8

Q15)

The function returns True (can win) if n is **not divisible by 4**. This is because you can always force a win by leaving multiples of 4 stones for your opponent.

def can\_win\_nim(n):

return n % 4 != 0

stones = int(input("Enter number of stones: "))

if can\_win\_nim(stones):

print("You can win!")

else:

print("You cannot win.")