1. (a) Evaluate the following expression: 5.

   - 5
   - $5 + 10 - 200//5$
   - $5 + 10 - 40$
   - $15 - 40$
   - -25

   (b) What will be the output produced on the execution of the following code snippet? *python for index in range(30,12,-5): print(index, end = '')*

   - 30252015

   (c) In the given code snippet, state the value of M after execution of each statement:

   - The provided code snippet is incomplete and contains a syntax error (*M = M.append(80)*). Assuming M is a list and the intention is to modify M:

     Let's assume *M = [10, 20, 30]*.

     - *M.append(80)*: The *append()* method modifies the list in-place and returns *None*. So *M = M.append(80)* would assign *None* to *M*, which would lead to an error in subsequent operations. Assuming the intention was just *M.append(80)*: *M* will be *[10, 20, 30, 80]*

- *print(M)*: *[10, 20, 30, 80]*
- *M.remove(20)*: *M* will be *[10, 30, 80]*
- *print(M)*: *[10, 30, 80]*
- *M.extend(['fragrance'])*: *M* will be *[10, 30, 80, 'fragrance']*
- *print(M)*: *[10, 30, 80, 'fragrance']*

(d) Differentiate between Encapsulation and Abstraction with suitable examples.

- **Encapsulation:**
  - **Definition:** Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit or class. It also involves restricting direct access to some of an object's components, which means direct access to the attributes is prevented, and they can only be accessed through the methods of the class. This is often achieved using access specifiers like private or protected.
  - **Purpose:** To protect data from unauthorized access and ensure data integrity. It provides a way to control how data is accessed and modified.

- **Example:** Consider a *Car* class. The internal working of the engine, braking system, etc., are encapsulated within the *Car* object. You interact with the car using methods like *start()*, *accelerate()*, *brake()*, without needing to know the complex details of how these actions are performed internally. The engine's RPM or fuel consumption data might be private attributes, accessible only via public methods that provide specific information.

- **Abstraction:**
  - **Definition:** Abstraction is the process of hiding the complex implementation details and showing only the essential features of an object or system. It focuses on "what" an object does rather than "how" it does it.
  - **Purpose:** To simplify complex systems by modeling classes based on their essential properties and behaviors. It reduces complexity and improves efficiency by allowing developers to focus on higher-level design.
  - **Example:** When you drive a car, you use the steering wheel, accelerator, and brakes. You don't need to know the intricate details of how the engine works, how the braking fluid pressure is

applied, or how the power steering mechanism functions. The complex mechanics are "abstracted away," and you only interact with the essential controls. Similarly, a *TV* remote control abstracts the complex circuitry of the television; you only see buttons for essential functions like power, volume, and channel change.

(e) Consider the popularity of different programming languages stored in form of list as shown below : *python languages = ["Python", "JavaScript", "Java", "C#", "C++", "Ruby"] popularity = [29.9, 19.1, 15.2, 10.3, 8.7, 5.1]* Give appropriate title to the graph and add xticks separated by a distance of 4 units and yticks separated by a distance of 3 units.

- **Title:** "Popularity of Programming Languages"
- **X-ticks:** These would represent the *languages*. Since they are categorical, setting a distance of 4 units directly on categories isn't standard in a simple plot. Typically, if this were a numerical axis, you'd set numerical intervals. For categorical data, you might control the spacing between labels. If this were a bar chart, the bars would be at integer positions, and you could set the tick locations based on those positions, perhaps showing every 4th label if there were many. For a typical bar plot, all language names would be shown as x-ticks.

- **Y-ticks:** These would represent the *popularity* percentages.
  - Possible y-ticks at a distance of 3 units: 0,3,6,9,12,15,18,21,24,27,30.

(f) Differentiate between the keywords break and continue in Python with suitable examples.

- ***break* keyword:**
  - **Purpose:** The *break* keyword is used to terminate the execution of the innermost *for* or *while* loop immediately.
  - **Effect:** When *break* is encountered, the control flow exits the loop entirely, and execution resumes at the statement immediately following the loop.
  - **Example:**

```
for i in range(1, 10):
    if i == 5:
        break  # Loop will terminate when i becomes
5
    print(i)
# Output:
# 1
# 2
# 3
# 4
```

- ***continue* keyword:**
    - **Purpose:** The *continue* keyword is used to skip the rest of the current iteration of the loop and proceed to the next iteration.
    - **Effect:** When *continue* is encountered, the control flow jumps to the beginning of the loop for the next iteration, skipping any code below *continue* in the current iteration.
    - **Example:**

```
for i in range(1, 10):
    if i % 2 == 0:
        continue  # Skips even numbers
    print(i)
# Output:
# 1
# 3
# 5
# 7
# 9
```

2. (a) Write a program that reads a number and check whether it is positive, negative or zero.

-
  ```
  # Read a number from the user
  num = float(input("Enter a number: "))

  # Check if the number is positive, negative, or zero
  if num > 0:
      print("The number is positive.")
  elif num < 0:
      print("The number is negative.")
  else:
      print("The number is zero.")
  ```

(b) What will be the output produced on the execution of the following code snippet? *python num = 10 while num > 0: print(num) num = 2 if num == 3: break else: print("Done")*

-
  ```
  10
  Done
  ```

  Justification:

  - Initially, *num* is 10.
  - The *while* loop starts. *num > 0* (10 > 0) is true.
  - *print(num)* prints *10*.
  - *num* is then set to *2*.
  - The *if num == 3:* condition (2 == 3) is false.

- The *else* block executes, *print("Done")* prints *Done*.
- The loop condition *num > 0* (2 > 0) is true.
- *print(num)* prints *2*.
- *num* is then set to *2*.
- The *if num == 3:* condition (2 == 3) is false.
- The *else* block executes, *print("Done")* prints *Done*.
- This creates an infinite loop where 2 and Done are printed repeatedly. The provided snippet would continuously output:

  10
  Done
  2
  Done
  2
  Done
  ... (and so on indefinitely)

  However, typically in such questions, the expectation is for a finite output unless specified. Given the repetition, the first few iterations are what would be observed before a halt (e.g., due to resource limits or manual termination). If the question implies a single execution trace, it would be the first *10* and *Done*. If it's about what *will* be produced, it's an infinite loop of *2* and *Done* after the initial *10*.

(c) Identify the output/error (if any) for the following. Justify your answer if it is an error.

- (i) *set_of_code = {'Alpha', 'Beta', 'Gamma'} print(set_of_code)*
    - **Output:** *{'Gamma', 'Alpha', 'Beta'}* (Order may vary as sets are unordered)
    - **Justification:** This creates a set, and printing a set displays its elements. The order of elements in a set is not guaranteed.
- (ii) *tup1=(8) print(tup1.index(8))*
    - **Output:** *0*
    - **Justification:** *tup1=(8)* defines an integer *8*, not a tuple. To define a single-element tuple, a comma is required: *tup1=(8,)*. However, Python treats *(8)* as just the integer *8*. Integers do not have an *index()* method. Therefore, this will result in an *AttributeError*. **Error:** *AttributeError: 'int' object has no attribute 'index'*

- (iii) *list1 = [10, 7, 3, 'Add', 20] print(sum(list1))*
  - **Error:** *TypeError: unsupported operand type(s) for +: 'int' and 'str'*
  - **Justification:** The *sum()* function in Python is used to sum numeric items in an iterable. *list1* contains a string element *'Add'*. The *sum()* function cannot add an integer to a string, hence it raises a *TypeError*.
- (iv) *str1='Hi, where are you?' str1='w'*
  - **Output:** This snippet itself doesn't produce an output. If a *print(str1)* statement followed, the output would be *w*.
  - **Justification:** This is a simple assignment. The variable *str1* is first assigned the string *'Hi, where are you?'* and then immediately reassigned to the string *'w'*. There is no error.
- (v) *Ascii= {'A':65, 'B':66, 'A':67} Ascii.pop('A') print(Ascii)*
  - **Output:** *{'B': 66}*
  - **Justification:** Dictionaries in Python cannot have duplicate keys. When *Ascii= {'A':65, 'B':66, 'A':67}* is created, the duplicate key *'A'* with value *67* overwrites the previous value *65*. So, *Ascii*

initially becomes *{'A': 67, 'B': 66}*. Then, *Ascii.pop('A')* removes the key-value pair where the key is *'A'*. Finally, *print(Ascii)* prints the remaining key-value pair.

3. (a) Consider the following nested list : *python studMarks = [['Sandhya', 90], ['Sita', 76], ['Shyam', 56]]* Write a program to determine the number of students who have obtained distinction (>=75).

   -
     ```
     studMarks = [['Sandhya', 90], ['Sita', 76], ['Shyam', 56]]
     distinction_count = 0

     for student in studMarks:
         # student[1] contains the mark
         if student[1] >= 75:
             distinction_count += 1

     print("Number of students with distinction:", distinction_count)
     ```

   (b) Consider the following string : *python greeting = 'Good Morning. Have a Good Day!!'* What will be the output produced when executing each of the statements/function calls?

   - (i) *greeting[-len(greeting): len(greeting)]*
     - *Good Morning. Have a Good Day!!*

    - **Justification:** *len(greeting)* is 31. So, -*len(greeting)* is -31. *greeting[-31:31]* effectively slices the entire string from the beginning to the end.
- (ii) *greeting[:-12] + greeting[-12:]*
    - *Good Morning. Have a Good Day!!*
    - **Justification:** *greeting[:-12]* gives *'Good Morning. Have a '* (up to 12 characters from the end excluded). *greeting[-12:]* gives *'Good Day!!'* (the last 12 characters). Concatenating them reconstructs the original string.
- (iii) *greeting.isalpha()*
    - *False*
    - **Justification:** The string contains spaces, periods, and exclamation marks, which are not alphabetic characters. *isalpha()* returns True only if all characters in the string are alphabetic and the string is not empty.
- (iv) *greeting.istitle()*
    - *False*
    - **Justification:** For *istitle()* to return True, every word in the string must start with an uppercase letter, and all other characters in the word must

be lowercase. In "Good Morning. Have a Good Day!!", "a" is lowercase.

- (v) *greeting.replace('Good', 'Sweet')*
    - *Sweet Morning. Have a Sweet Day!!*
    - **Justification:** The *replace()* method replaces all occurrences of the specified substring with another substring.
- (vi) *greeting.endswith('!!!')*
    - *False*
    - **Justification:** The string *greeting* ends with *!!*, not *!!!*.

(c) Consider the following function : *python def myFun(a, b = 1): return a + b* What will be the output produced when the following calls are made?

- (i) *myFun(b = 7)*
    - **Error:** *TypeError: myFun() missing 1 required positional argument: 'a'*
    - **Justification:** When calling *myFun(b = 7)*, a value is provided for the default parameter *b*, but the required positional argument *a* is not provided.
- (ii) *myFun(2)*

- *3*
- **Justification:** The value *2* is assigned to *a*. The parameter *b* uses its default value of *1*. So, *2 + 1 = 3*.

4. (a) Define a function *areaTriangle()* that takes the lengths of three sides: *side1*, *side2*, and *side3* of the triangle as the input parameters and returns the area of the triangle as the output. Also, define a function *main()* that accepts inputs from the user interactively and computes the area of the triangle using the function *areaTriangle()*. Also, assert that sum of the length of any two sides is greater than the third side.

```
import math

def areaTriangle(side1, side2, side3):
    """
    Calculates the area of a triangle given its three side lengths.
    Raises AssertionError if the sides do not form a valid triangle.
    """
    # Assert that the sum of the length of any two sides is greater than the third side
    assert (side1 + side2 > side3) and \
           (side1 + side3 > side2) and \
           (side2 + side3 > side1), \
            "Invalid triangle: Sum of any two sides must
be greater than the third side."

      # Calculate the semi-perimeter
      s = (side1 + side2 + side3) / 2

      # Calculate the area using Heron's formula
      area = math.sqrt(s * (s - side1) * (s - side2) * (s -
side3))
      return area

  def main():
      """
      Accepts inputs from the user for triangle sides and
computes its area.
      """
      print("Enter the lengths of the three sides of the
triangle:")
      try:
          side1 = float(input("Side 1: "))
          side2 = float(input("Side 2: "))
          side3 = float(input("Side 3: "))

          # Ensure sides are positive
          assert side1 > 0 and side2 > 0 and side3 > 0,
"Side lengths must be positive."
```

```
        triangle_area = areaTriangle(side1, side2,
side3)
        print(f"The area of the triangle is:
{triangle_area:.2f}")

    except ValueError:
        print("Invalid input: Please enter numeric
values for side lengths.")
    except AssertionError as e:
        print(f"Error: {e}")

  # Call the main function to run the program
  main()
```

(b) If $a = 10, b = 12, c = 0$. find the values of following expressions :

- (i) *a!=6 and b>5*
  - *True*
  - **Justification:** *a != 6* (10 != 6) is *True*. *b > 5* (12 > 5) is *True*. *True and True* is *True*.
- (ii) *a==9 or b <3*
  - *False*
  - **Justification:** *a == 9* (10 == 9) is *False*. *b < 3* (12 < 3) is *False*. *False or False* is *False*.

- (iii) *not (a<10)*
  - *True*
  - **Justification:** *a < 10* (10 < 10) is *False*. *not False* is *True*.
- (iv) *not(a>5 and c)*
  - *True*
  - **Justification:** *a > 5* (10 > 5) is *True*. *c* (which is *0*) evaluates to *False* in a boolean context. *True and False* is *False*. *not False* is *True*.
- (v) *5 and c!=8 or not c*
  - *True*
  - **Justification:**
    - *5* evaluates to *True*.
    - *c != 8* (0 != 8) is *True*.
    - *5 and c != 8* evaluates to *True and True,* which is *True*.
    - *not c* (*not 0*) evaluates to *True*.
    - *True or True* is *True*.
- (vi) *ab*c**c*
  - **Error:** *NameError: name 'ab' is not defined*

- **Justification:** *ab* is interpreted as a single variable name, which is not defined. If it was intended as *a * b*, then it would be *10 * 12 * 0**0*. In Python, *0**0* is *1*. So *10 * 12 * 1 = 120*. However, as written, it's a *NameError*.

(c) What will be the output produced on the execution of the following code snippet? *python a = 1 def f(): print('Value of 'a' Inside f() : ', a) def g(): a = 2 print('Value of 'a' Inside g() : ', a) def h(): global a a = 3 print('Value of 'a' Inside h() : ', a) print('global : ', a) f() print('Value of 'a' after f() : ', a) g() print('Value of 'a' after g() : ', a) h() print('Value of 'a' after h() : ', a)*

- *global : 1*
  Value of a Inside f() : 1
  Value of a after f() : 1
  Value of a Inside g() : 2
  Value of a after g() : 1
  Value of a Inside h() : 3
  Value of a after h() : 3

- **Justification:**
  - *a = 1*: Global variable *a* is initialized to 1.
  - *print('global : ', a)*: Prints *global : 1*.
  - *f()*:

    - *print('Value of 'a' Inside f() : ', a)*: Accesses the global *a* (since no local *a* is defined in *f*). Prints *Value of a Inside f() : 1*.

- *print('Value of 'a' after f() : ', a)*: Prints the global *a*, which is still *1*.
- *g()*:
    - *a = 2*: Creates a *new local* variable *a* within *g* and assigns it *2*. This does not affect the global *a*.
    - *print('Value of 'a' Inside g() : ', a)*: Prints the *local a*. Prints *Value of a Inside g() : 2*.
- *print('Value of 'a' after g() : ', a)*: Prints the global *a*, which is still *1*.
- *h()*:
    - *global a*: Declares that the *a* inside *h* refers to the global *a*.
    - *a = 3*: Modifies the *global a* to *3*.
    - *print('Value of 'a' Inside h() : ', a)*: Prints the global *a*. Prints *Value of a Inside h() : 3*.
- *print('Value of 'a' after h() : ', a)*: Prints the global *a*, which is now *3*.

5. (a) Write a program to check whether a given number is prime or not.

-
  ```
  def is_prime(num):
      """
      Checks if a given number is prime.
      A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.
      """
      if num <= 1:
          return False  # Numbers less than or equal to 1 are not prime
      if num == 2:
          return True   # 2 is the only even prime number
      if num % 2 == 0:
          return False  # Other even numbers are not prime

      # Check for divisibility from 3 up to the square root of num, only odd numbers
      i = 3
      while i * i <= num:
          if num % i == 0:
              return False
          i += 2
      return True
  ```

```
# Main part of the program
try:
    number = int(input("Enter an integer: "))
    if is_prime(number):
        print(f"{number} is a prime number.")
    else:
        print(f"{number} is not a prime number.")
except ValueError:
    print("Invalid input. Please enter an integer.")
```

(b) What will be the output produced on the execution of the following code snippet? *python color = set(['White', 'Green', 'Yellow', 'Blue']) primary = set(['Red', 'Green']) print('Green' in color) allcolor=color.union(primary) print(allcolor) print(color.intersection(primary)) print(color.difference(primary))*

-
  ```
  True
  {'Yellow', 'Blue', 'White', 'Red', 'Green'}
  {'Green'}
  {'Yellow', 'Blue', 'White'}
  ```
- **Justification:**
    - *print('Green' in color)*: Checks if 'Green' is an element of the *color* set. It is, so *True* is printed.
    - *allcolor=color.union(primary)*: Performs a union of the two sets, combining all unique elements

from both sets. The order of elements in a set's string representation is not guaranteed.

- *print(color.intersection(primary))*: Finds the common elements between *color* and *primary*. 'Green' is the only common element.
- *print(color.difference(primary))*: Finds elements present in *color* but not in *primary*. These are 'White', 'Yellow', and 'Blue'.

(c) Consider the following string : *python Pname = "Programming Fundamentals Using Python"* What will be the output produced on executing the following statements?

- (i) *Pname.count('P')*
  - *2*
  - **Justification:** Counts the occurrences of the character 'P'. 'Programming' and 'Python' both start with 'P'.
- (ii) *Pname.swapcase()*
  - *pROGRAMMING fUNDAMENTALS uSING pYTHON*
  - **Justification:** Converts all uppercase characters to lowercase and all lowercase characters to uppercase.
- (iii) *Pname.rfind('h')*

- *29*
- **Justification:** *rfind()* returns the highest index in the string where the substring 'h' is found. The 'h' in 'Python' is at index 29 (0-indexed).

- (iv) *Pname.split(' ')*
  - *['Programming', 'Fundamentals', 'Using', 'Python']*
  - **Justification:** Splits the string into a list of substrings using the space character (' ') as a delimiter.

6. (a) Differentiate between class variable and instance variable. Write a complete program that defines a class Bank that keeps track of the bank customers. The class should contain the following data members:
   - *accountNum* - Account number of the customer
   - *name* - Name of the customer
   - *balance* - Amount deposited in account. The class should contain the following methods :
   - *(i) init* for initializing the data members.
   - *(ii) deposit()* to deposit money in the bank account.
   - *(iii) str ()* that returns string representation for displaying the data members in a suitable manner.

- **Class Variable vs. Instance Variable:**
  - **Class Variable:**
    - **Definition:** A variable that is shared by all instances (objects) of a class. It is defined directly within the class body, outside of any methods.
    - **Scope:** Belongs to the class itself, not to any specific instance. Changes made to a class variable by one instance will be reflected in all other instances.
    - **Access:** Accessed using the class name (e.g., *ClassName.class_variable*) or through an instance (e.g., *instance_name.class_variable*), though direct modification via an instance (e.g., *instance_name.class_variable = new_value*) typically creates a new instance variable of the same name, rather than modifying the class variable.
    - **Use Case:** Useful for storing data that is common to all instances of a class, such as a constant, a counter for the number of objects created, or a default value.
  - **Instance Variable:**

- **Definition:** A variable that belongs to a specific instance (object) of a class. Each instance has its own copy of the instance variables. It is typically defined inside the constructor (__*init*__ method) using *self.variable_name*.
- **Scope:** Specific to the object. Changes made to an instance variable of one object do not affect the instance variables of other objects.
- **Access:** Accessed using the instance name (e.g., *instance_name.instance_variable*).
- **Use Case:** Used to store data that varies from one object to another, representing the unique state of each object.

- **Program for Bank Class:**

```
class Bank:
    # Class variable to keep track of the next available account number
    # (assuming sequential account numbers starting from a base)
    next_account_num = 1001

    def __init__(self, name, initial_balance=0):
        """
        (i) Initializes the data members for a new bank
```

```
customer.
        """
        self.accountNum = Bank.next_account_num
        Bank.next_account_num += 1  # Increment for
the next customer
        self.name = name
        self.balance = initial_balance
        print(f"Account created for {self.name} with
Account Number: {self.accountNum}")

    def deposit(self, amount):
        """
        (ii) Deposits money into the bank account.
        """
        if amount > 0:
            self.balance += amount
            print(f"Deposited ${amount:.2f}. New balance:
${self.balance:.2f}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        """
        Withdraws money from the bank account. (Added
for completeness, though not explicitly asked)
        """
        if amount > 0:
            if self.balance >= amount:
                self.balance -= amount
                print(f"Withdrew ${amount:.2f}. New
balance: ${self.balance:.2f}")
            else:
                print("Insufficient balance.")
        else:
            print("Withdrawal amount must be positive.")

    def __str__(self):
        """
        (iii) Returns a string representation for
displaying the data members.
        """
        return (f"Account Number: {self.accountNum}\n"
                f"Customer Name: {self.name}\n"
                f"Current Balance: ${self.balance:.2f}")

# Example Usage:
print("--- Creating Bank Accounts ---")
customer1 = Bank("Alice Smith", 500)
customer2 = Bank("Bob Johnson") # Default initial
balance is 0

print("\n--- Displaying Account Details ---")
print(customer1)
print("\n" + str(customer2)) # Using str() explicitly
print("\n--- Performing Transactions ---")
customer1.deposit(200)
customer1.withdraw(100)
customer1.withdraw(700) # Should show insufficient
balance

customer2.deposit(150.75)

print("\n--- Account Details After Transactions ---")
print(customer1)
print("\n" + str(customer2))
```

(b) Consider the following Pandas DataFrame that represents the monthly revenue of a company for the first quarter of the year : *python import pandas as pd data = {"Revenue": } df = pd.DataFrame(data, index=["January", "February", "March"])* Perform the following tasks:

- (i) Calculate the total revenue for the first quarter.
- (ii) Identify the month with the highest revenue.
- (iii) Create a new column called "Revenue Category" where revenue greater than 60,000 is labeled as "High" and the rest as "Low."

Assuming the data dictionary was intended to have values, let's use *data = {"Revenue": [55000, 65000, 70000]}* for "January", "February", "March" respectively.

```
o   import pandas as pd

    # Assuming the data for Revenue was intended to be
  like this:
    data = {"Revenue": [55000, 65000, 70000]}
    df = pd.DataFrame(data, index=["January",
  "February", "March"])

    print("Original DataFrame:")
    print(df)
    print("-" * 30)

    # (i) Calculate the total revenue for the first quarter.
    total_revenue = df["Revenue"].sum()
    print(f"(i) Total Revenue for the first quarter:
  ${total_revenue:,.2f}")
    print("-" * 30)

    # (ii) Identify the month with the highest revenue.
    highest_revenue_month = df["Revenue"].idxmax()
    highest_revenue_amount = df["Revenue"].max()
    print(f"(ii) Month with the highest revenue:
  {highest_revenue_month}
  (${highest_revenue_amount:,.2f})")
    print("-" * 30)

    # (iii) Create a new column called "Revenue
```

```
Category" where revenue greater than 60,000
  # is labeled as "High" and the rest as "Low."
  df["Revenue Category"] =
df["Revenue"].apply(lambda x: "High" if x > 60000
else "Low")
  print("(iii) DataFrame with 'Revenue Category'
column:")
  print(df)
```

- **Output for (b):**

```
Original DataFrame:
         Revenue
January    55000
February   65000
March      70000
------------------------------
(i) Total Revenue for the first quarter: $190,000.00
------------------------------
(ii) Month with the highest revenue: March
($70,000.00)
------------------------------
(iii) DataFrame with 'Revenue Category' column:
         Revenue Revenue Category
January    55000             Low
February   65000             High
March      70000            High
```

(c) Write a program that takes the number of lines *n* as an input from the user and prints the following pattern (say, for *n = 6*): *

- This looks like a simple rectangular pattern. For *n = 6*, it shows 6 rows of 6 asterisks. If the pattern is *n* lines of *n* asterisks.

```
def print_pattern():
    try:
        n = int(input("Enter the number of lines (n): "))
        if n <= 0:
            print("Please enter a positive integer for n.")
            return

        for _ in range(n):
            print("* " * n) # Prints n asterisks separated
by spaces

    except ValueError:
        print("Invalid input. Please enter an integer.")

print_pattern()
```

- **Output for n = 6:**

```
* * * * * *
* * * * * *
* * * * * *
* * * * * *
* * * * * *
* * * * * *
```

- If the user meant exactly * (a single line with 20 asterisks regardless of *n*), then the program would be:

```
def print_fixed_pattern():
    print("* " * 20)

# print_fixed_pattern()
```

Given the phrasing "prints the following pattern (say, for n = 6): *", it is ambiguous. The most common interpretation of "pattern for n=X" implies X influences the output. Assuming a square pattern where *n* lines each have *n* asterisks is the intention. If not, the provided output for *n=6* is a single line of 20 asterisks, which makes *n* irrelevant to the length of the line. Sticking with the interpretation that *n* affects the pattern structure.

7. (a) The monthly sale and income values of a shop are as below : Extracted Table Table: Monthly Sale and Income Values

Month	1	2	3	4	5	6
Sale	100	200	300	300	250	400

Month	1	2	3	4	5	6
Income	10	15	30	20	20	45
Write a program to show Monthly Sale and Income as subplots.						

-
  ```
  import matplotlib.pyplot as plt
  import numpy as np

  # Data from the table
  months = [1, 2, 3, 4, 5, 6]
  sale = [100, 200, 300, 300, 250, 400]
  income = [10, 15, 30, 20, 20, 45]

  # Create a figure and a set of subplots
  fig, axes = plt.subplots(2, 1, figsize=(8, 6)) # 2 rows,
  1 column of subplots

  # Subplot 1: Monthly Sale
  axes[0].plot(months, sale, marker='o', color='blue')
  axes[0].set_title('Monthly Sale')
  axes[0].set_xlabel('Month')
  axes[0].set_ylabel('Sale Value')
  axes[0].set_xticks(months) # Ensure all months are
  shown as ticks
  axes[0].grid(True, linestyle='--', alpha=0.7)
  ```

```
  # Subplot 2: Monthly Income
  axes[1].plot(months, income, marker='x',
color='green')
  axes[1].set_title('Monthly Income')
  axes[1].set_xlabel('Month')
  axes[1].set_ylabel('Income Value')
  axes[1].set_xticks(months) # Ensure all months are
shown as ticks
  axes[1].grid(True, linestyle='--', alpha=0.7)

  # Adjust layout to prevent overlapping titles/labels
  plt.tight_layout()

  # Display the plots
  plt.show()
```

(b) Consider the following recursive function : *python def recurFunc(n): if n==0: return 1 else: return n * recurFunc(n-1)* Determine and explain the output produced on execution of the following statements : *python result = recurFunc(3) print(result)*

- **Output:** *6*
- **Explanation:** The *recurFunc(n)* is a recursive function that calculates the factorial of *n* (n!). Here's how it executes for *recurFunc(3)*:

  i. ***recurFunc(3):***

- *n* is *3*. *n == 0* is false.
- Returns *3 * recurFunc(2)*

ii. ***recurFunc(2)*****:** (Called from *recurFunc(3)*)

- *n* is *2*. *n == 0* is false.
- Returns *2 * recurFunc(1)*

iii. ***recurFunc(1)*****:** (Called from *recurFunc(2)*)

- *n* is *1*. *n == 0* is false.
- Returns *1 * recurFunc(0)*

iv. ***recurFunc(0)*****:** (Called from *recurFunc(1)*)

- *n* is *0*. *n == 0* is true.
- Returns *1* (Base case reached)

Now, the results are returned up the call stack:

- *recurFunc(1)* receives *1* from *recurFunc(0)*, so it calculates *1 * 1 = 1*.
- *recurFunc(2)* receives *1* from *recurFunc(1)*, so it calculates *2 * 1 = 2*.
- *recurFunc(3)* receives *2* from *recurFunc(2)*, so it calculates *3 * 2 = 6*.

Finally, *result* is assigned *6*, and *print(result)* outputs *6*.

(c) What will be output for each print statements listed below (i) to (v) for the code snippet? *python import numpy as np arr_2d = np.array([, ,])* Assuming the *arr_2d* was intended to have valid numerical data, for example: *arr_2d = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])*

- (i) *print(arr_2d)*

  -
    ```
    [[10 20 30]
     [40 50 60]
     [70 80 90]]
    ```
  - **Justification:** Prints the entire 2D NumPy array.
- (ii) *print(arr_2d)*
  - **Error:** This is a duplicate of (i) and would produce the same output. Assuming it meant a different operation, but as written, it's just repeating.
- (iii) *print(arr_2d)*
  - **Error:** This is a duplicate of (i) and (ii) and would produce the same output. Assuming it meant a different operation, but as written, it's just repeating.
- (iv) *print(arr_2d[2,:])*
  - *[70 80 90]*

- **Justification:** Slices the array to get the row at index *2* (the third row). The *:* indicates all columns in that row.

- (v) *print(arr_2d[:,:])*

  -
    ```
    [[10 20 30]
     [40 50 60]
     [70 80 90]]
    ```
  - **Justification:** Slices the entire array. The first *:* indicates all rows, and the second *:* indicates all columns. This effectively prints the entire array.