

# Top 43 Pattern Programs in Python to Master Loops and Recursion

By Rohit Sharma

Updated on May 09, 2025 | 38 min read | 47.58K+ views

Share:   

 [Table of Contents](#)

*Did you know ? Python Pattern programs are used in the product recommendation mechanism of global e-commerce giant Amazon!*

A pattern program in Python involves printing characters, numbers, or symbols in specific sequences to form distinct visual structures. These structures can be pyramids, triangles, diamonds, and other creative shapes. In a developer's world, pattern programs help you sharpen your understanding of loops, [conditional statements](#), and output formatting.

Here is why Python pattern programs matter:

- They strengthen your ability to write efficient loops and manage nested iterations.
- They [reinforce logical thinking](#) because each pattern demands a clear plan for spacing and alignment.
- They help you practice concise code by tackling repetitive printing tasks in a structured way.

In this blog, you will see a variety of Python pattern programs, from basic shapes to more advanced ones. This range helps you apply different logical and programming concepts so you can handle any pattern-related requirement. Let's get started.

*Build the future with code! Explore our range of [Software Engineering Courses](#) and kickstart your journey to becoming a tech expert.*

## What Are the Essential Concepts for Printing a Pattern Program in Python? Prerequisites You Must Know



This section focuses on the core ideas that guide you when you build a pattern program in Python. You will see them in action each time you handle loops or output formatting.

## Free Courses

Explore courses related to Data Science



### Introduction to Data Analysis using Excel

899.16K+ learners

9 hrs of learning

Learn For Free

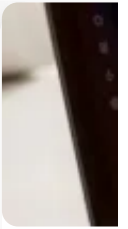


### Learn Basic Python Programming

42.74K+ learners

5 hrs of learning

Learn For Free



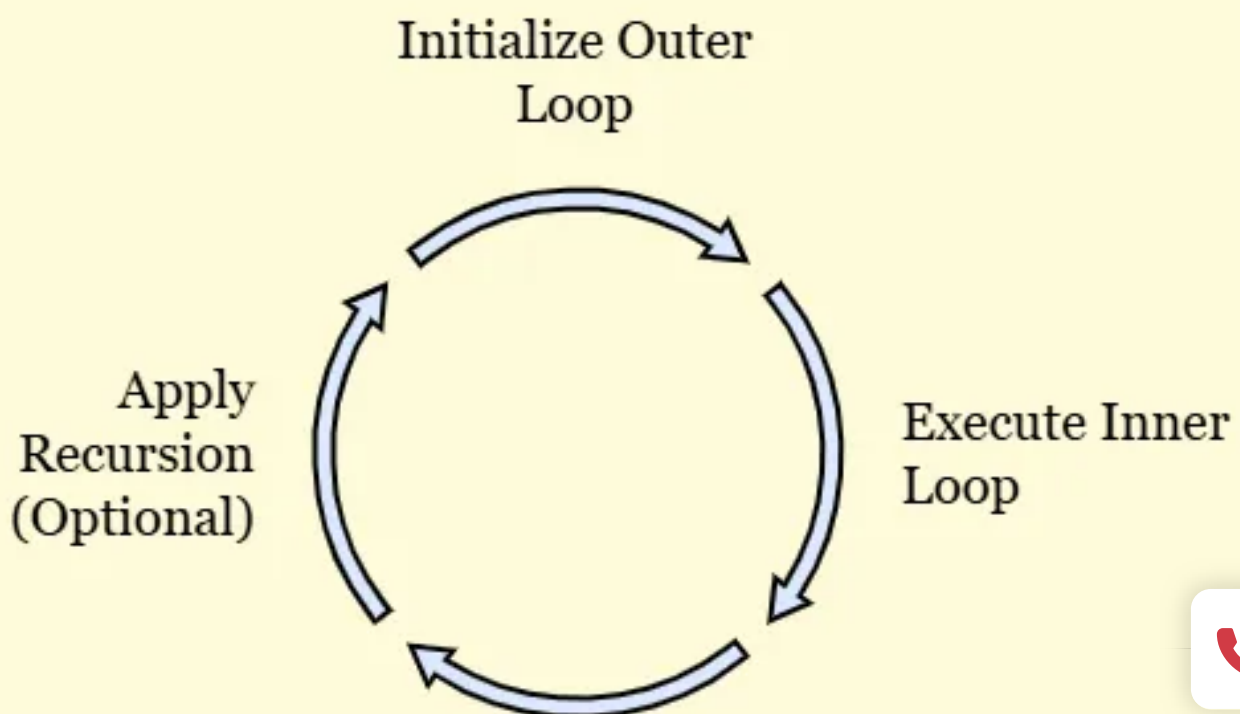
### Analysis Storytelling

40.1K+ learners

6 hrs of learning

[Explore all >](#)

## Cycle of Pattern Program Development



# Manage Output

Pave the path for a rewarding tech career with some of our top programs:

- [Generative AI Mastery Certificate for Software Development](#)
- [AI-Powered Full Stack Development Course by IIITB](#)
- [AI-Driven Full-Stack Development Bootcamp](#)

Take note of these three key points before you begin.

## 1. Nested Loops

You will always rely on two loops for pattern programs:

- The outer loop tracks rows
- The inner loop handles columns

The inner loop is where you choose what to print for every position in a row. That might be a star, a number, or a character. Think of the outer loop as setting the row count while the inner loop fills each row.

Here is a quick example where:

- The outer loop runs from 0 to three
- The inner loop prints the exact number of stars for each row

```
n = 4

# Outer loop handles the rows.
for row in range(n):
    # Inner loop prints exactly (row + 1) stars.
    for col in range(row + 1):
        print("*", end=" ")
    print() # Moves to a new line after each row
```

</> Cop



Output:

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

[</> Copy Code](#)

This code prints one star in the first row, two in the second, and so on until it reaches four stars in the last row.

Also Read: [Nested for Loop in Python: A Complete Guide](#)

## 2. Managing Spaces and Output

You can align shapes by adding spaces before or between printed items. A simple way to do this is with string multiplication, such as " " \* 4, to add four spaces at once.

Also, `print(..., end=" ")` keeps output on the same line, which is a handy technique for any pattern that needs characters in a single row.

Take a look at this example: It prints an increasing number of stars but centers them by adding leading spaces:

```
rows = 4

# Outer loop creates the rows, from 0 to 3.
for row in range(rows):
    # Print leading spaces so the stars appear centered.
    print(" " * (rows - row - 1), end="")
    # Print a series of stars in the current row.
    print("* " * (row + 1))
```

[</> Copy Code](#)

Output:

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



```
* * * *
```

[</> Copy Code](#)

In this code:

- `print(" " * (rows - row - 1), end="")` aligns the stars by shifting them to the right.
- Then `print("* " * (row + 1))` prints an increasing number of stars in each subsequent row.

### 3. Recursion (Optional)

Sometimes, a function can call itself to form patterns. This method can be helpful for pyramids or similar shapes that expand row by row. You can build from the smallest row to the largest or the other way around. It is best to test [recursion in Python](#) on small patterns first because it might complicate debugging.

Let's understand this with the help of an example code.

In the following code:

- The function stops calling itself when level is zero.
- Each call prints one row of stars, then returns to the previous call

```
def print_pyramid(level):  
    if level == 0:  
        return  
    # Build the smaller part of the pyramid first.  
    print_pyramid(level - 1)  
    # Print the stars for the current level.  
    print("* " * level)  
  
print_pyramid(4)
```

[</> Copy Code](#)

Output:


```
*
```



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


[</> Copy Code](#)

The recursion starts at `level = 4` but calls itself for `level = 3`, `level = 2`, and so on. Each of those calls prints a row of stars, resulting in a neat ascending pattern.



Liverpool John Moores University  
**MS in Data Science**  
Dual Credentials  
Master's Degree 17 Months

[View Program](#)[↓ Syllabus](#)



IIIT Bangalore  
**Post Graduate Certificate in Data Science & AI (Executive)**  
Placement Assistance  
Certification 6 Months

[View Program](#)[↓ Syllabus](#)

Want to strengthen your basics in Python? Check out this [free certificate course by upGrad, Learn Basic Python Programming](#). Master fundamentals, real-world applications & hands-on exercises with just 12 hours of learning.

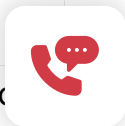
## 13 Python Star Pattern Programs (With Code and Output)

Star patterns are a staple when you explore pattern programs in Python. You will see half pyramids, diamonds, and other shapes that highlight how a few changes in loops or spacing lead to new designs.

Each example in this section helps you practice simple row-by-row logic and produce eye-catching output with minimal code.

### 4 Basic Star Pattern Programs in Python

These patterns highlight fundamental star shapes. You will find half pyramids the



expand or shrink each row and discover ways to center them for a full pyramid effect. Working through these variations builds a solid base for more elaborate star patterns.

## 1. Half Pyramid (Upright)

Here, you add one extra star in each new row until you reach the total number of rows. The loop uses the row index to decide how many stars to print, and then it prints them on the same line.

```
n = 5
for i in range(n):
    for j in range(i + 1):
        print("*", end=" ")
    print()
```

[</> Copy Code](#)

Output:

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

[</> Copy Code](#)

Each row grows by one star, so the final row contains five stars.

## 2. Half Pyramid (Inverted)

In this code, you begin with the full count of stars in the top row and subtract one star in every subsequent row. This approach reverses the logic of the upright half pyramid.

```
n = 5
for i in range(n, 0, -1):
    for j in range(i):
        print("*", end=" ")
    print()
```



[</> Copy Code](#)

Output:

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

[</> Copy Code](#)

You start with five stars and drop one star each time until only one remains.

### 3. Full Pyramid (Center-Aligned)

In this code, you center the stars by printing spaces before the stars. The number of spaces falls as the row grows, while the stars themselves form a symmetrical shape.

```
n = 5
for i in range(n):
    print(" " * (n - i - 1), end="")
    print("* " * (i + 1))
```

[</> Copy Code](#)

Output:

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

[</> Copy Code](#)

Leading spaces shift the stars to the right, creating a centered triangular look.

### 4. Full Pyramid (Inverted)

In this code, you flip the centered shape so the widest row appears at the top. Tl





you begin with no leading spaces in the first row, then increase them each time.

```
n = 5
for i in range(n, 0, -1):
    print(" " * (n - i), end="")
    print("* " * i)
```

[</> Copy Code](#)

Output:

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

```

[</> Copy Code](#)

The first row prints the maximum stars. Each subsequent row has one more space and one fewer star, producing an upside-down pyramid.

*If you're a true beginner, upGrad's free tutorial, [Python for Loop: A Comprehensive Guide to Iteration](#), will greatly benefit you.*

## 2 Right-Angled & Mirrored Triangles

These triangles focus on where you position the stars. One uses leading spaces to shift stars to the right, while the other lets stars stack up on the left side without extra spacing.

### 1. Right-Aligned Triangle

In this code, you print enough spaces in each row to move the stars toward the right boundary. The number of spaces decreases on every row while the number of stars increases.

```
n = 5
for i in range(1, n + 1):
    # Print spaces to shift stars to the right
```



```
for space in range(n - i):  
    print(" ", end="")  
# Print the stars for the current row  
for star in range(i):  
    print("*", end="")  
print()
```

[</> Copy Code](#)

Output:

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

[</> Copy Code](#)

- The first nested loop inserts (n - i) spaces
- The second nested loop prints i stars.

The result is a neat triangle pushed to the right edge.

## 2. Left-Aligned Triangle

This shape grows one star at a time and does not include extra spaces in front. Each row simply prints more stars than the previous row.

```
n = 5  
for i in range(1, n + 1):  
    print("*" * i)
```

[</> Copy Code](#)

Output:

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



```
*****
```

[</> Copy Code](#)

Here, `print("'" * i)` directly multiplies the star symbol by the row number. That keeps the stars flush on the left side, with no space-based indentation.

Also Read: [While Loop in Python \[With Syntax and Examples\]](#)

## 2 Diamond Pattern Programs in Python

These diamond patterns create a symmetrical shape that expands from a single row of stars to a maximum width, then contracts back again. In the hollow version, stars appear on the boundary while the inside remains blank.

### 1. Solid Diamond

This code builds an upper triangle first, then a lower triangle. Each line calculates the number of spaces and stars to print, ensuring the shape remains centered.

```
n = 4
# Upper half
for i in range(1, n + 1):
    print(" " * (n - i), end="")
    print("'" * (2 * i - 1))
# Lower half
for i in range(n - 1, 0, -1):
    print(" " * (n - i), end="")
    print("'" * (2 * i - 1))
```

[</> Copy Code](#)

Output:

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



The first loop goes from 1 to n, increasing the stars from 1 to 7 in this case, while spaces decrease. The second loop mirrors this logic by counting down and reducing the star count row by row.

## 2. Hollow Diamond

In this code, the outer stars form the diamond border, and the interior remains empty. Each row still calculates leading spaces for positioning.

```
n = 4
# Upper half
for i in range(1, n + 1):
    print(" " * (n - i), end="")
    for j in range(1, 2 * i):
        if j == 1 or j == 2 * i - 1:
            print("*", end="")
        else:
            print(" ", end="")
    print()
# Lower half
for i in range(n - 1, 0, -1):
    print(" " * (n - i), end="")
    for j in range(1, 2 * i):
        if j == 1 or j == 2 * i - 1:
            print("*", end="")
        else:
            print(" ", end="")
    print()
```

Output:

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



The code checks whether `j` is at the beginning or end of each row's star count. If it is, it prints `*`; otherwise, it prints a space. This way, only the outline of the diamond is visible, leaving the center hollow.

## Hourglass / Sandglass & Pant/Bow-Tie Patterns

Both of these patterns revolve around mirroring. The hourglass shape starts with a full row of stars and narrows down. The pant or bow-tie shape creates two triangular sections facing each other.

### 1. Hourglass or Sandglass

The code prints a decreasing number of stars row by row until it hits the narrowest point, then prints increasing stars again. Spaces at the start of each line keep the shape centered.

```
n = 5
# Upper half
for i in range(n, 0, -1):
    print(" " * (n - i), end="")
    print("* " * i)
# Lower half
for i in range(2, n + 1):
    print(" " * (n - i), end="")
    print("* " * i)
```

Output:



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

[</> Copy Code](#)

Initially, you have five stars centered at the top. Each row adds a space and reduces a star, forming the hourglass. Then, you reverse the pattern from two to five stars to complete the shape.

## 2. Pant / Bow-Tie Style

This pattern creates two back-to-back triangles, making it look like pants or a bow tie. You often print underscores or spaces in the middle.

```
rows = 5
# Print the first row of stars
print("*" * (2 * rows), end="\n")
i = (rows // 2) - 1
j = 2
# Create the "legs" or "bow" sections
while i != 0:
    print("*" * i, end="")
    print(" " * j, end="")
    print("*" * i)
    i -= 1
    j += 2
```

[</> Copy Code](#)

Output:



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

[</> Copy Code](#)

The code prints a full row of stars, then gradually reduces the stars on each side while inserting more spaces in the center. This mirrored approach leads to the distinct pant or bow-tie layout.

## Two Pyramids (Side-by-Side or Stacked)

Sometimes, you want to display two pyramids next to each other or one on top of the other. This approach involves duplicating the logic of a half pyramid but assigning different alignment or direction to each segment.

### 1. Side-by-Side Pyramids

The code prints one half pyramid, then immediately prints the second half pyramid on the same row. This creates a mirrored effect with a gap or no gap in the middle.

```
rows = 4
for i in range(1, rows + 1):
    # First half pyramid
    print("* " * i, end="")

    # Optional space or separator between pyramids
    print(" ", end="")

    # Second half pyramid
    print("* " * i)
```

[</> Copy Code](#)

Output:

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



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

[</> Copy Code](#)

Here, the first `print("* " * i, end="")` statement builds the left pyramid, and the next `print("* " * i)` statement builds the right pyramid. A small space in the middle prevents the patterns from merging into one.

## 2. Stacked Pyramids

One pyramid appears above the other. The top pyramid often goes from one star to a maximum row of stars, while the lower pyramid goes in reverse.

```
rows = 4
# Upper pyramid
for i in range(1, rows + 1):
    print("* " * i)
print() # Blank line to separate the two

# Lower pyramid
for i in range(rows, 0, -1):
    print("* " * i)
```

[</> Copy Code](#)

Output:

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

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

[</> Copy Code](#)

The first loop builds an upright half pyramid until it reaches four stars. A blank line then separates it from the next loop, which prints the same shape in reverse.





## Hollow Squares (Stars)

A hollow square shows stars along the perimeter and leaves the center empty. You typically check if the current position is on the first or last row or column, printing a star if true and a space otherwise.

```
size = 5
for row in range(size):
    for col in range(size):
        if row == 0 or row == size - 1 or col == 0 or col == size - 1:
            print("*", end=" ")
        else:
            print(" ", end=" ")
    print()
```

[</> Copy Code](#)

Output:

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

[</> Copy Code](#)

The `if` statement checks whether you are at the boundary row or column. If yes, it prints `*`. All other spots print a space, creating that hollow center.

Enhance your Python skills further with upGrad's [Data Science](#) and [Machine Learning courses](#) from top universities — take the next step in your learning journey!

## 18 Number Pattern Programs in Python (With Code and Output)



Number patterns let you arrange digits in structured rows and columns, from simple counting to complex sequences. They typically rely on loops in creative ways, but they add a numeric twist that helps you practice everything from basic increments to binomial coefficients.

This section covers half pyramids, inverted patterns, Pascal's Triangle, and other variations.

## Simple & Half-Pyramid Number Patterns

These patterns show you how to arrange numbers in basic ascending rows. Some grow from left to right, while others start at a maximum and shrink each time. They rely on looping through rows and columns to print an increasing or decreasing sequence of digits.

### 1. Half Pyramid (1, 2, 3...)

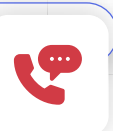
You print row indices in ascending order, so row one shows one digit, row two shows two digits, and so on until you reach the final row.

```
rows = 5
for i in range(1, rows + 1):
    for j in range(1, i + 1):
        print(j, end=" ")
    print()
```

[</> Copy Code](#)

Output:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

[</> Cop](#)

Each row uses  $(i + 1)$  in the inner loop, which prints all integers from 1 up to the cu

row number. That is why the second row shows 1 2, the third row shows 1 2 3, and so on.

## 2. Inverted Half Pyramid

Here, you start with the maximum set of numbers in the top row and reduce the count in each new row until you reach one digit.

```
rows = 5
for i in range(rows, 0, -1):
    for j in range(1, i + 1):
        print(j, end=" ")
    print()
```

[</> Copy Code](#)

Output:

```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

[</> Copy Code](#)

In the code:

- The loop for i runs backward, starting from 5 down to 1.
- The inner loop prints numbers from 1 through that row's count, which steadily decreases every time.

## Full / Centered Number Pyramids

These patterns arrange digits in a centered format, creating shapes that expand in the middle. Some build up symmetrically around a peak row, and others start with the widest row and shrink as you move downward.

### 1. Symmetrical Pyramid



This pattern shows numbers increasing from 1 up to the row number, then decreasing back to 1. Spaces on the left keep everything centered.

```
n = 5
for i in range(1, n + 1):
    # Print leading spaces
    print(" " * (n - i), end="")
    # Ascending numbers
    for j in range(1, i + 1):
        print(j, end="")
    # Descending numbers
    for j in range(i - 1, 0, -1):
        print(j, end="")
    print()
```

[</> Copy Code](#)

Output:

```
  1
 121
12321
1234321
123454321
```

[</> Copy Code](#)

Each row begins with  $(n - i)$  spaces for alignment. Then the code prints ascending numbers from 1 to  $i$ , followed by descending numbers back down to 1, creating a mirror effect.

## 2. Inverted Full Pyramid

Here, the widest row is at the top, and each step down reduces the number of digits. Spaces increase steadily to keep the pattern centered.

```
n = 5
for i in range(n, 0, -1):
    # Leading spaces for centering
    print(" " * (n - i), end="")
    # Print the numbers from 1 to i
    for j in range(1, i + 1):
```



```
    print(j, end="")
# Print the numbers from i-1 down to 1
for j in range(i - 1, 0, -1):
    print(j, end="")
print()
```

[</> Copy Code](#)

Output:

```
123454321
1234321
 12321
   121
    1
```

[</> Copy Code](#)

The loop counts down from 5 to 1. At each level, the code prints the digits in ascending and then descending order, all shifted to the right by  $(n - i)$  spaces to form an inverted pyramid.

## Reverse & Descending Number Patterns

These patterns focus on counts going from higher numbers to lower ones. One version starts with the largest row of digits at the top, while another decreases digit values in each row or column.

### 1. Reverse Pyramid

You begin at the maximum limit and work your way down per row. The code places numbers side by side, reducing the row's length after each iteration.

```
rows = 5
for i in range(rows, 0, -1):
    for j in range(1, i + 1):
        print(j, end=" ")
    print()
```

[</> Cop](#)

Output:

```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

[</> Copy Code](#)

The outer loop decreases from 5 to 1, and the inner loop prints numbers up to the current row count. This creates a top-down approach, with each row having fewer numbers than the last.

## Advanced Number Sequences

These patterns move beyond straightforward ascending and descending lists. One example uses odd or even numbers to form pyramids, while another arranges continuous sequences that span multiple rows in a single numeric run.

### 1. Alternate / Odd / Even Number Pyramids

You can decide to print only odd values, only even values, or use a doubling approach. The row index typically influences the number of repetitions and the numeric progression.

```
# Example: Odd Number Pyramid
rows = 5
for i in range(1, rows + 1):
    for j in range(i):
        # Each row prints the same odd number multiple times
        print((2 * i - 1), end=" ")
    print()
```

[</> Copy Code](#)

Output:

```
1
3 3
5 5 5
7 7 7 7
```



9 9 9 9 9

[</> Copy Code](#)

In this snippet,  $(2 * i - 1)$  calculates the odd number for the current row. You repeat that value  $i$  times, so row one shows 1 once, row two shows 3 twice, and so on.

## 2. Continuous Natural Number Pyramids

This style keeps counting through a single numeric thread. Each new row just picks up where the last row stopped, so you never reset to 1 at the start of a row.

```
rows = 3
current = 1
stop = 2
for i in range(rows):
    for col in range(1, stop):
        print(current, end=" ")
        current += 1
    print()
    stop += 2
```

[</> Copy Code](#)

Output:

```
1
2 3 4
5 6 7 8 9
```

[</> Copy Code](#)

Here, the `current` variable tracks the next number to print. Each row demands more numbers than the previous one, so by row three, you print five consecutive values, continuing from where row two left off.

## Pascal's Triangle

Pascal's Triangle arranges numbers in rows, each built from the one above it. Every number is the sum of two values from the previous row. This structure introduces binomial coefficients, which appear often in combinatorial mathematics.



Below is how you can produce Pascal's Triangle step by step:

```
def print_pascal_triangle(n):
    for i in range(n):
        num = 1
        # Print leading spaces for basic alignment
        print(" " * (n - i - 1), end="")
        for j in range(i + 1):
            print(num, end=" ")
            # Update num based on binomial coefficient logic
            num = num * (i - j) // (j + 1)
        print()

print_pascal_triangle(5)
```

[</> Copy Code](#)

Output:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

[</> Copy Code](#)

Each row starts with **num = 1**, then you multiply and divide to calculate the next coefficient. You shift the rows to the right with spaces so the triangle stays somewhat centered.

## Floyd's Triangle

Floyd's Triangle is a continuous sequence of numbers arranged in rows. Each new row adds one more number than the row before it. This pattern starts at 1, then moves on without resetting.

```
def floyds_triangle(rows):
    num = 1
    for i in range(1, rows + 1):
        for j in range(i):
            print(num, end=" ")
```





```
        num += 1
    print()

floyds_triangle(5)
```

[</> Copy Code](#)

Output:

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

[</> Copy Code](#)

Each row consumes the next set of natural numbers. The first row prints just **1**, the second row prints **2 3**, and so on. This pattern grows row by row without resetting **num**.

## Multiplication Table Patterns

These patterns arrange multiplication results in rows, either focusing on row-by-row increments or showing more columns with each step. They showcase loop nesting while multiplying indices.

### 1. Simple Multiplication Table in Rows

Each row multiplies the row index by columns that go from 1 to the row number. This creates a standard multiplication progression.

```
rows = 5
for i in range(1, rows + 1):
    for j in range(1, i + 1):
        print(i * j, end=" ")
    print()
```

[</> Copy Code](#)

Output:



```
1
2 4
3 6 9
4 8 12 16
5 10 15 20 25
```

[</> Copy Code](#)

The outer loop starts at 1 and goes to **rows**. The inner loop runs up to the current row number, so the first row prints one product, the second row prints two products, and so on.

## 2. Horizontal Tables

Each row still relies on nested loops, but it produces a table-like view of multiplication, often expanding columns in a more uniform way.

```
n = 4
for i in range(1, n + 1):
    for j in range(1, n + 1):
        # Print the product with some spacing
        print(f"{i*j:3}", end=" ")
    print()
```

[</> Copy Code](#)

Output:

```
1  2  3  4
2  4  6  8
3  6  9 12
4  8 12 16
```

[</> Copy Code](#)

This code loops **i** from 1 to 4 for the rows and **j** from 1 to 4 for each column, printing **i\*j** in a grid. The format string **"{i\*j:3}"** spaces the results evenly.

## Spiral, Zigzag, & Hourglass Number Patterns

These patterns use more complex logic and often involve two-dimensional array



mirrored sequences. Each example goes beyond basic row-by-row printing.

### 1. Spiral Number Pattern (2D Grid)

In this code, you fill a square matrix with numbers in a spiral layout. The loop manages boundaries: top, bottom, left, and right. You move in one direction, then adjust a boundary, and continue until the matrix is full.



```
def spiral_print(n):
    matrix = [[0] * n for _ in range(n)]
    left, right = 0, n - 1
    top, bottom = 0, n - 1
    num = 1

    while left <= right and top <= bottom:
        # Move from left to right
        for col in range(left, right + 1):
            matrix[top][col] = num
            num += 1
        top += 1

        # Move from top to bottom
        for row in range(top, bottom + 1):
            matrix[row][right] = num
            num += 1
        right -= 1

        if top <= bottom:
            # Move from right to left
            for col in range(right, left - 1, -1):
                matrix[bottom][col] = num
                num += 1
            bottom -= 1

        if left <= right:
            # Move from bottom to top
            for row in range(bottom, top - 1, -1):
                matrix[row][left] = num
                num += 1
            left += 1

    for row in matrix:
        print(" ".join(str(x) for x in row))

spiral_print(5)
```

[</> Copy Code](#)

Output:



```
1 2 3 4 5
16 17 18 19 6
15 24 25 20 7
14 23 22 21 8
13 12 11 10 9
```

[</> Copy Code](#)

The algorithm updates the matrix in concentric rings. Each time you traverse an edge, you increment a boundary (top or left) or decrement it (bottom or right) to move inward.

## 2. Zigzag Number Pattern

Here, the sequence flips its direction on alternate rows. One row ascends, the next row descends. This produces a zigzag across the output.

```
rows = 5
current = 1
for i in range(1, rows + 1):
    if i % 2 == 0:
        # Even row goes in descending order
        start = current + i - 1
        for j in range(i):
            print(start, end=" ")
            start -= 1
        current += i
    else:
        # Odd row goes in ascending order
        for j in range(i):
            print(current, end=" ")
            current += 1
        print()
```

[</> Copy Code](#)

Output:

```
1
3 2
4 5 6
10 9 8 7
11 12 13 14 15
```



[</> Copy Code](#)

Odd rows increment as usual, while even rows read backwards. Row two prints **3 2**, row four prints **10 9 8 7**, and so on.

### 3. Hourglass (Numbers)

You start with a wide set of digits on top that shrinks each row until it hits a single line, then you reverse and expand downward. Spaces center the shape.

```
n = 5
# Upper hourglass
for i in range(n, 0, -1):
    print(" " * (n - i), end="")
    for num in range(1, 2*i):
        print(num, end="")
    print()
# Lower hourglass
for i in range(2, n + 1):
    print(" " * (n - i), end="")
    for num in range(1, 2*i):
        print(num, end="")
    print()
```

[</> Copy Code](#)

Output:

```
123456789
1234567
12345
123
1
123
12345
1234567
123456789
```

[</> Copy Code](#)

The first loop counts down from **n** to **1**. Each row prints fewer numbers, shifting the leading spaces. The second loop brings the count back up, recreating the widened



shape at the bottom.

## Pant / Bow-Tie (Numbers)

This pattern arranges numbers in a bow-tie shape. The top portion starts with a wide range of descending digits on one side and ascending digits on the other, then narrows as you move downward. The middle row is often the widest point, and the pattern contracts again in the second half. The style gets its name from the two mirrored triangles that resemble pants legs or a bow tie.

Here is how the code snippet works: Each row prints descending numbers from  $(rows - 1)$  down to  $i + 1$ , then inserts some spacing, then moves back up with ascending numbers from  $(i + 1)$  to  $(rows - 1)$ . The loops must be carefully timed so the rows mirror each other perfectly.

```
rows = 6
for i in range(rows):
    # Print descending numbers
    for j in range(rows - 1, i, -1):
        print(j, end=" ")
    # Insert spacing in the center
    for l in range(i):
        print(" ", end=" ")
    # Print ascending numbers
    for k in range(i + 1, rows):
        print(k, end=" ")
    print()
```

[</> Copy Code](#)

Output:

```
5 4 3 2 1 1 2 3 4 5
4 3 2 1      1 2 3 4
3 2 1        1 2 3
2 1          1 2
1            1
```

[</> Cop](#)

In this output, the top row shows the widest span of numbers on both sides. Each

subsequent row removes one number from the outer segments while adding more spacing in the middle. By the time you reach the last row, numbers converge toward the center.

## Special Combined Patterns

These patterns merge numbers with star characters or use exponential growth in columns. They introduce more variety to your output by combining multiple logic streams, such as arithmetic progressions or repeated symbols.

### 1. Combining Numbers & Stars

You print a mix of numbers and stars on the same row. This code uses the row number to decide how many items appear, then alternates between printing a digit and a star.

```
rows = 4
for i in range(1, rows + 1):
    val = 1
    for j in range(2 * i - 1):
        if j % 2 == 0:
            print(val, end=" ")
            val += 1
        else:
            print("*", end=" ")
    print()
```

[</> Copy Code](#)

Output:

```
1
1 * 2
1 * 2 * 3
1 * 2 * 3 * 4
```

[</> Copy Code](#)

Each row prints digits and stars in an interleaved pattern. The loop for `j` goes up to  $(2 * i - 1)$ , ensuring an alternating sequence of numbers and stars.





## 2. Random Doubling / Exponential Patterns

This code prints powers of two in each row. Columns expand by one at a time, displaying values like 1, 2, 4, 8, and so on.

```
rows = 5
for i in range(1, rows + 1):
    for j in range(i):
        print(2 ** j, end=" ")
    print()
```

[</> Copy Code](#)

Output:

```
1
1 2
1 2 4
1 2 4 8
1 2 4 8 16
```

[</> Copy Code](#)

The inner loop runs from 0 to  $(i - 1)$ , and  $2 ** j$  calculates the power of two for each column index  $j$ . Each row produces a growing list of exponentials.

Start your coding journey with upGrad's complimentary Python courses designed just for you — dive into [Python programming fundamentals](#), explore [key Python libraries](#), and engage with [practical case studies](#)!

Also Read: [Python's do-while Loop](#)

## 8 Character (Alphabet) Pattern Programs in Python (With Code and Output)

Character-based patterns replace numbers and stars with letters from the alphabet.



They often rely on ASCII values to map integers to letters. You can create simple alphabet pyramids, inverted patterns, or even diamond shapes by adjusting how many letters to print in each row.

## Alphabet Triangles

These patterns arrange letters in ascending or descending order, similar to number pyramids but using alphabetic characters. You often rely on the ASCII value of 'A' (which is 65) and increment it to get subsequent letters.

### 1. Half Pyramid

This version starts at A and prints increasing letters in each row. Every row moves to the next ASCII value.

```
rows = 5
ascii_value = 65 # 'A'
for i in range(rows):
    for j in range(i + 1):
        print(chr(ascii_value), end=" ")
        ascii_value += 1
    print()
```

[</> Copy Code](#)

Output:

```
A
B C
D E F
G H I J
K L M N O
```

[</> Copy Code](#)

The code begins at ASCII 65 for A. Each row prints a growing list of letters, and `ascii_value` keeps incrementing, so the letters never reset to A again.

### 2. Inverted Half Pyramid



Here, you start with a row of letters, then each subsequent row prints one fewer letter. You still count up in ASCII, but use a separate strategy to decide how many letters appear in each row.

```
rows = 5
ascii_value = 65 # 'A'
# Build a list of letters first
letters = [chr(ascii_value + i) for i in range(rows * (rows + 1) // 2)]
index = 0

for i in range(rows, 0, -1):
    for j in range(i):
        print(letters[index], end=" ")
        index += 1
    print()
```

[</> Copy Code](#)

Output:

```
A B C D E
F G H I
J K L
M N
O
```

[</> Copy Code](#)

This code prepares a list of all required letters. The first row prints five letters, the next row prints four, and so on. Each time you finish a row, you move further along in the `letters` list.

## Full / Centered Alphabet Pyramids

These patterns arrange letters in a mirrored format. Each row expands from A to a certain letter, then reverses back to A on the same line.

For example, row one prints `A`, row two prints `A B A`, and row three might print `A B C B A`. Spaces before each row keep the structure aligned in the middle. This style demonstrates how to combine ascending and descending alphabets in one row.



Here is how the code works: It calculates the required letters for each row, then prints them forward and backwards with just the right number of leading spaces.

```
n = 5
for i in range(1, n + 1):
    # Print leading spaces for centering
    print(" " * (n - i), end="")

    # Print ascending letters from 'A' up to the current row
    for j in range(i):
        print(chr(65 + j), end=" ")
    # Print descending letters from one less than the current row back down to 'A'
    for j in range(i - 2, -1, -1):
        print(chr(65 + j), end=" ")
    print()
```

[</> Copy Code](#)

Output:

```
  A
 A B A
A B C B A
A B C D C B A
A B C D E D C B A
```

[</> Copy Code](#)

Each row prints  $(n - i)$  spaces, shifting the pyramid to the right. The first loop (`for j in range(i)`) builds letters from **A** onward, and the second loop (`for j in range(i - 2, -1, -1)`) rebuilds them in reverse order. This meets in the center to form a symmetrical pattern around the row's peak letter.

## Diamond & Hollow Diamond (Alphabets)

You can arrange letters just like a star-based diamond, but you rely on ASCII values to determine which alphabet goes in each position. A solid diamond prints letters in a growing-to-shrinking sequence, while a hollow diamond places letters only on the outer edges.

### 1. Solid Diamond



You split the shape into two loops. The first loop increases the letter count, and the second loop decreases it. Each letter comes from an ASCII calculation that depends on the current row and position in that row.

```
n = 4
# Upper half
for i in range(1, n + 1):
    # Leading spaces
    print(" " * (n - i), end="")
    # Print letters from A to the needed ASCII
    for j in range(2 * i - 1):
        print(chr(65 + j), end="")
    print()
# Lower half
for i in range(n - 1, 0, -1):
    print(" " * (n - i), end="")
    for j in range(2 * i - 1):
        print(chr(65 + j), end="")
    print()
```

[</> Copy Code](#)

Output:

```
A
ABC
ABCDE
ABCDEFG
ABCDE
ABC
A
```

[</> Copy Code](#)

The first loop counts from  $i = 1$  to  $n$ , growing the number of letters each time, while the second loop reverses that pattern. Spaces on the left keep the rows centered.

## 2. Hollow Diamond

This version only prints letters at the start and end of each row, leaving blanks in between. It checks whether you are on the boundary of the row or column before printing a letter.



```

n = 4
# Upper half
for i in range(1, n + 1):
    print(" " * (n - i), end="")
    for j in range(2 * i - 1):
        if j == 0 or j == (2 * i - 2):
            print(chr(65 + j), end="")
        else:
            print(" ", end="")
    print()
# Lower half
for i in range(n - 1, 0, -1):
    print(" " * (n - i), end="")
    for j in range(2 * i - 1):
        if j == 0 or j == (2 * i - 2):
            print(chr(65 + j), end="")
        else:
            print(" ", end="")
    print()

```

[</> Copy Code](#)

Output:

```

  A
 A C
A   E
A     G
A   E
  A C
   A

```

[</> Copy Code](#)

The condition `if j == 0 or j == (2 * i - 2)` ensures letters appear only at the edges of each row. The rest are replaced with spaces, which leaves the middle hollow.

## Equilateral Triangle & Hollow Squares (Alphabets)

These patterns extend the typical row-based approach by centering letters in a triangular shape or printing them only on the edges of a square. Each one relies on ASCII increments to generate alphabetical sequences.



## 1. Equilateral Triangle

Each row prints one additional letter. Leading spaces ensure the triangle is centered. You track ASCII values to figure out which letters go where.

```
n = 5
ascii_val = 65 # 'A'
for i in range(n):
    # Print spaces for centering
    print(" " * (n - i - 1), end="")
    # Print letters for this row
    for j in range(i + 1):
        print(chr(ascii_val), end=" ")
        ascii_val += 1
    print()
```

[</> Copy Code](#)

Output:

```
  A
 B C
D E F
G H I J
K L M N O
```

[</> Copy Code](#)

The code first uses  $(n - i - 1)$  spaces to center the letters. Then, it prints  $(i + 1)$  letters starting from 'A'. The variable `ascii_val` moves to the next letter each time, so the next row begins right where the previous row ended.

## 2. Hollow Alphabet Squares

Letters appear around the outer boundary, but you leave the inner portion blank. An if condition checks whether the current position is on the top row, bottom row, left column, or right column.

```
size = 5
for row in range(size):
    for col in range(size):
```



```

    if row == 0 or row == size - 1 or col == 0 or col == size - 1:
        print(chr(65 + col), end=" ")
    else:
        print(" ", end=" ")
    print()

```

[</> Copy Code](#)

Output:

```

A B C D E
A         E
A         E
A         E
A B C D E

```

[</> Copy Code](#)

For each row, the code checks if `row` or `col` is at the boundary. If yes, it prints the corresponding letter using `chr(65 + col)`. Otherwise, it prints a space to keep the center hollow.

## Word-Based Patterns

Instead of relying on ASCII values for single letters, these patterns use an entire word. Each row reveals one more character from the word in a stepwise fashion, which can be done by slicing or building a substring.

Here is how you might print partial slices of "Python" row by row:

```

word = "Python"
for i in range(1, len(word) + 1):
    print(word[:i])

```

[</> Copy Code](#)

Output:

```

P
Py
Pyt
Pyth

```





In this code:

- You start with the first character, then expand your slice (word[:i]) as i grows.
- The loop runs until you include the entire string in the final row

Also Read: [ord in Python - Unraveling Character to Integer Conversion](#)

## 4 Python Pattern Programs Using Recursion (With Code and Output)

Recursion provides a distinctive way to build pattern programs in Python. You might prefer to call the same function repeatedly to generate each row, whether you are heading from small to large shapes or the other way around. This approach can shorten your code but calls for careful attention to a base case. You also need to watch for how many stack frames you use when rows increase.

Here is where you might consider a recursive pattern:

- You want to build or display shapes from the smallest row to the largest row.
- You prefer a function-based layout that stops when it reaches a specific row.
- You find it more straightforward to visualize shapes through a series of smaller calls instead of nested loops.

Let's explore some Python pattern programs using recursion now.

### 1. Recursive Half Pyramid Using Stars

A half pyramid is a classic test of row-by-row logic. In the recursive version, each call handles one row of stars and then invokes the next call for the subsequent row.

Here is how this code works: you print the correct number of stars for the current



then call the same function for the next row until you reach zero rows.

```
def recursive_half_pyramid(rows):  
    if rows <= 0:  
        return  
    recursive_half_pyramid(rows - 1)  
    print("* " * rows)  
  
recursive_half_pyramid(4)
```

[</> Copy Code](#)

Output:

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

[</> Copy Code](#)

It starts at **rows = 4** and calls itself with **rows - 1** until **rows** is zero. Once the deepest call prints one star, the functions return up the call stack, printing more stars on each upward step.

## 2. Recursive Full Pyramid Using Stars

A full pyramid centers the stars. In a recursive approach, each call handles one level of the pyramid. The function prints spaces followed by stars and then calls itself to process the next row.

This code calculates how many spaces and stars to print based on the current row. When the row exceeds the desired number, recursion stops.

```
def print_space(space):  
    if space == 0:  
        return  
    print(" ", end="")  
    print_space(space - 1)  
  
def print_star(star):
```



```

    if star == 0:
        return
    print("*", end="")
    print_star(star - 1)

def recursive_full_pyramid(n, current=1):
    if current > n:
        return
    # Print spaces for centering
    print_space(n - current)
    # Print stars for the current row
    print_star(2 * current - 1)
    print()
    recursive_full_pyramid(n, current + 1)

recursive_full_pyramid(4)

```

[</> Copy Code](#)

Output:

```

    *
   ***
  *****
 *****

```

[</> Copy Code](#)

Each row prints  $(2 * \text{current} - 1)$  stars. A separate helper function prints the spaces first. After printing one row, it calls `recursive_full_pyramid` for the next row until the target count is reached.

### 3. Recursive Inverted Pyramid Using Stars

This shape starts wide and shrinks each time. A recursive call prints the current row first, then shifts to the next row with one less set of stars.

You print the full row of stars first, then call the function again with `rows - 1`, which eventually bottoms out at zero.

```

def recursive_inverted_pyramid(rows):
    if rows <= 0:
        return

```



```
print("* " * rows)
recursive_inverted_pyramid(rows - 1)

recursive_inverted_pyramid(4)
```

[</> Copy Code](#)

Output:

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

[</> Copy Code](#)

The code prints four stars, then three, then two, then one. After the last row, the base case (`rows <= 0`) stops any further recursion.

## 4. Recursive Number Pyramid

This variant handles numeric shapes. You print a row of increasing digits, then call the function to build the next row. A helper prints the numbers for the row in question.

```
def print_numbers(count, start=1):
    if start > count:
        return
    print(start, end=" ")
    print_numbers(count, start + 1)

def recursive_num_pyramid(rows):
    if rows == 0:
        return
    recursive_num_pyramid(rows - 1)
    print_numbers(rows)
    print()

recursive_num_pyramid(4)
```

[</> Copy Code](#)

Output:



```
1
1 2
1 2 3
1 2 3 4
```

[</> Copy Code](#)

The helper function `print_numbers` prints values from 1 up to `count`. Each call to `recursive_num_pyramid` prints the row for its level, but only after the lower levels have been printed, which creates the ascending effect.

Also Read: [Python Recursive Function Concept: Python Tutorial for Beginners](#)

## Recursion vs. Looping: Which is Better for Pattern Problems in Python?

Recursion is a valid alternative to for or while loops when you want to build patterns, yet each approach has its own strengths and weaknesses.

Below is a quick look at how they stack up against each other.

Aspect	Recursive Approach	Iterative Approach
Code Structure	Often looks cleaner for shapes (e.g., top-down pyramids). Might be shorter.	Typically straightforward loops that are easy to read and follow.
Memory Usage	Each recursive call adds a new stack frame, which can be a concern for large patterns.	Uses a constant amount of stack space, relying on loop counters instead.
Debugging	Can be trickier to track variable changes and call depth.	Easier to follow logic due to one level of execution flow.
Performance	Similar in time complexity, but overhead from repeated function calls might be higher.	Generally efficient, with minimal overhead per loop iteration.



Control Flow	Natural for building certain shapes from smaller subproblems upward.	Straight-line logic with nested loops that clearly define each row/column.
--------------	--	--

Also Read: [Iterator in Python: A Step by Step Guide](#)

## How Can upGrad Help You Learn Python?

With upGrad, you can access global standard education facilities right here in India. [upGrad offers Python courses](#) that come with certificates, making them an excellent opportunity if you're interested in data science and machine learning.

By enrolling in upGrad's Python courses, you can benefit from the knowledge and expertise of some of the best educators from around the world. These instructors understand the diverse challenges that Python programmers face and can provide guidance to help you navigate them effectively.

Here are some of the best data science and machine learning courses designed to meet your learning needs:

- [Professional Certificate Program in Business Analytics & Consulting in association with PwC India](#)
- [Advanced Certificate Program in Generative AI](#)
- [Post Graduate Certificate in Machine Learning and Deep Learning](#)
- [Post Graduate Certificate in Data Science & AI \(Executive\)](#)
- [Executive Program in Generative AI for Business Leaders](#)
- [Master of Science in Machine Learning & AI](#)
- [Master's Degree in Artificial Intelligence and Data Science](#)
- [DBA in Emerging Technologies with Specialization in Generative AI](#)

### Related Blogs:

- [7 Different Methods to Check Prime Numbers in Python](#)
- [Leap Year Program in Python](#)



- [Difference Between List and Tuple in Python](#)
- [Difference between List, Tuple, Set, and Dictionary in Python](#)
- [Operator Precedence in Python](#)

Unlock the power of data with our popular Data Science courses, designed to make you proficient in analytics, machine learning, and big data!

## Explore our Popular Data Science Courses

<a href="#">Executive Post Graduate Programme in Data Science from IITB</a>	<a href="#">Data Science Bootcamp with AI</a>	<a href="#">Master of Science in Data Science from LJMU</a>
<a href="#">Advanced Certificate Programme in Data Science from IITB</a>	<a href="#">Professional Certificate Program in Data Science and Business Analytics from University of Maryland</a>	<a href="#">Data Science Courses</a>

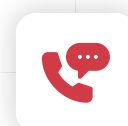
Elevate your career by learning essential Data Science skills such as statistical modeling, big data processing, predictive analytics, and SQL!

## Top Data Science Skills to Learn

<a href="#">Data Analysis Course</a>	<a href="#">Inferential Statistics Courses</a>
<a href="#">Hypothesis Testing Programs</a>	<a href="#">Logistic Regression Courses</a>
<a href="#">Linear Regression Courses</a>	<a href="#">Linear Algebra for Analysis</a>

Stay informed and inspired with our popular Data Science articles, offering expert insights, trends, and practical tips for aspiring data professionals!

## Read our popular Data Science Articles



<a href="#">Data Science Career Path: A Comprehensive Career Guide</a>	<a href="#">Data Science Career Growth: The Future of Work is here</a>	<a href="#">Why is Data Science Important? 8 Ways Data Science Brings Value to the Business</a>
<a href="#">Relevance of Data Science for Managers</a>	<a href="#">The Ultimate Data Science Cheat Sheet Every Data Scientists Should Have</a>	<a href="#">How to Become a Data Scientist</a>

## Frequently Asked Questions

1. How to learn printing pattern programs easily? +
2. How to print a name with a pattern in Python? +
3. How can I print a pattern using a while loop in Python? +
4. How can we print the star pattern in Python using for loop and if-else conditions? +
5. How to solve Python pattern problems? +
6. How to print 1 23 456 in Python? +
7. How to print an inverted pyramid in Python? +
8. How to make a diamond pattern in Python? +
9. How to reverse a number in Python? +
10. How to reverse a string in Python? +
11. How to make a circle in Python? +

Rohit Sharma

761 articles published

## Get Free Consultation



+91 ∨

Phone Number \*

Submit

By submitting, I accept the [T&C](#) and [Privacy Policy](#)

## Start Your Career in Data Science Today

↓ Career Repor ..



Top Resources

## Recommended Programs





UpGrad

## Business Analytics & Consulting with PWC India

[Placement assistance](#)

 Certification

 3 Months

[View Program](#)

[↓ Syllabus](#)

Liverpool John Moores University

## MS in Data Science

[Dual Credentials](#)

 Master's Degree

 17 Months

[View Program](#)

[↓ Syllabus](#)

## Suggested Blogs

DATA SCIENCE

### Top Python Programs to Create Pattern

By Pavan Vadapalli

30 Jan 2025 | 8 min read

DATA SCIENCE

### 15 Key Skills Every Business Analyst Needs In Order to Excel

By upGrad

15 May 2025 | 29 min read

DATA SCIENCE

### 4 Types of Data Science Explained

By Rohit Sharma

15 May 2025 | 12 min read

[Explore all >](#)

Building Careers of Tomorrow



 GET THE ANDROID APP

 GET THE IOS APP

UPGRAD  
SUPPORT

MBA

DATA SCIENCE & ANALYTICS

DOCTORATE



SOFTWARE & TECH

AI & ML

MARKETING

MANAGEMENT

LAW

JOB LINKED

BOOTCAMPS

STUDY ABROAD

FOR COLLEGE STUDENTS

SUPPLY CHAIN MANAGEMENT

ARCHIVED PROGRAMS

© 2015-2025 upGrad Education Private Limited. All rights reserved

