

**“Everybody in this country
should learn to program
a computer... because it
teaches you
HOW TO Think.”**

— Steve Jobs



**Please apply ([Ctrl + Click](#)) on the above YouTube icon
to access the [Python crash course for beginners](#).**

Python

Wise Head

Junior

By

M O H M A D | Y A K U B

Technical Review: Sadique S. Naikwadi
Proof Reader: Mrs. Shabana Mursal

Editing
Mohmad Yakub

Copyright @ 2021 Mohmad Yakub

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the author's express written permission except for the use of brief quotations in a book review.

The author has made every effort in the preparation of this book to ensure the accuracy of the information. However, information in this book is sold without warranty, either expressed or implied. The author will not be held responsible for any damages caused or alleged to be caused either directly or indirectly by this book.

Contact Information: I will appreciate it if any queries raised or corrections need to be applied. I do appreciate the honest feedback. Readers of this book can reach me at the following mentioned email address.

Email: yakublancer@gmail.com

I dedicate this book as a medium of hope for those

- Who is unaware of any approach to craft any programming logic?
- Who had a hard time learning programming?
- Who had some experience in programming and have no confidence.
- Who carries the False notion that coding is only for super-smart people.
- Who are looking for 1st solid move to become a self-taught programmer.
- Who are victims of discouragement comments similar to the following;
 - Actually, you aren't interested.
 - You lack patience and determination.
 - Your IQ is well below average.

Preface

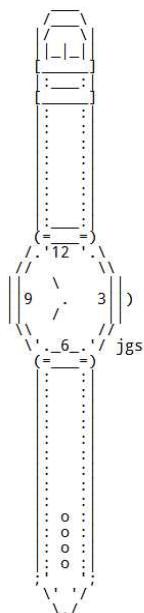
EVEN THE GREATEST WAS ONCE A BEGINNER.
DON'T BE AFRAID TO TAKE THAT FIRST STEP.

- Muhammad Ali

Programming is a kind of mental aerobics that teaches us to keep focus and nourishes good patience, discipline, and forecasting. These are some main qualities of an achiever.

Programming is for everybody and not just an academic exercise.

When you visualize and process the information, the learning will become active. When you apply the information naturally, there is no need for memorization. Programming is not about memorizing the programming logic. It is about applying techniques to solve problems. In order for one to apply an approach correctly, one should be able to visualize it correctly first. The best introduction to programming is to learn the logic of programming first. This is a means to build confidence and interest in programming. A programmer should be able to solve the following logic-based problem. Based on different inputs, a person should be able to generate a wristwatch of customized sizes. Because of a lack of wisdom in programming, most beginners cannot accomplish this. The excellent point is that learning logic-building skills does not require much technical baggage.



Many beginners judge that if they can learn syntax or explore some common built-in features of the programming language, they will learn how to solve problems. That is not the truth. Learning traffic rules and signboards by themselves won't help someone become proficient in driving. You'll have to take certain steps in the right direction. Syntax-oriented learning about programming language just gets you started. Syntax-oriented learning for a programming language is suitable for experienced programmers. Experienced programmers already possess that wisdom, so they only need to discover the language's rules and features to use. This does not apply to beginners.

Developing the fertile ground for visualizing programming logic should be the focus of an absolute beginner. Unfortunately, this approach is foreign to most beginners and even to the teaching group as well. The exercises should be such that they will assist novices in learning or improving visualization of programming logic. Two major reasons why the approach mentioned in this book works for the novice or not so confident programmers;

A) Visual learning is a great way to improve our learning skills. I had experimented using the mentioned approach with many students and found this to be an incredibly interesting way to learn to program quickly. These teachings are outlined in the book to help spark interest in programming. It is a proven practice built on common sense and basic mathematics; thus, practically anyone can learn programming without getting bogged down with technicalities.

Without sufficient logic-building skills, the efforts to learn advanced programming topics will be like talking to Gorilla; it will not be productive.

B) In addition, the content receives further refinement after interacting with beginner programmers on **Quora.com**. My contribution was recognized, and I believe that it will help a lot of beginners.



You wrote 3.2k answers - and people noticed

We loved your answers - and so did others! Your contributions helped people find the knowledge they were looking for.

324.6k

upvotes

605

shares

Please check the following post to get more details.

<https://www.quora.com/How-can-I-improve-my-programming-skills-18/answers/140798300>

The learning process is always better when it is fun. They can treat this book as a supplementary text for both the students and the teachers, trainers, and whoever wants to pursue coding as a hobby. In addition, this resource can be useful to parents who want to teach their children coding. Parents, please save your hard-earned money and start coaching your children right away. Gift them this platform so that further learning for them will be a breeze.

Happy learning !!!

Table of Contents

<i>Preface</i>	5
PART 1: BASICS OF PROGRAMMING LOGIC – Quick Revision.....	12
Introduction.....	13
The Right Attitude.....	14
The Clarity Principles.....	15
Any pre-requisite?.....	16
About the book.....	16
How to make the best use of this book?.....	17
Python Project.....	18
First Python Program.....	20
Welcome.py.....	20
Console Output.....	20
MultiplePrintMethod.py.....	20
DefaultPrint.py.....	21
PrintInSameLine.py.....	21
PrintInSameLine1.py.....	22
PrintInSameLine2.py.....	22
Storing different data types.....	23
Variable assignment.....	23
VariableInfo.py.....	24
Input From Keyboard.....	25
PersonDetails.py.....	25
Operators.....	28
Arithmetic Operators.....	28
ArithmeticOperations.py.....	29
Comparison Operators.....	30
ComparisonOperators.py.....	31
Logical Operators.....	33
Decision Making.....	34
a) if-statement.....	34
StatementIf.py.....	35
StatementIf1.py.....	35
b) if-else statement.....	36
StatementIfElse.py.....	37
StatementIfElse.py.....	37
c) if-elif-else statement.....	38
StatementIfElseIf.py.....	39
StatementIfElseIf.py.....	40
StatementIfElseIf.py.....	41
LadderIfElseToSwitchCase.py.....	43
About for-loop.....	44
SequenceWithoutForLoop.py.....	44
SequenceWithForLoop.py.....	46
ForLoopWithRangeTwoInputs.py.....	46
ForLoopWithRangeOneInput.py.....	46
ForLoopDecrement.py.....	47
For-loop with break and continue keywords.....	48

ForLoopContinueNext.py.....	48
Nested for-loop/Inner for-loop.....	49
NestedForLoop.py.....	50
Nested Loop: Beginner Programmer's Best Friend.....	50
Why are nested loops such an important deal?.....	50
ForLoopEveryRowAllColumns.py.....	51
ForLoopEveryRowAllColumnsHorizontally.py.....	52
ForLoopCoordinateRepresentation.py.....	52
RowColumnStars.py.....	53
JustLoopIt: Write a separate program for each of the given outputs.....	54
Art of Applying Logical Conditions.....	55
Convert Logical Conditions to Simple Picture.....	57
Convert Simple Picture to Logical Conditions.....	60
VerifyAlphabetA.py.....	62
VerifyAlphabetB.py.....	65
VerifyAlphabetC.py.....	67
VerifyAlphabetD.py.....	69
VerifyAlphabetE.py.....	71
VerifyAlphabetF.py.....	73
VerifyAlphabetG.py.....	75
VerifyAlphabetK.py.....	80
VerifyAlphabetM.py.....	84
Learning by experiments and guesswork.....	90
Logic trace table: Brain behind programming logic.....	93
PART:2 PROGRAMMING IN ACTION.....	94
Cursor Movement.....	97
Pattern 1A.....	98
Analysis of output.....	98
Pattern1A1If.py.....	100
PatternA1NestedLoop.py.....	103
Pattern1A_1.py.....	111
Pattern1A_1.py.....	111
Pattern1B.....	112
Analysis of output.....	112
PatternB1If.py.....	114
PatternB1NestedLoop.py.....	116
Approach to solution.....	118
PatternB1.py.....	123
Alternative solution2.....	124
PatternB2.py.....	125
Alternative solution3.....	126
PatternB3.py.....	127
Pattern 1C.....	129
Analysis of output.....	129
Approach to solution.....	132
PatternC1.py.....	134
Alternative solution2.....	135
PatternC2.py.....	137
Alternative solution3.....	138
PatternC3.py.....	140

Pattern1D.....	141
Analysis of output.....	141
PatternD1.py.....	146
Pattern1E.....	148
PatternE1.py.....	155
Alternative solution2.....	156
PatternE2.py.....	159
Pattern 1F.....	160
PatternF1.py.....	165
Alternative solution2.....	166
Pattern 1G.....	167
Approach to solution.....	171
PatternG1.py.....	176
PatternG2.py.....	177
Alternative solution2.....	178
PatternG3.py.....	179
Pattern1H.....	180
Approach to solution1.....	183
PatternH1.py.....	188
PatternH1_2.py.....	189
Alternative solution2.....	190
PatternH2_1.py.....	193
PatternH2_2.py.....	196
Alternative solution3.....	197
PatternH3.py.....	200
PatternAA.....	202
PatternCC.....	202
PatternII.....	203
NumberPattern 1A.....	206
Analysis of output.....	207
NumberPattern1A.py.....	210
NumberPattern 1B.....	211
Analysis of output.....	212
Observations.....	212
NumberPattern1B.py.....	216
Alternative solution.....	217
NumberPattern1B_2.py.....	218
NumberPattern 1C.....	219
Analysis of output.....	220
NumberPattern1C_1.py.....	222
NumberPattern1C_2.py.....	223
NumberPattern1C_3.py.....	224
NumberPattern2C.py.....	226
NumberPattern 1D.....	227
Analysis of output.....	228
Observations.....	228
NumberPattern1D.py.....	231
Alternative solution.....	231
NumberPattern2D.py.....	231

Alternative solution.....	232
NumberPattern3D.py.....	234
NumberPattern 1E.....	235
Analysis of output.....	236
NumberPattern1E.py.....	239
NumberPattern 1F.....	240
Analysis of output.....	241
NumberPattern1F.py.....	244
NumberPattern AA.....	245
Analysis of output.....	246
Observations.....	246
NumberPatternAA.py.....	248
NumberPattern BB.....	249
Analysis of output.....	250
Observations.....	250
NumberPatternBB.py.....	253
Alternative solution.....	253
NumberPatternBB2.py.....	254
NumberPattern CC.....	255
Analysis of output.....	256
NumberPatternCC1.py.....	258
NumberPatternCC2.py.....	259
NumberPatternCC3.py.....	261
NumberPattern DD.....	262
Analysis of output.....	263
NumberPatternDD1.py.....	265
NumberPatternDD2.py.....	266
NumberPatternDD3.py.....	267
NumberPattern EE.....	268
Analysis of output.....	269
NumberPatternEE.py.....	272
NumberPattern FF.....	273
Analysis of output.....	274
Alternate solution.....	276
NumberPatternFF.py.....	280
If-else Shortcut.....	281
When to apply if-else shortcut.....	281
NumberPatternFF2.py.....	282
NumberPattern GG.....	283
Writing our own functions.....	286
NumberPattern HH.....	289
Analysis of output.....	290
NumberPattern AAA.....	294
Analysis of output.....	296
Observations.....	296
NumberPatternAAA.py.....	298
NumberPattern BBB.....	299
Observations.....	300

NumberPatternBBB1.py.....	303
Alternative solution.....	303
NumberPatternBBB2.py.....	304
NumberPattern CCC.....	304
Analysis of output.....	306
NumberPatternCCC1.py.....	308
Alternative solution.....	309
NumberPattern DDD.....	313
Analysis of output.....	314
Observations.....	314
NumberPatternEEE.....	319
Analysis of output.....	320
String concatenation.....	325
Substrings or Slicing.....	326
Patterns by Single Looping.....	329
Accessing each string's character.....	333
Appendix-A: Installation of Python and project set-up.....	340
References:.....	340
Acknowledgments.....	341

PART 1: BASICS OF PROGRAMMING LOGIC

Introduction

Without a doubt, every subject has some key fundamental basis. If key concepts of any subject are truly internalized, then the rest of the learning will be productive.

Many novice programmers and many frustrated programmers do ask similar kind of questions which are as follows

“How to learn programming?”

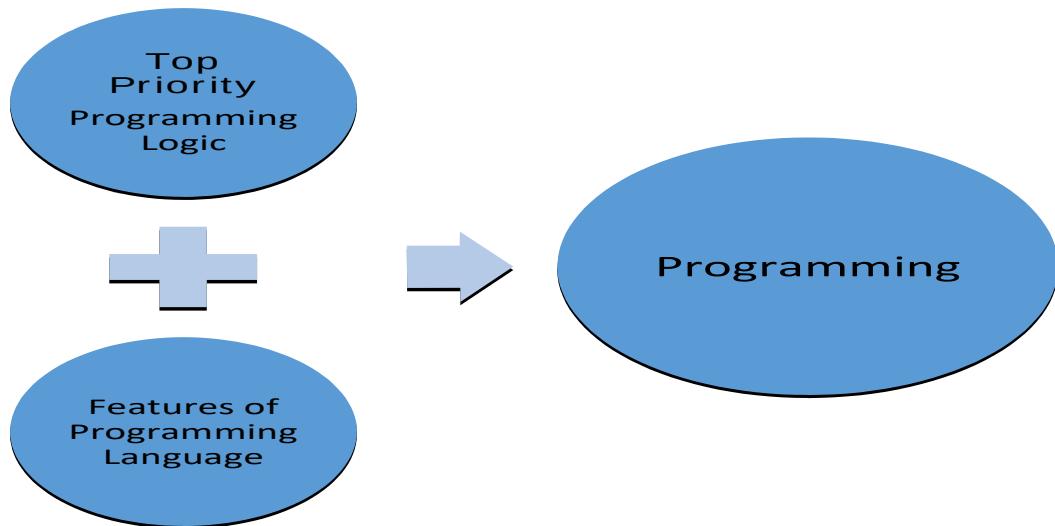
“How to develop logic-building skills?”

“How to learn coding?”

“How to improve programming logic?”

Programming is an applied field. In my opinion, if a programmer who is weak in programming logic is not even a programmer. We know that complaining is the easiest job and if someone is an expert, they haven't achieved anything big. So, whenever there is a problem, we need to think, “**what could be the easiest possible way out?**”

In order to become confident in any programming language, the first step should be to gain some level of proficiency in self-hack programming logic. This technical manual is entirely dedicated to non-programmer/beginner or intermediate students who have reached a dead end while trying to become confident in programming. Additionally, if you are among those who have limited time to learn programming, this is the guide that can serve you well. **So, the rule of the thumb is that to learn programming language fast and properly, first learn the tactics of programming logic sufficiently.**



The most common mistake that many non-programmers/absolute beginners or trainers make is their focus on grammar/syntax and feature(s) of the programming language. Such a shallow outlook about programming language doesn't help beginners much. Computer science is an applied field to solve practical problems. Therefore, it is a foremost and essential need that a beginner should be trained sufficiently in programming logic skills as the first priority. How appropriate physical exercises and healthy food tones the body muscles; similar is developing logical programming skills. It does need the right kind of logic-based programming exercises to groom visualization of programming logic.

It also happens that a beginner may know almost all the programming language concepts but still feels something is missing. That missing element is due to insufficient logic-building capabilities. Without a comfortable level of logic-building skills, the beginner becomes stagnated, and, on top of that, they face a lot of confusion while developing programs.

The Right Attitude

Failure taught me things about myself that I could have learned no other way.

- J.K. Rowling

By default, children are not afraid to be wrong, and so they are creative. It's well-said that **hiding mistakes are a mistake of higher order**. So, it is always good to notice our mistakes as early as possible so that it shouldn't give a hard blow in the future during the much-needed time.

Experimenting and learning from mistakes work like a body detoxification program. Initially, any detoxification program doesn't taste nice, but still, a person goes for it because they will love to see how the body will respond after 2-3 weeks. Anything new that we do take some practice. Everything valuable has a cost associated with it. Failures are one such type of cost that is to be paid to learn and to remember lessons.

Getting accurate output after executing a program is just a by-product. So, the total learning will be failures plus the success part. So we need to think in this way; more initial failures will help us reduce the chances of failures in the future instead of thinking that we are not good learners. When a learner knowingly or unknowingly applies mismatching logical conditions, they may encounter unexpected output. This, in turn, creates curiosity about which specific logical condition is impacting the output differently. It's easy for a brain to remember surprises. Moreover, a picture helps the brain remember more than the flowing texts, let it be any fact or concept. So, it's much useful if a learner experiments with a weird or with some typical logical conditions and tries to find the reasons behind it. This will work like a self-motivating exercise, and it simply creates the urge to learn more.

We all have hidden potential, and it is up to us to explore ourselves. You are the only person who can explore your hidden potential by taking challenges by experimenting with your capabilities. However, unless and until you take this journey of challenges, you will not know the **actual YOU**, or you will remain rather somebody else, a common class. ***So you, I, and everybody had to do this exercise of a treasure hunt within ourselves.***

It is all about the game of attitude and applying our actions in trying for excellence. We need to bounce back repetitively with another honest attempt. One of the main aims of this guide is to ignite the thought process and develop a natural tendency to solve problems by touching bare minimal technical details. We know that attitude measures altitude. The right intention gives a lot of power to manage even tough tasks with confidence. It is all about the perspective, the way we think, and approach the situation accordingly.

The Clarity Principles

Let's help ourselves by putting these principles as our reminders. We must keep reminding ourselves, the important key values that help to cleanse our subconscious mind. Definitely, there are benefits in reminders, so give reminders to others or our-self.

You can't manage what you can't measure.

- Peter Drucker

Science is simply common sense at its best, that is, rigidly accurate in observation, and merciless to fallacy in logic.

- Thomas Huxley

Most of man's problems upon this planet, in the long history of the race, have been met and solved either partially or as a whole by experiment based on common sense and carried out with courage.

- Frances Perkins

I read, I study, I examine, I listen, I think, and out of all this I try to form an idea into which I put as much common sense as I can.

- Marquis de Lafayette

Common sense is calculation applied to life.

- Henri-Frédéric Amiel

The most important aspect of calculation or measurement is that it gives clarity. We will try to tap this "**clarity**" factor to get the hidden programming logic right using the **logic trace table**.

All learning covered by this guide majorly revolves around the concept of a **logic trace table**. In the section **Programming Patterns**, its practice is **dealt with in greater detail**.

We can't expect quality software without quality programs. Good programming logic is one of the essential qualities to write error-free programs and quality software.

Any pre-requisite?

The intention is to give an honest try to learn programming. No prior experience in a programming language is required. If you have any, it is good, and this is optional as we are not dealing with much of the syntactical technical matter. We only need very minimal language features to practice programming exercises to cultivate interest and confidence in programming. We are majorly dealing with two main technical concepts.

a) **Conditional statements (If-statement, if-else statement)**

b) **Looping construct— just *for-loop*.**

Programming syntax of conditional statements and looping of many programming languages have many things in common. By learning from this book, anyone can easily experiment in other programming languages of their interest. The major emphasis is on learning the approach, not the syntax. One of the common qualities of highly successful people is as soon as they acquire information, their speed of implementation is very fast. This quality helps them to stay ahead in the competition. **The best thing to compete with anyone is to compete with ourselves and always try to break our own record.**

About the book

All the material presented here is only meant for action, a practical to-do session. We are not covering as a language but programming logic as a subject using programming language's syntax. This book is not a product of guesswork. This book is the outcome of experiments conducted among college students to improve their logic-building skills or to learn programming from scratch but in a quick way.

The learner has to practice the approach mentioned in the book for logic-building skills related to 2-dimensional simple picture-based exercises. Practice exercises arranged in an increasingly challenging way motivates you to refine programming skills continuously in a step-by-step manner. The solved exercises give a complete idea, and unsolved problems urge applying the logical concept. So, it is going to be **YOU versus YOU** in an interesting way.

Two major requirements expected from every learner are as follows:

a) Intention that you want to learn programming

b) You need to spend some time consistently, and it depends on how quickly you want to become a confident programmer. **Don't break the learning pattern for at least 3 months; spend at least a minimum of 20 – 30 minutes daily.** Just follow the mentioned course of action step by step; you will surely get a substantial grip on the programming.

It's up to you afterward to explore new horizons in programming or software development. I can only promise an excellent start in a short time regarding your journey of learning programming logic. **So, let's start the journey of learning.**

How to make the best use of this book.?

- Start a topic and learn till you reach a point where you are ready to analyze a solved example. Concentrate on that example till you get a feel that you have understood everything about the program.

Then try to implement that solved example without referring to it. Make a strong will that once you start implementing an already studied program, you will not refer back to that particular solved program. You can refer to the theoretical explanation but not the program. There are a lot of chances that you may get many errors syntactically or logically as well. You can refer theoretical explanation about the concept in that situation but do not allow yourself to see the solution. Try your best to resolve issues by yourself, referring to theoretical explanation.

- Please maintain an Excel or spreadsheet file to note down the errors you encountered in a program. When you can resolve the errors, please note down the reasons you have adopted that approach. Also note down, your initial and final observations when you had resolved a particular logical error. The study of such gap analysis helps us to know how to think in the right direction.
- Every new day when you want to start your learning, please have a quick look at your error file that you were maintaining every time you encountered any error(s). Referring back to those errors while learning every new day will help you do better next time, and it will also help you quickly revise the concept. We should maintain this practice to avoid common logical errors, so those errors should not become part of our habit. So, it is good to correct those loopholes in the beginning since breaking a habit consumes sufficient effort. What we practice will become our habit, whether good or bad. Here, we are trying to **convert failures into lessons**. It is more rewarding when we improve from mistakes as early as possible to invest more time learning other important subjects in our lives.
- If at least two times, you can solve all the solved examples from the book back to back without referring to the solutions, you will be in a better position to write the program effectively.
- Hints are there for unsolved problems in the explanations of solved exercises. If you cannot solve unsolved problems, attempt 2-3 attempts and move forward to complete the book. Come back again and retry the remaining ones.
- Students should be able to solve all the solved exercises without referring to the solutions, at least two times back-to-back. The same applies to unsolved problems too. It makes learners flexible in building a program's logic, builds confidence, and inspires confidence in tackling more advanced programming subjects like data structures and design patterns.

Python Project

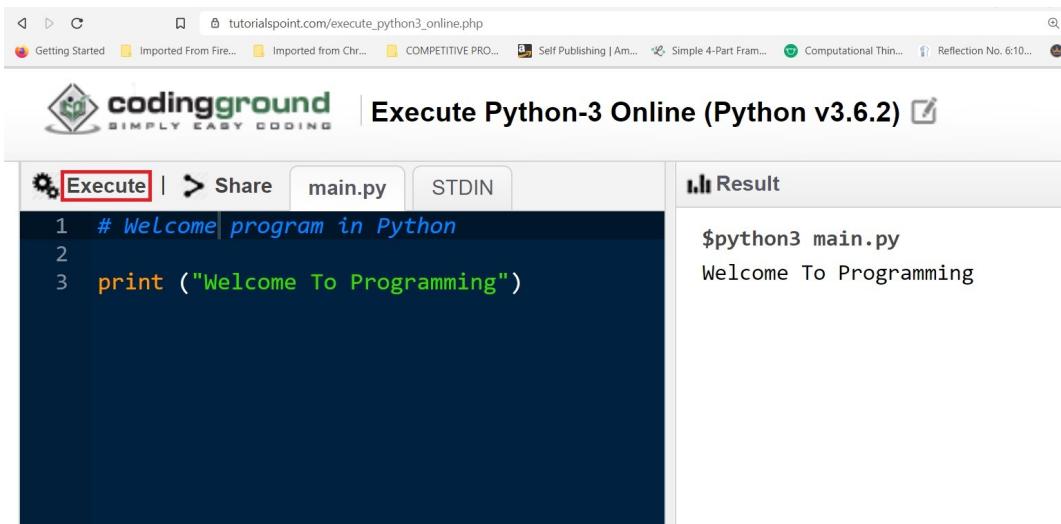
Please refer to **Appendix-A** at the end of this book.

Appendix-A: Installation of IDE and project set-up

It shows a step-by-step procedure to create a **Python project** using IDE. It also shows screen-shots of how to **create and execute a Python program**.

Alternatively, you can use any free online editor, for example,
https://www.tutorialspoint.com/execute_python3_online.php

Click the **Execute** button to run the program. By default, the **main.py** tab will be opened to write the program. A separate Result window is shown to display the output of the program.



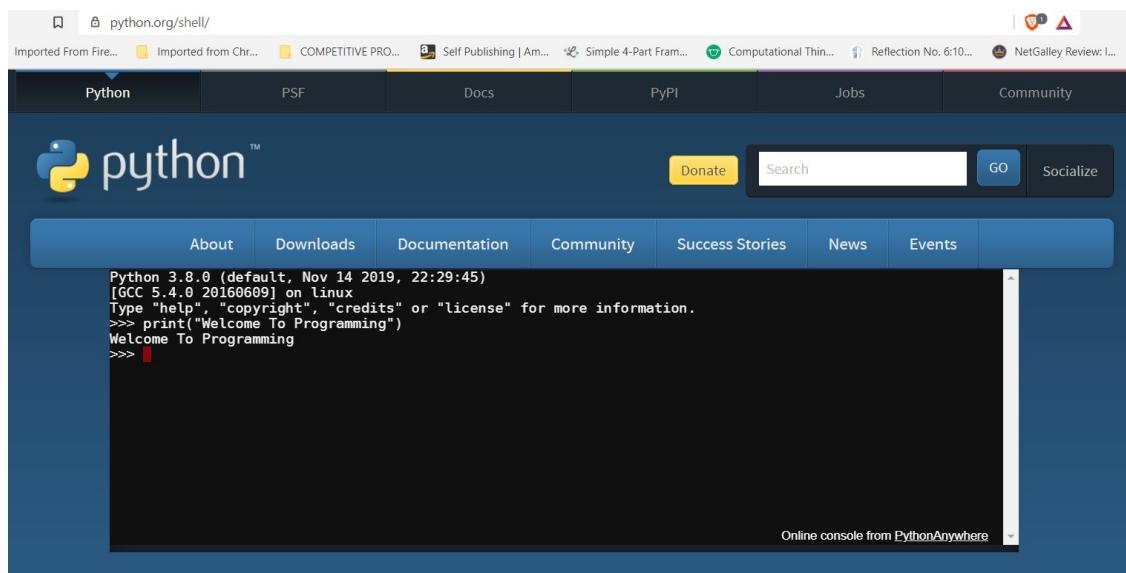
The screenshot shows the CodingGround online Python executor interface. The URL in the browser bar is https://www.tutorialspoint.com/execute_python3_online.php. The page title is "Execute Python-3 Online (Python v3.6.2)". The interface has two main sections: a code editor on the left and a result viewer on the right. In the code editor, the "main.py" tab is selected, showing the following Python code:

```
1 # Welcome program in Python
2
3 print ("Welcome To Programming")
```

The "Execute" button in the toolbar is highlighted with a red box. In the result viewer, the output of the executed code is displayed:

```
$python3 main.py
Welcome To Programming
```

<https://www.python.org/shell/>



After completing program writing, **press 'enter'** on the keyboard to see the program's output.

We should save each program in its own file for later reference. Every file has a name and extension.

The extension of the program file is **".py."** Suppose the name of our first program is Welcome, which will output a welcome message.

We should name the programming file with a meaningful name as per the purpose. So, we should create a **Welcome.py** file.

First Python Program

It is a good gesture to introduce programming to someone with a greeting message, and it is not compulsory. Most often, first-time learners try to print “Hello World” or their own name. The following `Welcome.py` python program outputs the text `Welcome to Programming`.

```
Welcome.py
# Print welcome message.

print("Welcome To Programming")
print('Welcome To Programming')
```

Output

```
Welcome To Programming
Welcome To Programming
```

To print textual content, we are required to put in `print()` method within double quotes(“**running text**”) or single quotes(‘**running text**’). Methods are ready-made facilities provided by a programming language. We need to learn how to apply these methods in what appropriate situation. It is just like how we refer dictionary of a particular language and use it to construct a sentence. Similarly, we need to use these methods in our programming statements. The step of writing effective, logical programming statements is one of the essentials of the writing program and is commonly known as logic building skills. Logic-building skills are must have to gain confidence and interest in programming.

Comments are for our understanding/documentation purpose and later reference only. It is not a part of our programming statements. The environment which executes a program ignores it.

The multi-line comments are given by `#` for each line.

The single-line comment is given by `#` for single line.

Console Output

We can start learning programming logic simply by repeatedly printing small text horizontally in the same line or vertically in every line. We can use method `print()` to output results in horizontal or vertical ways. Let’s try to learn how method `print()` behaves in different situations.

Exercise 1. Printing the text “`Welcome To Programming`” using `print("text")` method.

```
MultiplePrintMethod.py
# Print welcome message
print("Welcome ")
print("To ")
print("Programming")
```

Output

```
Welcome
To
Programming
```

We are not expecting a multi-line welcome message. We want a welcome message to be printed in a single line. It also suggests that the method `print("any text")`, every time chooses a new line to print “`any text.`” We need to use attribute `end` of the method `print("text", end="\n")`.

The attribute **end** forces it to end the printing text in a specific way. If we don't mention attribute **end** in a method `print("text")`, then it forces it to move to the new line after printing the specified `"text"` in the display screen. This is the default behavior of method `print("text")`.

Exercise 2. Learning the default behavior of method `print("text")`.

```
DefaultPrint.py
# default usage of method print without using attribute 'end='
print("Welcome")
print("To")
print("Programming")

print()
print()
print()

# use of attribute 'end=' to end the text with a new line
print("Welcome", end="\n")
print("To", end="\n")
print("Programming", end="\n")
```

Output

Line No.1	Welcome
Line No.2	To
Line No.3	Programming
Line No.4	
Line No.5	
Line No.6	
Line No.7	Welcome
Line No.8	To
Line No.9	Programming

Line No.4, Line No.5, and Line No.6 are blank lines due to method `print()`. This demonstrates the default behavior of attribute `end="\n,"` which moves the control to the new line. It suggests that next time the printing of text will start with this new line.

To print the text in the same line using multiple `print("text")` methods, we are required to change this default behavior using the attribute `end`.

Exercise 3. Printing the welcome message `"Welcome To Programming"` with an empty line.

```
PrintInSameLine.py
#attribute end="" to end the text with an empty line
print("Welcome", end="")
print("To", end="")
print("Programming", end="")
```

Output

WelcomeToProgramming

The welcome message was printed with no space between words. Let's try to add space in between.

Exercise 3.1. Printing the welcome message "Welcome To Programming" in the same line.

```
PrintInSameLine1.py
# use of attribute 'end=' to end the text with an empty line or space
print("Welcome", end="")# text ends with empty line
print("", end="")# empty text ends with a space
print("To", end="")# text ends with empty line
print("", end="")# empty text ends with a space
print("Programming", end="")# text ends with empty line
```

Output

```
Welcome To Programming
```

We can combine printing text with space within a single **print()** method.

Exercise 3.2. Printing the welcome message "Welcome To Programming" in the same line.

```
PrintInSameLine2.py
# use of attribute 'end=' to end the text with space
print("Welcome", end="")
print("To", end="")
print("Programming", end=".")# text ends with full stop(.)
```

Output

```
Welcome To Programming.
```

We can summarize how we used method **print()** and its attribute **end** in different ways.

Method	Purpose
print("text")	Prints the text and moves the control to the new line.
print("text", end="")	Prints the text in the same line.
print("", end="")	Prints a space in the same line.
print()	Moves the control to the next line.

Given programming statements means the same. It prints the text and moves the control to the next line.

```
print("text")      print("text", end="\n")      print("text", end="")
                  print()
```

We will learn only the required minimum technical knowledge in order to sharpen our intelligence in programming logic. We are targeting to dive deep into the art of building programming logic rather than going deep into the basic grammar/syntax of programming language. Let's check out some basic building blocks of programming logic. We are mainly targeting the following basics. Focusing properly on the given mentioned few only basics are enough to promote interest and build confidence in coding.

a) variables
b) logical operators

c) conditional statements
d) Nested loops

Storing different data types

The intelligence of software is due to a programming language(s). The processing of different data types in some form is something very core to almost every software. The facility to manipulate **different types of data correctly** is a necessary feature of any programming language—the examples of commonly used different data types.

General data type	Value examples
Integer	23,-4, 8908
Real Numbers	23.45, 3.1456,-45.27
Truth Values	True, False
text	"Just do it", "impossible"
Single Character	A, a,@,\$,%,*, 9

Don't we use a different variety of data in our daily life? We actually do, for example

Equivalent variable assignment	Filled in personal details
name ="Guido Van Rossum" age = 63 weightInKgs = 53.6 personalLoan = False gender = 'M'	Name : Guido Van Rossum Age: 63 Weight(In Kgs): 53.6 Personal loan (True/False): False Gender : M

We are sometimes required to fill correctly personal details in some way or another way. Few examples are, like school or college admission process, flight booking, or any online purchasing. We can easily distinguish different data types, whether it's a type like numeric, textual, or any other kind. For example, the data value 25 could be age, price, and many more possibilities. The name of the variable should be appropriately chosen. The variable's chosen name helps us know the meaning of the stored value contained in it.

How to store data/value in a variable?

Yes, it is made possible through the facility of variable assignment using assignment operator (=).

Variable assignment

The value of a variable might be over-written during the execution of a program and which is why it's called a variable. Any real-time situation of manually filling data can be simulated using a programming language feature known as a variable assignment. A variable assignment is a simple act of storing an appropriate value in an appropriately chosen variable. We can assign the value to the variable using the assignment operator (=). **We can store or assign only one data at a time.**

The contents of variables can be over-written programmatically during the course of running or executing the program. The format for variable assignment is as follows:

variable_name = value # assign only one data at a time

The variable name doesn't start with a number. Hence, a variable name like **4Age is invalid**

4Age = 20 # invalid variable name.

The name of a variable represents a memory location. It isn't easy to do programming using the name of memory locations. Memory address location resembles not less than any secret key. The name of a variable that we choose should be a user-friendly name representing a specific memory location allocated to store data. Python provides the inbuilt methods -**id()** and **type()** to know the variable's memory location and data type.

Example4. Print basic information of a variable.

```
VariableInfo.py
# variable assignment of different data types
#+ can be used to join different texts/strings

name = "Guido " +"Van " +"Rossum"
age = 63
weightInKgs = 53.6
personalloan = False
gender = 'M'

# output the content of variables
print("Memory location =", id(name),end="|")
print("Data Type =", type(name),end="|")
print("name =",name)

print("Memory location =", id(age),end="|")
print("Data Type =", type(age),end="|")
print("age =",age)

print("Memory location =", id(weightInKgs),end="|")
print("Data Type =", type(weightInKgs),end="|")
print("weightInKgs =",weightInKgs)

print("Memory location =", id(personalLoan),end="|")
print("Data Type =", type(personalLoan),end="|")
print("personalloan =",personalLoan)

print("Memory location =", id(gender),end="|")
print("Data Type =", type(gender),end="|")
print("gender =",gender)
```

Output

```
Memory location = 140646950853848 | Data Type =<class 'str'>| name = Guido Van Rossum
Memory location = 140646971403136 | Data Type =<class 'int'>| age = 63
Memory location = 140646951903496 | Data Type =<class 'float'>| weightInKgs = 53.6
Memory location = 140646971214752 | Data Type =<class 'bool'>| personalLoan = False
Memory location = 140646950860312 | Data Type =<class 'str'>| gender = M
```

Please revise these points regarding variables before you start writing programs.

- a. Variable stores value or data.
- b. Variable stores only a single value at a time.
- c. Assignment operator (=) is used to overwrite the already stored value of a variable.
- d. The name of a variable should be meaningful, and it refers to the complicated address in a memory.

The space in memory location can be expressed in terms of byte units. So, the size of primitive data types can be expressed in terms of bytes to store the value.(Note: 1 byte = 8 bits.)

What is there in the variable's name?

We know that in geographical maps, the locations can be represented as a combination of longitude & latitude. Think for a moment if our city, town, street places are replaced by longitude and latitude values, how much it will become difficult to use it in our normal communication. So, the choice of a variable should be a meaningful name and must indicate some purpose associate with it. It is not uncommon to find such programmers who have this special **SLVNS** sickness.

What is SLVNS sickness?

SLVNS stands for **Single Letter Variable Name Syndrome**. Many programmers have this special habit of creating suspense through the name of a variable. They allocate just a single letter for a variable's name in writing programs. When such a program becomes lengthy, and someone else has to fix a bug/error for that programming code, it will surely turn out to be an irritating experience. Even **SLVNS** programmers may find it difficult to get his/her own programming logic quickly on revisiting after a gap of time. If you find such programmers, please request them to change this habit; it is an offense. Please prevent yourself from this sickness and **be patient enough to write meaningful variable names**.

Input From Keyboard

In the previous example (**Example4**), the person's information assigned to the respective variables is fixed. We can make it interactive so that the details can be supplied directly from the keyboard. To supply data directly from the keyboard, we need to use Python's inbuilt function **input()**. It will be more positive if we input a user-friendly message while using method **input("Input message")**.

Example5. Print person's details from the keyboard.

```
PersonDetails.py
# input person details
name = input("Enter Name:")
age = input("Enter Age:")
Weight = input("Enter Weight:")

# output
print()# Print blank line. Move to the next line
print(" Person Details")
print("=====")
print("Name :" + name)
print("Age :" + age)
print("Weight :" + Weight)
```

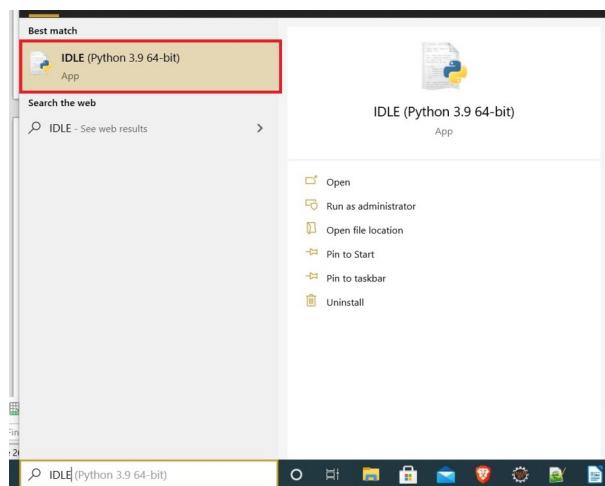
Output

```
Enter Name:jacob
Enter Age:45
Enter Weight:93

 Person Details
=====
Name: jacob
Age: 45
Weight: 93
```

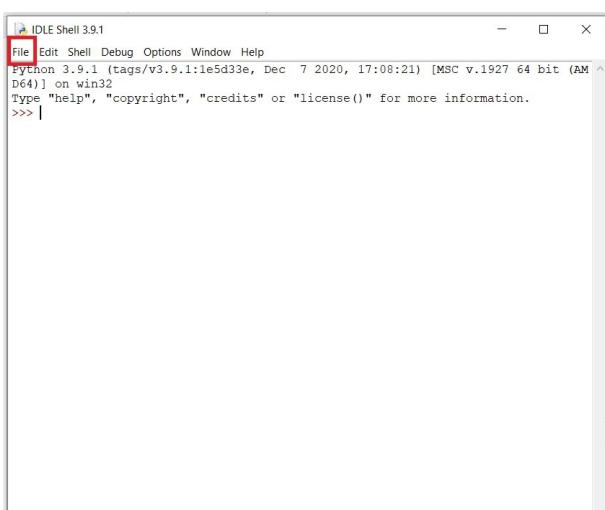
How to execute a .py file using IDLE Shell?

1.# Invoke IDLE.

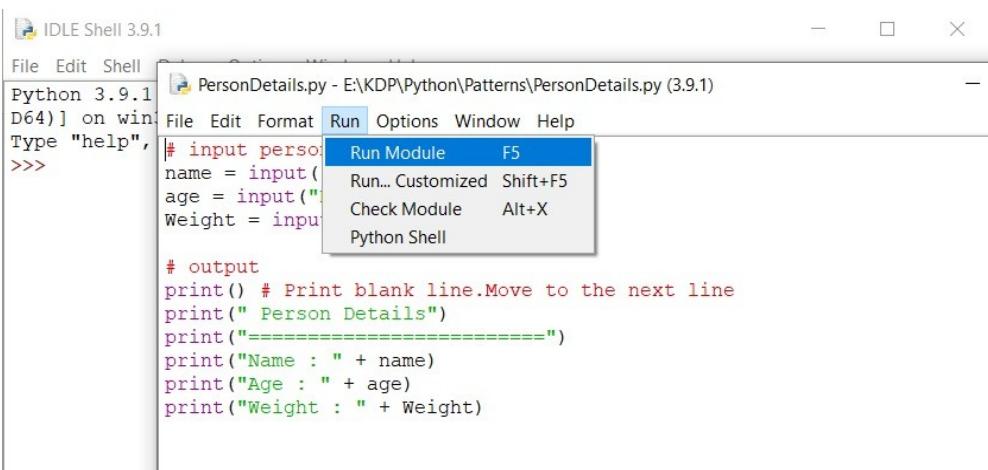


2.# Click the app to activate IDLE Shell.

3.# Navigate **File→ Open** to select the python programming file .py which we like to execute.



In our case, we had recently created **PersonDetails.py** file. We will select this file. From the newly opened window, please select the menu option **Run→ Run Module**.



4.# The IDLE shell will wait for keyboard input.

The screenshot shows the IDLE Shell 3.9.1 window. The title bar says "IDLE Shell 3.9.1". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The Python version is listed as "Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec 7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32". The shell prompt shows ">>>". The text "===== RESTART: E:\KDP\Python\Patterns\PersonDetails.py =====" is displayed. Below it, the text "Enter Name:" is shown, indicating the program is waiting for user input.

5.# After giving the input from the keyboard, please hit enter button. It will ask for the next input from the keyboard.

The screenshot shows the IDLE Shell 3.9.1 window and a separate code editor window for "PersonDetails.py". The code editor contains the following Python script:

```
# input person details
name = input("Enter Name:")
age = input("Enter Age:")
Weight = input("Enter Weight:")

# output
print() # Print blank line.Move to the next line
print(" Person Details")
print("=====")
print("Name : " + name)
print("Age : " + age)
print("Weight : " + Weight)
```

The IDLE Shell window shows the same setup as the previous screenshot, with the prompt "Enter Name:" and the message "===== RESTART: E:\KDP\Python\Patterns\PersonDetails.py =====".

6.#After giving all the inputs from the keyboard, please hit the enter button to see the output, which was programmed in that way.

The screenshot shows the IDLE Shell 3.9.1 window. The shell prompt shows ">>>". The text "===== RESTART: E:\KDP\Python\Patterns\PersonDetails.py =====" is displayed. Below it, the user has entered "Enter Name:jacob", "Enter Age:45", and "Enter Weight:95". The program then outputs " Person Details" followed by "=====". The final output is "Name : jacob", "Age : 45", and "Weight : 95".

Execution of this program demonstrated how to take data input from the keyboard. It outputted the same exactly on the console. A console is a utility that interacts with the OS to perform certain interactive tasks. The console was waiting for the user's data input from the keyboard. This exercise helps us know how to take dynamic input from the keyboard and pass it to the program. We need to repeat this process for a different set of data inputs to tag another person's details.

Operators

mathematical operations like addition, subtraction, comparing two numeric values, and many such operations. Calculation-based activities are very much common in our daily routine.

Arithmetic Operators

Operator	Description	Purpose	Coding Expression
+	Addition	Addition of <code>intNum1</code> and <code>intNum2</code> .	<code>intNum1 + intNum2</code>
-	Subtraction	Subtraction between <code>intNum1</code> and <code>intNum2</code> .	<code>intNum1 - intNum2</code>
*	Multiplication	Multiplying <code>intNum1</code> and <code>intNum2</code> .	<code>intNum1 * intNum2</code>
**	Exponentiation	<code>intNum1</code> raised to the power of <code>intNum2</code> .	<code>intNum1 ** intNum2</code>
/	Division	Quotient value when <code>intNum1</code> is divided by <code>intNum2</code> .	<code>intNum1 / intNum2</code>
//	Floor Division	Quotient value <u>without decimal values</u> when <code>intNum1</code> is divided by <code>intNum2</code> .	<code>intNum1 // intNum2</code>
%	Modulus	Remainder value when <code>intNum1</code> is divided by <code>intNum2</code> .	<code>intNum1 % intNum2</code>

Example 1. Demonstrate given basic arithmetic operations using arithmetic operators.

```
intNum1 = 6
intNum2 = 4

intNum1 + intNum2 = 10
intNum1 - intNum2 = 2
intNum1 * intNum2 = 24
intNum1 ** intNum2 = 1296
intNum1 / intNum2 = 1.5
intNum1 // intNum2 = 1
intNum1 % intNum2 = 2
```

```

ArithmeticOperations.py
intNum1 = 6
intNum2 = 4

print('intNum1 =', intNum1 )
print('intNum2 =', intNum2 )

result = intNum1 + intNum2
print('intNum1 + intNum2 =',result)

result = intNum1 - intNum2
print('intNum1 - intNum2 =',result)

result = intNum1 * intNum2
print('intNum1 * intNum2 =',result)

result = intNum1 ** intNum2
print('intNum1 ** intNum2 =',result)

result = intNum1 / intNum2
print('intNum1 / intNum2 =',result)

result = intNum1 // intNum2
print('intNum1 // intNum2 =',result)

result = intNum1 % intNum2
print('intNum1 % intNum2 =',result)

```

Output

```

intNum1 = 6
intNum2 = 4

intNum1 + intNum2 = 10
intNum1 - intNum2 = 2
intNum1 * intNum2 = 24
intNum1 ** intNum2 = 1296
intNum1 / intNum2 = 1.5
intNum1 // intNum2 = 1
intNum1 % intNum2 = 2

```

Exercise 2. Print basic arithmetic operations in the given tabular form.

Integer numbers: intNum1 = 6, intNum2 = 4

Operations	Operator	Expression	Result
Add	+	6 + 4	10
Subtract	-	6 - 4	2
Multiply	*	6 * 4	24
Divide	/	6 / 4	1.5
Remainder	%	6 % 4	2

Comparison Operators

The comparison result of two values will either be **True** or **False**, and these values are also known as **boolean values or truth values**. This feature is very important since it will help make decisions as True or False. The computer memory is mainly based on 1(True) or 0(False) values, called **a bit value**. Every data in a computer is a combination of one's and zero's called bit pattern.

Unique value or data in a computer's memory means a group of unique bit patterns.

Operator	Comparison Operation Description
<code>==</code>	Equal
<code>!=</code>	Not Equal
<code><</code>	Less than
<code><=</code>	Less than or equal
<code>></code>	Greater than
<code>>=</code>	Greater than or equal

Using comparison operators, we can compare two numeric values. The following example shows the use of comparison operators using numeric values.

Operator	Operand1(Operator) Operand2	Result
<code>==</code>	<code>4 == 4</code>	True
	<code>4 == 10</code>	False
<code>!=</code>	<code>6 != 4</code>	True
	<code>4 != 4</code>	False
<code><</code>	<code>6 < 10</code>	True
	<code>6 < 6</code>	False
<code><=</code>	<code>6 <= 6</code>	True
	<code>10 <= 6</code>	False
<code>></code>	<code>6 > 6</code>	False
	<code>10 > 6</code>	True
<code>>=</code>	<code>4 >= 6</code>	False
	<code>10 >= 6</code>	True

We can also test the above results by storing the input values in variables and then applying comparison operators one by one. **Philosophically, True and False are the only two perfect answers in this world**. In a program, it requires us to do decision-making based on perfect answers. Developing logic based on operators is the need for the implementation of decision-making. It allows flexibility in handling different logical situations in a program. Sense and sensibility of applying logical conditions are one of the core aspects of any good programmer.

So please, invest quality time studying this topic with special attention. A few special keyboard characters are used as operators in programming to denote frequent

Let's verify the results using comparison operators on operands 4,6 and 10.

```

ComparisonOperators.py
four = 4
six = 6
ten = 10
print("*** Comparison Operators ***")
print()
#equals to
print("equal (==)")
print()
isResult =(four == four)
print(four,"==",four,"|",isResult)

isResult =(four == ten)
print(four,"==",ten,"|",isResult)
print("-----")

#not equals to
print("notequal (!=")
print()
isResult =(four != four)
print(four,"!=",four,"|",isResult)

isResult =(four != ten)
print(four,"!=",ten,"|",isResult)
print("-----")

#less than
print("less than (<")
print()
isResult =(six <six)
print(six,"<",six,"|",isResult)

isResult =(six <ten)
print(six,"<",ten,"|",isResult)
print("-----")

#less than or equals to
print("less than or equal (<=")
print()
isResult =(six <= six)
print(six,"<=",six,"|",isResult)

isResult =(ten <= six)
print(ten,"<=",six,"|",isResult)
print("-----")

#greater than
print("greater than (>")
print()
isResult =(six >six)
print(six,">",six,"|",isResult)

```

```

isResult =(ten > six)
print(ten,>,six,"|",isResult)
print("-----")

#greater than or equal
print("greater than or equal (>=)")
print()
isResult =(six >= six)
print(six,>=,six,"|",isResult)

isResult =(six >= ten)
print(six,>=,ten,"|",isResult)
print("-----")

```

Output

*** Comparison Operators ***

equal (==)

```

4 == 4 | True
4 == 10 | False
-----
```

notequal (!=)

```

4 != 4 | False
4 != 10 | True
-----
```

less than (<)

```

6 < 6 | False
6 < 10 | True
-----
```

less than or equal (<=)

```

6 <= 6 | True
10 <= 6 | False
-----
```

greater than (>)

```

6 > 6 | False
10 > 6 | True
-----
```

greater than or equal (>=)

```

6 >= 6 | True
6 >= 10 | False
-----
```

Logical Operators

It also returns an ultimate value in either **True or False** similar to expression evaluation in comparison operator. It is primarily used to chain many logical expressions based on comparison operators. Following are types of logical operator's name and their respective symbol.

- 1) Logical AND (**and**)
- 2) Logical OR (**or**)
- 3) Logical NOT (**not**)

1) Logical AND: Returns **True** if and only if the result of **both logical expressions A and B are True** else False. The more we use this operator in a logical expression, the lesser the size of the data set we get out of total data. It is usually applied in a situation to get specific data or data of lesser range or size. Let's suppose there are two integer variables, month = 10 and days = 5. Please observe the column "Result."

A	B	A and B	A	B	A and B = Result
True	True	True	month > 7 (True)	days < 7 (True)	(True) and (True)= True
True	False	False	month > 7 (True)	days > 7 (False)	(True) and (False)= False
False	True	False	month < 10 (False)	days < 7 (True)	(False) and (True)= False
False	False	False	month < 10 (False)	days > 7 (False)	(False) and (False)= False

2) Logical OR: Returns **False** if and only if when the result of **both logical expressions A and B are False** else True. The more we add OR condition using this operator in a logical expression, the more it increases the size of data as an output result. So, it is used to get a broader range of data. Let's suppose there are two integer variables, month = 10 and days = 5. Please observe the column "Result."

A	B	A or B	A	B	A or B = Result
True	True	True	month > 7 (True)	days < 7 (True)	(True) or (True)= True
True	False	True	month > 7 (True)	days > 7 (False)	(True) or (False)= True
False	True	True	month < 10 (False)	days < 7 (True)	(False) or (True)= True
False	False	False	month < 10 (False)	days > 7 (False)	(False) or (False)= False

3) Logical NOT: Returns just opposite, **True** when the outcome of the logical expression is False or returns False when the outcome of a logical expression is True.

A	not A
True	False
False	True

Please note that True/False values, also known as boolean values or truth values.

Usage of logical operators is quite common and plays a very important role in designing the logic of programs. Every programmer must and should have an obvious idea about the usage of these operators. It is worth the time spent if a beginner puts extra effort into experimenting with these operators.

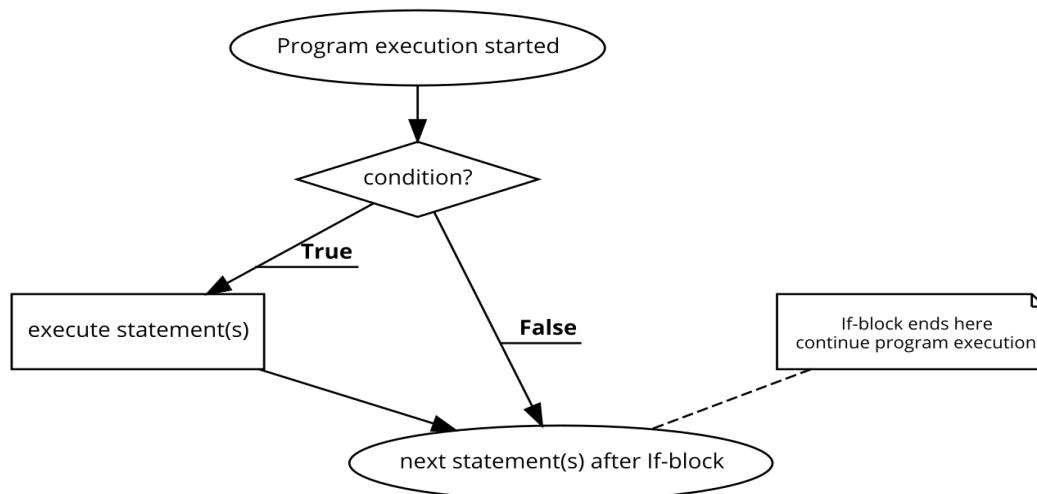
Decision Making

During the design of the program's logic, compute the decision-making first. There are only two perfect answers in this world, either **YES** or **NO**. In programming, these perfect answers represent the boolean data type. The boolean data type variable takes a single value at a time, which can be **True** or **False**. So boolean value **True** represents **YES**, and the boolean value **False** represents **NO**. While running the program, the control of the program may find logical expression. It then tests this logical expression to find the result. It expresses the result in terms of either **True** or **False**. Based on the boolean result **True** or **False** of the logical expression, the program's control decides whether to execute or skip a bunch of programming statements. We can compute decision-making using the following different versions of decision-making **if-statements**.

- a. **If – statement**
- b. **If – else: statement**
- c) **If – elif – else: statement (Also known as ladder if-else statement)**

Flow-chart helps to visualize the flow of control when the condition becomes either **True** or **False**.

a) if-statement



The general syntax for **if-statement** can be given as:

```
if (condition)
    execute statement(s)
    next statement(s) // continue program execution
```

If-block

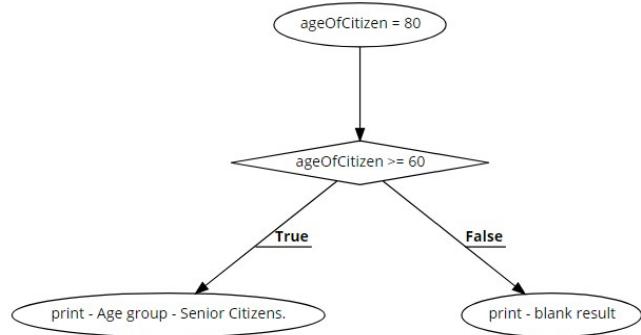
- **condition:** It will be an expression consisting of variables and operators in such a way whose outcome should be a boolean value, either **True** or **False**. Since logical and comparison operators give either **True** or **False** and are commonly used in a conditional expression.
- When the result of a **condition** is **True**, it will execute statement(s) enclosed in the **If-block**

Case 1.

`ageOfCitizen = 80`

`condition: (ageOfCitizen >= 60)` will evaluate to **True**

So, `if (80 >= 60)` or if (`condition`) will be **True**, then it executes the if-block.



StatementIf.py

```
ageOfCitizen = 80
if ageOfCitizen >= 60:
    print("Age group - Senior Citizens.")
```

Output

Age group - Senior Citizens.

Case 2. Change the value of ageOfCitizen in the program.

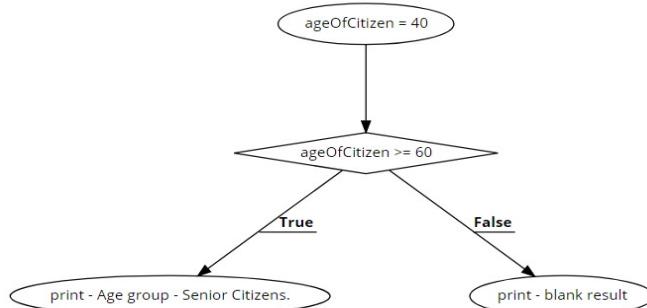
`ageOfCitizen = 40`

`condition: (ageOfCitizen >= 60)`

`if(40 >= 60)`, will evaluate to **False**.

`if(ageOfCitizen >= 60)` will be **False** and it will not execute the if-block.

The output will be blank.



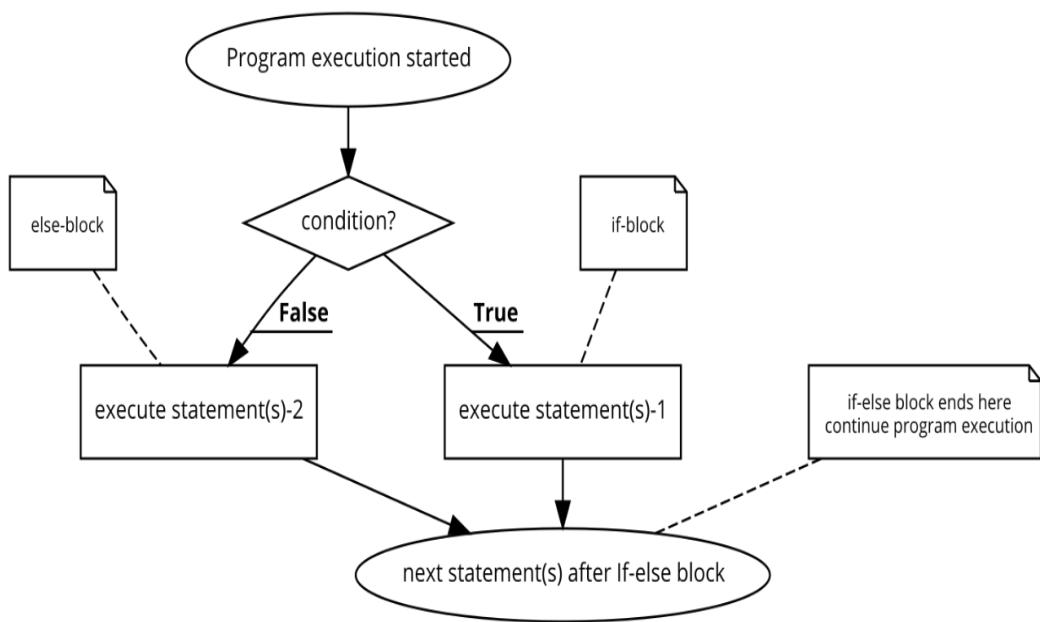
StatementIf1.py

```
ageOfCitizen = 40
if ageOfCitizen >= 60:
    print("Age group - Senior Citizens.")
```

Output

It didn't print anything since the condition becomes failed and code block within **if-statement** doesn't get executed.

b) if-else statement



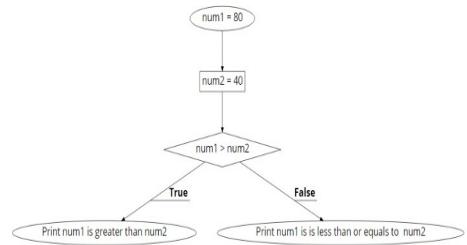
The general syntax for **if-else statement** can be given as:

if (condition)
execute statement(s)-1 If-block
else :
execute statement(s)-2 else-block
next statement(s) ← continue execution of program

if(logical_condition)
statement 1-1
statement 1-2
.....
statement 1-N
else :
statement 2-1
statement 2-2
.....
statement 2-N

Case 1.

```
num1 = 80 condition:( num1 > num2)
num2 = 40
    if(80 > 40), will evaluate to True.
    if( num1 > num2) conditional
    statement will be True and executes
    the if-block.
```



StatementIfElse.py

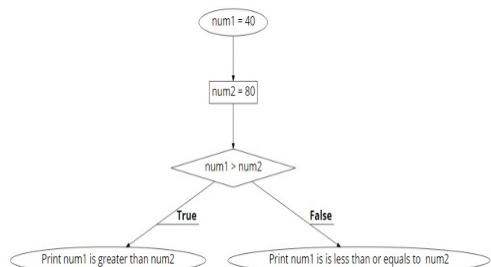
```
num1 = 80
num2 = 40
if num1 > num2:
    print(num1," is greater than ",num2)
else:
    print( num1," is less than or equals to ", num2)
```

Output

```
80 is greater than 40
```

Case 2. Change the value of num1 and num2 in the program.

```
num1 = 40 condition:( num1 > num2)
num2 = 80 if(40 > 80) statement will
evaluate to False. Hence,
if( num1 > num2)
conditional statement will be
False. Program should execute the
else code block.
```



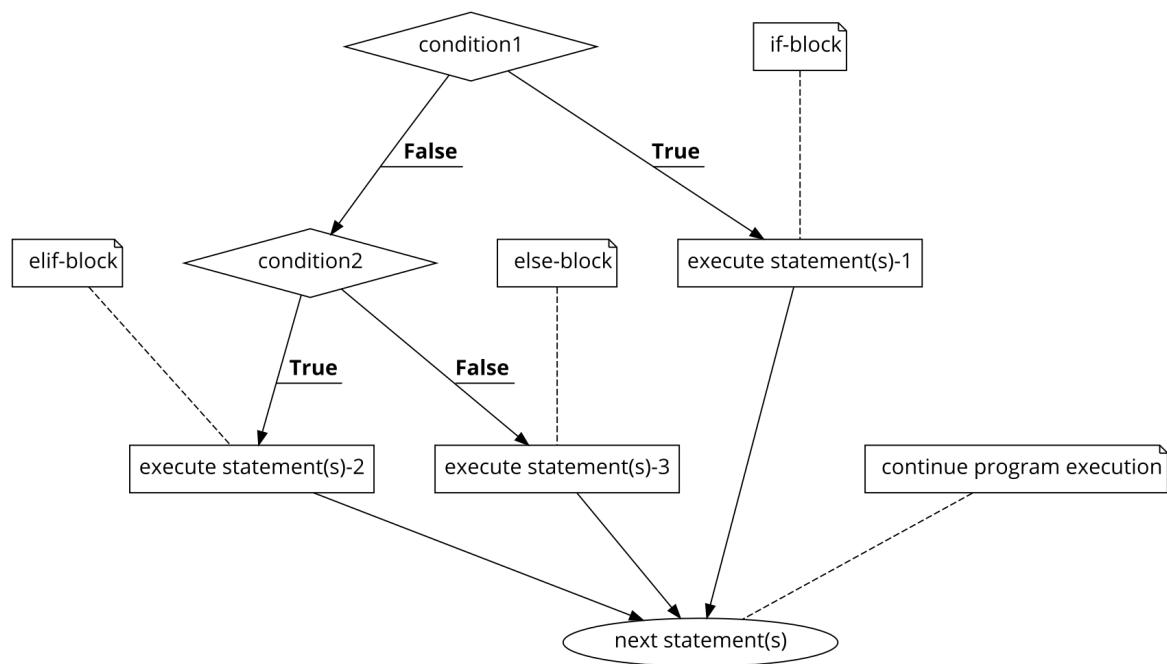
StatementIfElse.py

```
num1 = 40
num2 = 80
if num1 > num2:
    print(num1," is greater than ",num2)
else:
    print( num1," is less than ",num2)
```

Output

```
40 is less than 80
```

c) if-elif-else statement



The general syntax for **if-elif-else statement** can be given as:

```

if (condition1)
  execute statement(s)-1
elif(condition2)
  execute statement(s)-2
else:
  execute statement(s)-3
  
```

next statement(s) ← continue program execution

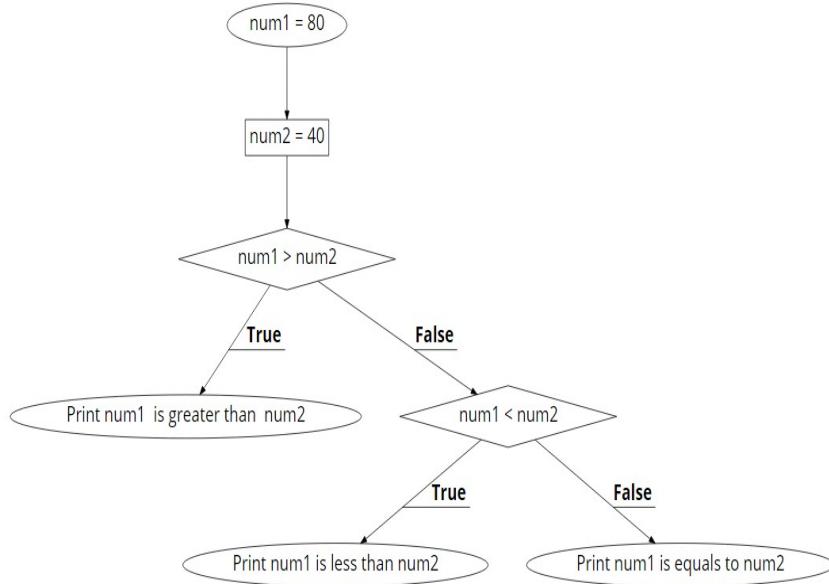
if block
else-if block
else block

```

if(logical_condition1)
  statement 1-1
  statement 1-2
  .....
  statement 1-N
elif (logical_condition2)
  statement 2-1
  statement 2-2
  .....
  statement 2-N
else:
  statement 3-1
  statement 3-2
  .....
  statement 3-N
  
```


Case 1.

```
num1 = 80
num2 = 40
condition :(num1 > num2)
if (80 > 40), will evaluate to True
Hence, if (num1 > num2) conditional statement will be True .
The program should execute the if code block.
```



StatementIfElseIf.py

```
num1 = 80
num2 = 40
if num1 > num2:
    print(num1," is greater than ", num2)
elif num1 < num2:
    print(num1," is less than %d", num2)
else:
    print(num1," equals to ", num2)
```

Output

```
80 is greater than 40
```


Case 2. Change the value of num1 and num2 in the program.

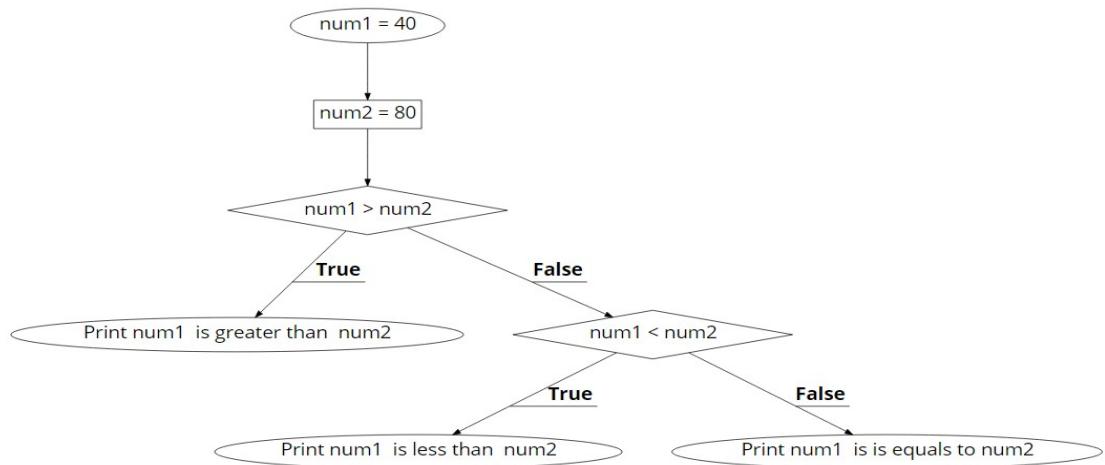
```
num1 = 40
```

```
num2 = 80
```

if(num1 > num2) conditional statement will evaluate to **False** and program's control will check next if conditional statement, which is **else if(num1 < num2)**.

if(num1 < num2) conditional statement will be **True**, since **(40 < 80)** evaluates to **True**.

The program should execute the **else if code block**.



StatementIfElseIf.py

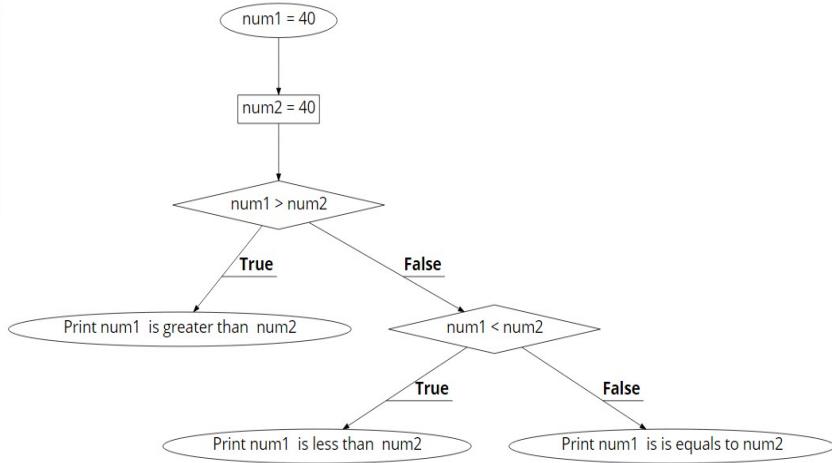
```
num1 = 40
num2 = 80
if num1 > num2 :
    print(num1," is greater than ", num2)
elif num1 < num2:
    print(num1," is less than ",num2)
else:
    print(num1," equals to ",num1)
```

Output

```
40 is less than 80
```


Case 3. Change the value of num2 in the program.

```
num1 = 40  
num2 = 40
```



The (`num1 > num2`) logical expression will test to **False**, and then the program's control will test the next logical expression (`num1 < num2`). The logical expression (`num1 < num2`) will test as **False** because (`40 < 40`) will test as False. So, else if (`num1 < num2`) will also **not get executed** and finally execute the last else code block. The else-block has no condition attached to it. The **last else code block** will get executed when all the conditions will become failed. It's a default block, and at least this block will get executed if no other block.

StatementIfElseIf.py

```
num1 = 40  
num2 = 40  
if num1 > num2 :  
    print(num1," is greater than ", num2)  
elif num1 < num2:  
    print(num1," is less than ",num2)  
else:  
    print(num1," equals to ",num1)
```

Output

```
40 equals to 40
```

ProblemLadderIfElse: A college came up with an idea to encourage CS students by refunding some percentage of their bi-semester CS Course fees. College management laid the following conditions to be eligible for the same. Moreover, if a student scored in Grade-0, they will be declared failed.

If a student scored in Grade-5, they would be eligible for a 35% refund of the college fees.

If a student scored in Grade-4, they would be eligible for a 25% refund of the college fees.

If a student scored in Grade-3, they would be eligible for a 20% refund of the college fees.

If a student scored in Grade-2, they would be eligible for a 10% refund of the college fees.

If a student scored in Grade-1, they would be passed with a margin.

LadderIfElse.py

```
grade = 6
if grade == 1:
    print("35% fee refund.")
elif grade == 2:
    print("25% fee refund.")
elif grade == 3:
    print("20% fee refund.")
elif grade == 4:
    print("10% fee refund.")
elif grade == 5:
    print("Passed with margin.")
elif grade == 0:
    print("Failed,please retry.")
else:
    print("Invalid grade ?", grade)
```

It's going to test **if-conditions** sequence-wise unless and until any one of the **if-condition** becomes **True**. This type of many elif is also called a **ladder else-if**, which is basically not a good practice. Two or three **elif conditions** are considered okay, and furthermore, **elif conditions** are strongly discouraged as it reduces the code readability.

Then what is the way out?

The answer is implementing a **switch-case** construct. We use Python's built-in dictionary construct that performs mapping between grade value and respective printing message. Dictionary basically is a collection of key-value pairs.

Grade (key)	Printing Message(value as string)	Dictionary Example
Mapping 0 →	"Failed,please retry."	grade_message = 0:"Failed,please retry.",
Mapping 1 →	"35% fee refund."	1:"35% fee refund.",
Mapping 2 →	"25% fee refund."	2:"25% fee refund."
Mapping 3 →	"20% fee refund."	3:"20% fee refund.",
Mapping 4 →	"10% fee refund."	4:"10% fee refund.",
Mapping 5 →	"Passed with margin."	5:"Passed with margin."
Mapping Other value→	"Invalid grade ?"	

Let's convert the ladder if-else statement to a switch-case. Then, the previous code `LadderIfElse.py` can be re-written as:

```
LadderIfElseToSwitchCase.py
# Dictionary for mapping score grade to message.
grade_message =
    0:"Failed,please retry.",
    1:"35% fee refund.",
    2:"25% fee refund.",
    3:"20% fee refund.",
    4:"10% fee refund.",
    5:"Passed with margin."

# use of get() method to print message which is mapped to score grade using #
# dictionary grade_message.
grade = 2
message = grade_message.get(grade,"Invalid grade ?")
print(" grade = 2 :", message)

# default message for unlisted score grade value.
grade = 6
message = grade_message.get(grade,"Invalid grade ?")
print(" grade = 6 :", message)
```

Output

```
grade = 2 : 25% fee refund.
grade = 6 : Invalid grade ?
```

Python's switch-case using the dictionary is one of the elegant ways of dealing with **ladder if-conditions**.

Problem: Write the appropriate dictionary and a program to print the weekdays: Tuesday & Sunday.

Grade (key)	Printing Message(value as string)	Dictionary ?
Mapping 0 →	"Sunday"	
Mapping 1 →	"Monday"	
Mapping 2 →	"Tuesday"	
Mapping 3 →	"Wednesday"	
Mapping 4 →	"Thursday"	
Mapping 5 →	"Friday"	
Mapping 6 →	"Saturday"	
Mapping Other value →	"Day – None "	

About *for-loop*

The feature of repetitiveness helps automate tedious data processing work with amazing efficiency compared to doing it manually. One of the types of repetitive control in programming is known as **for statement** or **for-loop**. Let us consider an integer variable name as ‘counter.’ By definition integer variable stores only one value at a time. We want this **variable: counter** to store single data in a planned way one by one mentioned in the following sequence-wise;

2, 4, 6, 8, 10, 12, 14, 16, 18, 20

Let's calculate the difference of value between all consecutive pairs of values in a sequence. It means we are required to calculate (**next value — previous value**)= **value difference**. The difference of value should be common across all the (**next value — previous value**) pairs.

Let's calculate the value difference for the given sequence.

Sequence	Next - Previous	Difference	variable:counter updates	Final value
2			counter = 2	2
4	4 – 2	2	counter = counter + 2	2 + 2 = 4
6	6 – 4	2	counter = counter + 2	4 + 2 = 6
8	8 – 6	2	counter = counter + 2	6 + 2 = 8
10	10 – 8	2	counter = counter + 2	8 + 2 = 10
12	12 -10	2	counter = counter + 2	10 + 2 = 12
14	14 -12	2	counter = counter + 2	12 + 2 = 14
16	16 – 14	2	counter = counter + 2	14 + 2 = 16
18	18 – 16	2	counter = counter + 2	16 + 2 = 18
20	20 – 18	2	counter = counter + 2	18 + 2 = 20

Since the difference of value is common, it will term it as a common difference, which is value 2 in this sequence. By applying the definition of a variable, we can express storing a sequence of values using the **variable: counter**. Please execute the following program to confirm the sequence.

SequenceWithoutForLoop.py

```
counter = 2
print(counter)
counter = counter + 2
print(counter)
```

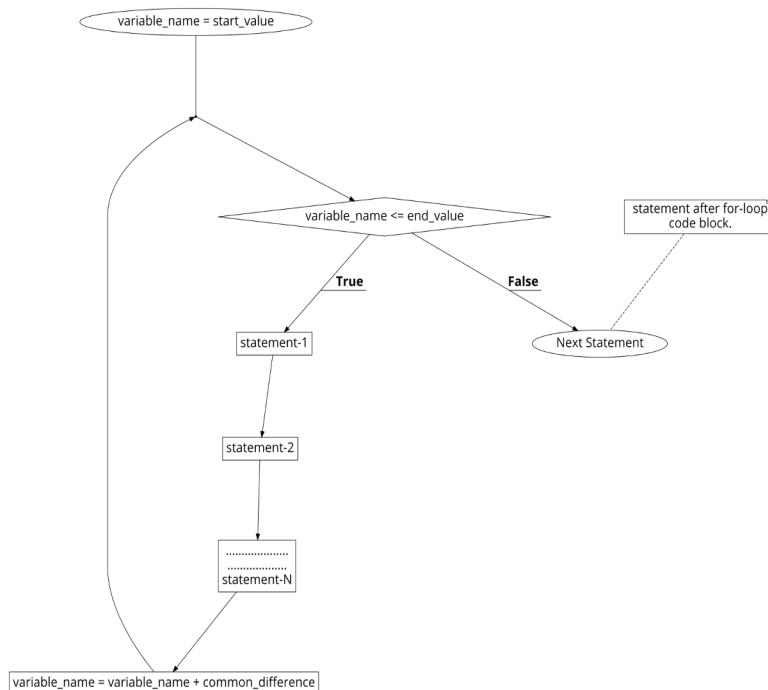
- Suppose this sequence becomes quite large like 2,4,6,....., 2000. We will write almost 4000 lines of code and repeat the same set of programming statements almost 2000 times.
- In the case of high calculation-intensive data or large data processing activities, humans are not reliable in terms of accuracy and productivity compared to computers. Nothing is automatic unless someone does it. We can automate the generation of the considered sequence using the **for-loop** and **range()** function.
- The **for-loop** helps to repeat a set of programming statement(s) over a **range of values**. When we talk about a range, then it must have a starting value and an ending value.

We can describe **range()** function with three inputs like the following;

range(start_value, end_value+1, common_difference) where **start_value < end_value**

For-loop with range() function conceptually does the following.

- **start_value** is a simple way of initialization a variable, and anyway, some point in the process of the repetitive cycle has to be started.
- **end_value** is a logical expression that evaluates a boolean value, either **True** or **False**. Based on this (**True/False**), the **for-loop** control structure decides whether to repeat the set of the statement(s) or not.
- **common_difference** is a change in a variable's value in a fixed smaller step (increment/decrement) only after all statement(s) get executed in the **for-loop** block. It will update the variable's value by incrementing or decrementing it till it matches the **end_value**.



```

# 2nd input -> end of for-loop. To include boundary value +1 is added.
for variable_name in range(start_value, end_value + 1, common_difference):
    statement-1
    statement-2
    .....
    .....
    statement-N
    Next Statement # statement after for-loop code block.
  
```

A few observations from the considered given sequence.

2, 4, 6, 8, 10, 12, 14, 16, 18, 20

The start value or minimum value is 2. The end value or maximum value is 20. We also knew that the common difference value is 2. Hence, the **range()** function with three inputs can be given as **range(2, 20 + 1, 2)**.

The **for-loop** with function `range(2, 20, 2)` can be coded as follows:

```
SequenceWithForLoop.py
# counter : An integer variable
# range function :- 
# 1st input -> start of for-loop.
# 2nd input -> end of for-loop. To include boundary value +1 is added.
# 3rd input -> common difference value
for counter in range(2, 20 + 1, 2):
    print(counter)
```

Output

```
2
4
6
8
10
12
14
16
18
20
```

The common difference value by default will be 1 if we exclude 3rd input in the range function.

```
ForLoopWithRangeTwoInputs.py
# counter : An integer variable
# range function :- 
# 1st input -> start of for-loop
# 2nd input -> end of for-loop. To include boundary value +1 is added.
for counter in range(2, 5 + 1):
    print(counter)
```

Output

```
2
3
4
5
```

The **range()** function with single input by default will start sequence with value 0, and the common difference value will be 1. The **for-loop** with function `range(5+1)` can be expressed as follows:

```
ForLoopWithRangeOneInput.py
# counter : An integer variable
# single input =(5 + 1)-> end of for-loop.
# The range function excludes the end boundary value and so +1 is added.
for counter in range(5 + 1):
    print(counter)
```

Output

```
0
1
2
3
4
5
```

Decrementing For-Loop:

Let's express `range(start_value, end_value -1, common_difference)` where `start_value > end_value` with `common_difference` as negative value. In general, we can give decrementing for-loop as;

```
# 2nd input -> end of for-loop. To include boundary value -1 is subtracted.  
for variable_name in range(start_value, end_value - 1, common_difference):  
    statement-1  
    statement-2  
    .....  
    .....  
    statement-N
```

Next Statement `# statement after for-loop code block.`

Let's try to reverse the previous sequence as;

20, 18, 16, 14, 12, 10, 8, 6, 4, 2

The start value or maximum value is 20. The end value or minimum value is 2. We also knew that the common difference value is -2. Hence, the `range()` function with three inputs can be given as `range(20, 2 -1,-2)`.

```
ForLoopDecrement.py  
# counter : An integer variable  
# range function :-  
# 1st input -> start of for-loop.  
# 2nd input -> end of for-loop. To include boundary value -1 is subtracted.  
# 3rd input -> common difference as negative value  
for counter in range(20, 2 -1,-2):  
    print(counter)
```

Output

20
18
16
14
12
10
8
6
4
2

Problem: Write for-loop for the given sequence.

-10
-9
-8
-7
-6
-5
-4
-3
-2

For-loop with break and continue keywords

- Any logical condition that executes the keyword **break** within the for-loop stops further repetition of the loop. It's similar to a **moving car that stops** exactly whenever the driver applies the **break**.
- Any logical condition that executes the keyword **continues** within the for-loop; it continues with the next iteration without executing programming statement(s) that come after it within the for-loop's code block.

ForLoopContinueNext.py

```
for cnt in range(1,10 + 1):
    if cnt <= 3:
        print("Keyword continue is throwing ",end="")
        print("control of program back.\n")
        continue

    print("Please Join Us...!!!\n")
    if cnt == 5:
        print("keyword break at cnt =",cnt,"\\n")
        break
    print("Please look back...\n")
```

Output

```
Keyword continue is throwing back.
Keyword continue is throwing back.

Keyword continue is throwing back.

Please Join Us...!!!

Please look back...

Please Join Us...!!!

keyword break at cnt = 5
```

Nested for-loop/Inner for-loop

We have previously seen examples of **for-loop** that have if-statements or any other programming statements. Instead of programming statement(s) within for-loop, if we place another **for-loop** within the **for-loop**, then how things will be. For example;

```
for counter in range(5,0,-1):
    print(1*counter)# statement1
    print(2*counter)# statement2
    print(3*counter)# statement3
```

We know it will execute these considered statements in order. Firstly, **statement1** will get executed, then **statement2** and lastly **statement3** will get executed. These statements will repeat five times when we execute the considered piece of the program. If we place another for-loop instead of any simple statement, the major style of work doesn't change. We need to think that **the inner for-loop** itself is like some other single programming statement. Suppose the control of a program executes programming statements in a sequence and finds an **inner for-loop**. It will then fully execute the **inner for-loop** and then execute the next programming statement(s) if there are any available. For example;

```
for counter in range(5,0,-1):
    print(1*counter)# statement1
    # The for-loop is a statement2

for inner in range(1,5):
    print(inner)

print(3*counter)# statement3
```

Please treat
nested for-
loop as a
single

As usual, **statement1** gets completed first. The inner for-loop will perform with 5 times repetitions, and it will finally execute **statements3**. These three statements will repeat five times, as suggested by the structure of the outer for-loop. A for-loop placed within the code block of another for-loop is known as **inner for-loop** or **nested for-loop**.

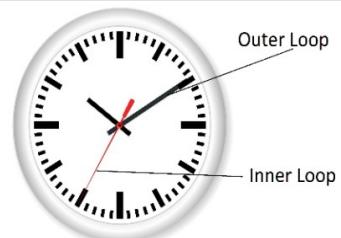
The inner for-loop will **restart from the beginning** for every new value of an outer for-loop.

It is easy to follow this concept if we try to relate it with a time piece clock with the following assumptions

Inner Loop → It's like a "**seconds**" hand in a clock

Outer Loop → It's like a "**minute**" hand in a clock

In a clock, when the "**seconds**" hand completes one cycle, only then will the "**minute**" hand get updated by counter value 1 and not before.



The **inner for-loop will execute completely before every new counter value of the outer for-loop** to continue repetition depends upon its loop condition.

NestedForLoop.py

```
for out in range(1,3):
    print("OUTER count No.",out)
        # nested for-loop: always re-starts and finishes
        # completely for every new value of outer for-loop.
    for In in range(1,4):
        print("<INNER> count No.",In)
```

Output

```
OUTER count No.1
<INNER> count No.1
<INNER> count No.2
<INNER> count No.3
OUTER count No.2
<INNER> count No.1
<INNER> count No.2
<INNER> count No.3
```

Nested Loop: Beginner Programmer's Best Friend

A best friend is a person, whenever required, who not only appreciates our wonderful qualities but also gives us a reality check by showing our weaknesses or mistakes. The nested loop plays such a best friend's role while learning to program logic, and it also gives a reality check of how much True our programming logic is. This is my opinion that **practicing programming logic just by juggling numbers is not a good idea**. This mostly is held True for a person who had less or no inclination in high school mathematics. Many of them feel bored practicing programming logic based on mathematical problems. Everybody is different, and everyone has a different buffer of patience and the pace of learning. There should be a simple approach to learning a program. Any beginner who wants to learn/improve programming should be able to do it with his own learning pace on the backdrop of minimal technical stuff. The approach uses a three-step learning process, guided by illustrated exercises, to ensure beginners become comfortable with the core of programming.

Step 1. The clarity about how logical conditions can be explored.

Step 2. The clarity about how nested-loop works.

Step 3. Research alternatives to simplify programming logic.

Learners should be confident in exploring advanced features of the programming language after studying these three steps.

Why are nested loops such an important deal?

In the outer for-loop and inner for-loop, each one is used for outputting data in different directions horizontally and vertically to give a 2-dimensional output. Usually, the outer for-loop moves the control vertically, and the inner for-loop outputs the data horizontally. This gives a grid kind of output. This particular feature of looping can be utilized to perfect the imagination of the learner. Yes, logic building is a skill and can be developed using simple math and common sense. It's not mandatory for a person to be very good at math to be successful programmers or software developers. If a learner can imagine the behavior of outer and nested loops correctly, the person will gain enough confidence to code. The first and foremost requirement for any subject-related teaching should be to create interest among the student or learner groups. If a learner gets deeply connected with the subject, then the rest of the learning will be without burden. This course is strictly meant to enjoy learning. If any learner takes this course and follows it honestly, they will surely see the fruits of effort without a doubt.

A Nested loop gives a good drilling exercise to a new/novice/beginner/unconfident programmer. Once a learner becomes interested in programming, then all the learning will become thrilling and exciting. That interest will naturally come when the learner gets some confidence in developing programming logic on its own. For that purpose, only very few things need to be practiced well and covered in appropriate depth. I strongly suggest reading and practicing this book first entirely before seeking out any other resources. This book helps you progress faster, and it's a kind of essential pre-workout for programming.

I don't agree that those who give up learning programming are less capable, but it doesn't excite them to put in efforts. Whatever may be the learning pace of a person, it doesn't matter but what matters is the involvement in learning. A true kind of involvement is not at all possible without interest. This book is all about proper installation of interest and confidence that any person can do the programming by putting some efforts in the right direction.

The selection of exercises and problems is made because learning should be naturally adaptable to any learner. The learner needs to follow the step-by-step instructions mentioned in this book. The confidence and interest in programming will be assured as you naturally learn and complete the training. This is not just a book, basically a training guide. The initial learning phase definitely requires some effort and patience. Once a learner understood the required fundamentals, further improvement will be sky-rocketing due to immense love for a subject. It will come naturally during practice. These sets of exercises result from experiments conducted to teach programming for the beginner's level and who had none or some programming experience. On completion of these exercises, everyone will be able to develop a good level of confidence within 2-3 months. The high point was that students became interested in programming. Programming has never remained boring or a burden for them. Learners will be able to naturally develop a tendency of patience to push him/her to solve problems and be totally focused. In short, it makes the learning interesting.

ExerciseGB1. For every single value of outer for-loop in an iteration, inner for-loop executes completely. Unless and until the inner for-loop gets executed completely, the outer for-loop next iteration won't happen. The following program gives the demonstration of printing the value of columns vertically for every single row value.

ForLoopEveryRowAllColumns.py

```
TOTAL_ROWS = 3
TOTAL_COLUMNS = 3
for row in range(1,TOTAL_ROWS + 1):
    for col in range(1,TOTAL_COLUMNS + 1):
        print("Row =", row, "| Col =", col)
    print("-----")
```

Output

```
Row = 1| Col = 1
Row = 1| Col = 2
Row = 1| Col = 3
-----
Row = 2| Col = 1
Row = 2| Col = 2
Row = 2| Col = 3
-----
Row = 3| Col = 1
Row = 3| Col = 2
Row = 3| Col = 3
-----
```

ExerciseGB2. Printing column iteration values horizontally for every iteration value of the row. The row value should come in every different row. So row-level iteration values should print vertically.

ForLoopEveryRowAllColumnsHorizontally.py

```
TOTAL_ROWS = 3
TOTAL_COLUMNS = 3
for row in range(1,TOTAL_ROWS+1):
    print("Row =",row,"|",end="")
    # prints in the same row
    for col in range(1,TOTAL_COLUMNS +1):
        print(" Col =",col,end="")
    # moves to the next line and prints
    print()
```

Output

```
Row = 1 | Col = 1 Col = 2 Col = 3
Row = 2 | Col = 1 Col = 2 Col = 3
Row = 3 | Col = 1 Col = 2 Col = 3
```

ExerciseGB3. Printing row iteration and column iteration values as (row, col) as shown in the following output

```
(1,1)(1,2)(1,3)(1,4)(1,5)
(2,1)(2,2)(2,3)(2,4)(2,5)
(3,1)(3,2)(3,3)(3,4)(3,5)
(4,1)(4,2)(4,3)(4,4)(4,5)
(5,1)(5,2)(5,3)(5,4)(5,5)
```

ForLoopCoordinateRepresentation.py

```
TOTAL_ROWS = 5
TOTAL_COLUMNS = 5

for row in range( 1, TOTAL_ROWS + 1 ):
    # prints in the same row
    for col in range(1,TOTAL_COLUMNS +1):
        print("(",row ,",",col,end="")")
    # moves to the next line and prints
    print()
```

Output

```
(1,1)(1,2)(1,3)(1,4)(1,5)
(2,1)(2,2)(2,3)(2,4)(2,5)
(3,1)(3,2)(3,3)(3,4)(3,5)
(4,1)(4,2)(4,3)(4,4)(4,5)
(5,1)(5,2)(5,3)(5,4)(5,5)
```

ExerciseGB4. Generating pattern using character “*” by printing it for every (row, col) iteration value.

RowColumnStars.py

```
TOTAL_ROWS = 7
TOTAL_COLUMNS = 7
for row in range(1,TOTAL_ROWS+1):
    for col in range(1, TOTAL_COLUMNS + 1 ):
        # print "*" characters in the same line
        print("*",end="")
        # moves the control to the next line
        # printing of "*" characters starts from next line
    print()
```

Output

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

ProblemGB3: Write a program for the following output considering rows and columns are 5.

```
*-*-*-*-
*-*-*-*-
*-*-*-*-
*-*-*-*-
*-*-*-*-
```

Problem GB4: Please correct the following program.

```
TOTAL_ROWS = 4
TOTAL_COLUMNS = 4
for row in range(1,TOTAL_ROWS+1):
    for col in range(1, TOTAL_COLUMNS + 1 ):
        # print "*" characters in the same line
        print("*",end="")
    print()
```

Output

```
*
```

Expected Output

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

JustLoopIt: Write a separate program for each of the given outputs.

LoopingProblemA

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

LoopingProblemB

```
1!=1 1!=2 1!=3 1!=4 1!=5 1!=6 1!=7 1!=8
2!=1 2!=2 2!=3 2!=4 2!=5 2!=6 2!=7 2!=8
3!=1 3!=2 3!=3 3!=4 3!=5 3!=6 3!=7 3!=8
4!=1 4!=2 4!=3 4!=4 4!=5 4!=6 4!=7 4!=8
5!=1 5!=2 5!=3 5!=4 5!=5 5!=6 5!=7 5!=8
```

LoopingProblemC

```
1<=1 1<=2 1<=3 1<=4 1<=5
2<=1 2<=2 2<=3 2<=4 2<=5
3<=1 3<=2 3<=3 3<=4 3<=5
4<=1 4<=2 4<=3 4<=4 4<=5
5<=1 5<=2 5<=3 5<=4 5<=5
```

LoopingProblemD

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

LoopingProblemE

```
1>=1 1>=2 1>=3 1>=4
2>=1 2>=2 2>=3 2>=4
3>=1 3>=2 3>=3 3>=4
4>=1 4>=2 4>=3 4>=4
1>=1 1>=1 1>=1 1>=1
2>=2 2>=2 2>=2 2>=2
3>=3 3>=3 3>=3 3>=3
4>=4 4>=4 4>=4 4>=4
```

LoopingProblemF

```
1>=1 1>=2 1>=3 1>=4 1>=5 1>=6 1>=7 1>=8
2>=1 2>=2 2>=3 2>=4 2>=5 2>=6 2>=7 2>=8
3>=1 3>=2 3>=3 3>=4 3>=5 3>=6 3>=7 3>=8
4>=1 4>=2 4>=3 4>=4 4>=5 4>=6 4>=7 4>=8
5>=1 5>=2 5>=3 5>=4 5>=5 5>=6 5>=7 5>=8
6>=8 5>=7 5>=6 5>=5 5>=4 5>=3 5>=2 5>=1
7>=8 5>=7 5>=6 5>=5 5>=4 5>=3 5>=2 5>=1
8>=8 5>=7 5>=6 5>=5 5>=4 5>=3 5>=2 5>=1
```

LoopingProblemG

```
1!=1 1!=3 1!=5 1!=7
2!=1 2!=3 2!=5 2!=7
3!=1 3!=3 3!=5 3!=7
4!=1 4!=3 4!=5 4!=7
5!=1 5!=3 5!=5 5!=7
4!=1 4!=3 4!=5 4!=7
3!=1 3!=3 3!=5 3!=7
2!=1 2!=3 2!=5 2!=7
1!=1 1!=3 1!=5 1!=7
```

LoopingProblemH

```
2>=2 2>=4 2>=6 2>=8
3>=2 3>=4 3>=6 3>=8
2>=2 2>=4 2>=6 2>=8
1>=2 1>=4 1>=6 1>=8
2>=2 2>=4 2>=6 2>=8
3>=2 3>=4 3>=6 3>=8
4>=2 4>=4 4>=6 4>=8
5>=2 5>=4 5>=6 5>=8
```

Art of Applying Logical Conditions

Logical conditions are the defense mechanism of working features of any software application. Most of the bugs/errors which occurred are because of the improper handling of logical condition(s). It mostly comprises **logical and comparison operators**. One of the most important elements of programming logic is to apply logical conditions. It controls the flow of the program similarly to how the road traffic control system works. Road traffic control system logically distributes the crowd by conditionally allowing or stopping the portion of the crowd directing towards a particular road. Similarly, the execution of logical conditions works like a traffic control system that allows or disallows the execution of a program's chosen set of programming statements. If the roadside traffic control system doesn't work appropriately, it may create traffic jams and accidents instead of guiding traffic. Similarly, wrong logical conditions give unpredictable results. We are going to learn how we can become fluent in applying logical conditions accurately.

The roadside signboards are for a specific purpose. Similarly, with programming, we have to understand the motive behind the usage of each operator. Therefore, we have to use operators in logical conditions based on their specific purpose, which can precisely influence a program's output. In the same way, we will try to understand how logical conditions influence the program's final output. We will use our version of a simple scratchpad to verify our logical conditions manually first. It may remind you of a cross-words puzzle box. Let us consider a similar box and consider that it comprises 8 rows and 8 columns. For representing rows and columns, we can consider two variables named **row** and **col**, respectively.

Variable **row** → denotes values of row=1, 2, 3, 4, 5, 6, 7, 8.

Variable **col** → denotes values of col=1, 2, 3, 4, 5, 6, 7, 8.

Each box is recognized by (row, col) value pair. So value pair (1, 1) means 1st row and 1st column.

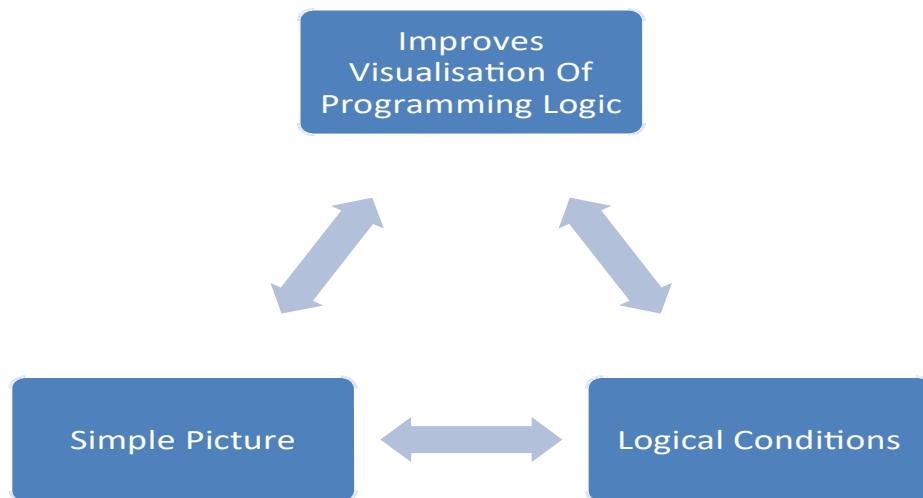
	col=1	col=2	col=3	col=4	col=5	col=6	col=7	col=8
row=1	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8
row=2	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8
row=3	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8
row=4	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8
row=5	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8
row=6	6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8
row=7	7,1	7,2	7,3	7,4	7,5	7,6	7,7	7,8
row=8	8,1	8,2	8,3	8,4	8,5	8,6	8,7	8,8

Rule of the game: Where ever logical condition matches, you need to put character * in a small box.

All (row, col) value pairs represent the complete information or data. It has 8 rows and 8 columns, making 64 (row, col) value pairs. We will select correct (row, col) value pairs from the total available 64 (row, col) value pairs into this box to map logical conditions. Applying different logical conditions means manually selecting (row, col) value pairs matching each logical condition. These are very simple and observation-based exercises that can help us understand how logical condition works at

the very core level. Overall, it will help improve our logical thinking capabilities definitively, not in an unsure or guessing way.

To begin, we practice writing logical conditions manually. We try to confirm understanding of applying logical conditions on simple picture-based exercises such as alphabets of very simple structure. Yes, of course, we can verify all these logical conditions by writing programs, which will be our next level of learning. Important, each logical condition can be verified easily by writing a program using **nested loops**. Working with nested loops will be our next major step in learning. We will try to transform our action of writing alphabets manually into a set of logical conditions. We will get the countable experience from such picture-based exercises because we can learn how to apply logical conditions. The considered alphabetic pictures suggest selecting only a particular set of (row, col) value pairs and leaving the others. It will force us to think about what could be the possible accurate, logical condition. Since it's based on simple math and common sense, manually verifying these logical conditions won't be difficult. When we do such exercises to transform from a simple picture to logical conditions or vice-versa, it helps us improve our visualization of programming logic.



Different alphabets mean different pictures. So in totality, it helps us know how the certain program works to give a targeted result. Additionally, we can verify all these logical conditions by placing them in a program and executing it to verify the results. The right time has not yet come for novice /absolute beginner programmers to verify it through writing programs.

For those who are already experienced in any programming language to the extent of conditional statements and nested loops, surely they can go ahead and test their logical conditions. The first important lesson we should know is how to give proper logical conditions. We can clearly learn this important aspect of logic building by practicing different alphabets of simple structure into equivalent logical conditions.

Convert Logical Conditions to Simple Picture

We need to express logical conditions in the form of simple picture-based exercises. We need to mark those boxes with * character, which rightly fits a logical condition. A unique combination of (row, col) value pair represents each box. To understand how the selection of boxes is made using specific logical conditions are shown in the following exercises.

Exercise YB1: Express Logical condition:(row == col) in the following 5X5 box.

The condition **(row == col)** asks to find those (row, column) pairs in the **5X5** box where row value and column value are equal. We notice that the values of row and column are equal on the diagonal squares.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1	1,2	1,3	1,4	1,5
row=2	2,1	2,2	2,3	2,4	2,5
row=3	3,1	3,2	3,3	3,4	3,5
row=4	4,1	4,2	4,3	4,4	4,5
row=5	5,1	5,2	5,3	5,4	5,5

	col=1	col=2	col=3	col=4	col=5
row=1	1,1 *	1,2	1,3	1,4	1,5
row=2	2,1	2,2 *	2,3	2,4	2,5
row=3	3,1	3,2	3,3 *	3,4	3,5
row=4	4,1	4,2	4,3	4,4 *	4,5
row=5	5,1	5,2	5,3	5,4	5,5*

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*				
row=2		2,2*			
row=3			3,3*		
row=4				4,4*	
row=5					5,5*

	col=1	col=2	col=3	col=4	col=5
row=1	*				
row=2		*			
row=3			*		
row=4				*	
row=5					*

*

*

*

*

We turned an expression of logic condition into a simple picture of a diagonal line drawn using * characters. Such a targeted attempt refines our skill of applying logical conditions.

Exercise YB2. Express Logical condition:(row == 5) or (col==1) in the following 5X5 box.

Let's take one part of the logical condition (**row==5**), which means rows equal to 5. The boxes where the row value is 5 will be our selection of boxes irrespective of the value of **col**. It can be brought into notice by highlighting the border of the selected boxes.

For clear understanding, boxes are marked with thick borders where row equals 5.					
	col=1	col=2	col=3	col=4	col=5
row=1	1,1	1,2	1,3	1,4	1,5
row=2	2,1	2,2	2,3	2,4	2,5
row=3	3,1	3,2	3,3	3,4	3,5
row=4	4,1	4,2	4,3	4,4	4,5
row=5	5,1	5,2	5,3	5,4	5,5

So as per the rule of the game, we should put * character on those boxes where <u>row equals 5</u> .					
	col=1	col=2	col=3	col=4	col=5
row=1	1,1	1,2	1,3	1,4	1,5
row=2	2,1	2,2	2,3	2,4	2,5
row=3	3,1	3,2	3,3	3,4	3,5
row=4	4,1	4,2	4,3	4,4	4,5
row=5	*5,1	*5,2	*5,3	*5,4	*5,5

Similarly, let's take the other part of the logical condition, which is (**col ==1**). Again, let's select the boxes where the value of col is 1 irrespective of row value and by highlighting the selected boxes' border.

For clear understanding, boxes are marked with thick borders where col equals 1.					
	col=1	col=2	col=3	col=4	col=5
row=1	1,1	1,2	1,3	1,4	1,5
row=2	2,1	2,2	2,3	2,4	2,5
row=3	3,1	3,2	3,3	3,4	3,5
row=4	4,1	4,2	4,3	4,4	4,5
row=5	*5,1	*5,2	*5,3	*5,4	*5,5

So as per the rule of the game, we should put * character on those boxes where <u>col equals 1</u> .					
	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2	1,3	1,4	1,5
row=2	2,1*	2,2	2,3	2,4	2,5
row=3	3,1*	3,2	3,3	3,4	3,5
row=4	4,1*	4,2	4,3	4,4	4,5
row=5	*5,1	*5,2	*5,3	*5,4	*5,5

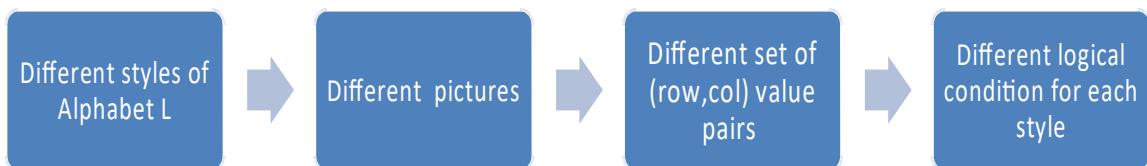
Removing (row, col) value pairs where character * is not present. The selected (row, col) value pair with character * will be like the following.					
	col=1	col=2	col=3	col=4	col=5
row=1	1,1*				
row=2	2,1*				
row=3	3,1*				
row=4	4,1*				
row=5	*5,1	*5,2	*5,3	*5,4	*5,5

Removing selected (row, col) value pair details. Removing the row and column details too. This looks similar to the English alphabet L.					
	*				
	*				
	*				
	*				
	*	*	*	*	*

We can see few different styles of alphabet L.



What possible thoughts shall cross our mind if we try to fit each different style alphabet L in 8 rows and 8 column box using * character. Different alphabet styles mean different pictures, and each will have different sets of (row, col) value pairs.



It's obvious that we can't get a pixel-perfect picture by using *character. The most important point we need to understand is that **different pictured outputs mean different logical conditions**. These pictures are somewhat advanced to start with it to learn to apply logical conditions.

We can surely start with something so simple that even school-going kids can't make any excuse to dislike programming. We will pick something which will be so easy and familiar to most of us. We will consider coding English alphabets of the very simplest structure, as shown in the following examples.

Alphabet A	Alphabet C	Alphabet D
* * * * *	* * * *	* * * *
*	*	*
*	*	*
*	*	*
*	*	*
*	*	*
*	*	*

Let's recall our learning experiences with respect to the English alphabet from our school days. In order to learn how to write alphabets, we were taught by our teachers, parents, and others. They taught us in some logical way to write alphabets. The total logic applied to writing one alphabet differs from another. Since each alphabet's entire structure is different, we have the opportunity to practice various kinds of logical conditions.

Convert Simple Picture to Logical Conditions

We will do some exercises by trying to get logical conditions from an English alphabet of very simple nature drawn using * character. We may consider 7 rows and 5 columns box. We have to select only those pairs of (row, col) values, which are helpful to make a required alphabet. We will be putting logical conditions only for those (row, col) value pairs matching * characters.

ExerciseYBA: Alphabet A needs to be fit in a 7X5 box.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1	1,2	1,3	1,4	1,5
row=2	2,1	2,2	2,3	2,4	2,5
row=3	3,1	3,2	3,3	3,4	3,5
row=4	4,1	4,2	4,3	4,4	4,5
row=5	5,1	5,2	5,3	5,4	5,5
row=6	6,1	6,2	6,3	6,4	6,5
row=7	7,1	7,2	7,3	7,4	7,5

In order to draw the alphabet A, let's start by putting * characters in the first row.

So logical condition will be **row==1** where == is equals comparator operator.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2	2,3	2,4	2,5
row=3	3,1	3,2	3,3	3,4	3,5
row=4	4,1	4,2	4,3	4,4	4,5
row=5	5,1	5,2	5,3	5,4	5,5
row=6	6,1	6,2	6,3	6,4	6,5
row=7	7,1	7,2	7,3	7,4	7,5

We will use a logical OR (or) conditional operator to add more data. In our case, we need to add some more * characters to complete one more part of the alphabet A. Now we need to put * characters in the first column and the last column. So, adding **(col==1)** and **(col==5)** logical conditions using **OR (or)** logical operator means more asterisk (*) characters are to be added in the following boxes.

The logical conditions updates are shown in the following

(row==1) or (col==1)(row==1) or (col==1) or (col==5)

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1*	2,2	2,3	2,4	2,5
row=3	3,1*	3,2	3,3	3,4	3,5
row=4	4,1*	4,2	4,3	4,4	4,5
row=5	5,1*	5,2	5,3	5,4	5,5
row=6	6,1*	6,2	6,3	6,4	6,5
row=7	7,1*	7,2	7,3	7,4	7,5*

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1*	2,2	2,3	2,4	2,5*
row=3	3,1*	3,2	3,3	3,4	3,5*
row=4	4,1*	4,2	4,3	4,4	4,5*
row=5	5,1*	5,2	5,3	5,4	5,5*
row=6	6,1*	6,2	6,3	6,4	6,5*
row=7	7,1*	7,2	7,3	7,4	7,5*

Adding one more condition (**row == 4**) to the previous logical conditions to complete the middle part of the alphabet A. Now total * characters that displays alphabet A can be shown as follows with logical conditions (**row==1 or (col==1) or (col==5) or (row==4)**).

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1*	2,2	2,3	2,4	2,5*
row=3	3,1*	3,2	3,3	3,4	3,5*
row=4	4,1*	4,2*	4,3*	4,4*	4,5*
row=5	5,1*	5,2	5,3	5,4	5,5*
row=6	6,1*	6,2	6,3	6,4	6,5*
row=7	7,1*	7,2	7,3	7,4	7,5*

Now removing the (row, col) value pairs that don't have * characters.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1*				2,5*
row=3	3,1*				3,5*
row=4	4,1*	4,2*	4,3*	4,4*	4,5*
row=5	5,1*				5,5*
row=6	6,1*				6,5*
row=7	7,1*				7,5*

Alphabet A					Logical conditions
*	*	*	*	*	(row == 1) or
*				*	(col == 1) or
*				*	(col == 5) or
*	*	*	*	*	(row == 4)
*				*	
*				*	
*				*	

Let's verify the following logical condition of alphabet A in 7 rows and 5 columns given as:
(row==1) or (col==1) or (col==5) or (row==4)

VerifyAlphabetA.py

```
TOTAL_ROWS = 7
TOTAL_COLUMNS = 5

# condition is a boolean variable for storing result of logical conditions.

for row in range(1, TOTAL_ROWS + 1):

    for col in range(1, TOTAL_COLUMNS + 1):

        condition =(row == 1) or (col == 1)
        condition = condition or (col == 5) or (row == 4)

        # space(s) are used just to show formatted output.

        if condition:

            print("*",end="")
        else:

            print(" ",end=" ")

    print()# Move to the next row.

    # Printing of characters will begin in the next new line.
```

Output

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

ExerciseYBB: Alphabet B

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2	2,3	2,4	2,5
row=3	3,1	3,2	3,3	3,4	3,5
row=4	4,1	4,2	4,3	4,4	4,5
row=5	5,1	5,2	5,3	5,4	5,5
row=6	6,1	6,2	6,3	6,4	6,5
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

In order to draw alphabet B, let's start by putting * characters in the first row and the last row. Then, the equivalent logical conditions will be using logical conditions **row==1** and **row ==7**.

The logical condition becomes

(row==1) or (row==7)

Now we need to put * characters in the second column and the last column. So, adding (col==2) and (col==5) logical conditions using OR (or) logical operator means more * characters are to be added in the following boxes. So, the updated logical conditions can be shown as the following:

(row==1) or (row==7) or (col==2) or (col==5)

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2*	2,3	2,4	2,5*
row=3	3,1	3,2*	3,3	3,4	3,5*
row=4	4,1	4,2*	4,3	4,4	4,5*
row=5	5,1	5,2*	5,3	5,4	5,5*
row=6	6,1	6,2*	6,3	6,4	6,5*
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

Adding one more condition **(row ==4)**. One drawback with this logical condition is that

we don't want * character in the (row == 4), especially in the box with value pair (4, 1).

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2*	2,3	2,4	2,5*
row=3	3,1	3,2*	3,3	3,4	3,5*
row=4	4,1*	4,2*	4,3*	4,4*	4,5*
row=5	5,1	5,2*	5,3	5,4	5,5*
row=6	6,1	6,2*	6,3	6,4	6,5*
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

We apply logical AND (and) conditional operators to select fewer data. In our case, we need to select one * character less in the **(row == 4)** to complete the middle part of the alphabet B.

The updated logical condition for the middle part of the alphabet B will be **(row ==4 and col>=2)**.

Now the logical condition becomes

(row==1) or (row==7) or (col==2) or (col==5) or (row ==4 and col>=2)

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2*	2,3	2,4	2,5*
row=3	3,1	3,2*	3,3	3,4	3,5*
row=4	4,1	4,2*	4,3*	4,4*	4,5*
row=5	5,1	5,2*	5,3	5,4	5,5*
row=6	6,1	6,2*	6,3	6,4	6,5*
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

Now removing the (row, col) value pairs that don't have * characters.

	col=1	col=2	col=3	col=4	col=5	
row=1	1,1*	1,2*	1,3*	1,4*	1,5*	*
row=2		2,2*			2,5*	*
row=3		3,2*			3,5*	*
row=4		4,2*	4,3*	4,4*	4,5*	OR *
row=5		5,2*			5,5*	* *
row=6		6,2*			6,5*	* *
row=7	7,1*	7,2*	7,3*	7,4*	7,5*	* * * *

The complete logical condition becomes

(row==1) or (row==7) or (col==2) or (col==5) or (row ==4 and col>=2)

Let's verify following logical condition of alphabet B in 7 rows and 5 columns given as:
(row==1) or (row==7) or (col==2) or (col==5) or (row ==4 and col>=2)

VerifyAlphabetB.py

```
TOTAL_ROWS = 7
TOTAL_COLUMNS = 5

# condition is a boolean variable for storing result of logical conditions.

for row in range(1, TOTAL_ROWS + 1):

    for col in range(1, TOTAL_COLUMNS + 1):

        condition =(row == 1) or (row == 7)
        condition = condition or (col == 2) or (col == 5)
        condition = condition or (row == 4 and col >= 2)

        # space(s) are used just to show formatted output.

        if condition:

            print("*",end="")# single space

        else:

            print("",end="")# double space

print()# Move to the next row.

# Printing of characters will begin in the next new line.
```

Output

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

ExerciseYBC: Alphabet C

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2	2,3	2,4	2,5
row=3	3,1	3,2	3,3	3,4	3,5
row=4	4,1	4,2	4,3	4,4	4,5
row=5	5,1	5,2	5,3	5,4	5,5
row=6	6,1	6,2	6,3	6,4	6,5
row=7	7,1	7,2	7,3	7,4	7,5

On observing alphabet C and trying to fit into the box of size 7X5, we can start putting * characters in the first row. Now we need to put * characters in the first row. So logical condition will become **row==1**.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1*	2,2	2,3	2,4	2,5
row=3	3,1*	3,2	3,3	3,4	3,5
row=4	4,1*	4,2	4,3	4,4	4,5
row=5	5,1*	5,2	5,3	5,4	5,5
row=6	6,1*	6,2	6,3	6,4	6,5
row=7	7,1*	7,2	7,3	7,4	7,5

The equivalent logical condition for the first column will be **(col==1)** condition. Using **logical OR (or) conditional operator** to combine logical conditions becomes **(row==1) or (col==1)**

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1*	2,2	2,3	2,4	2,5
row=3	3,1*	3,2	3,3	3,4	3,5
row=4	4,1*	4,2	4,3	4,4	4,5
row=5	5,1*	5,2	5,3	5,4	5,5
row=6	6,1*	6,2	6,3	6,4	6,5
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

Adding one more logical condition **(row==7)** to the previous. Now the * character set resembling the alphabet C and complete logical conditions can be:
(row==1) or (col==1) or (row==7)

Now removing the (row, col) value pair which doesn't have * character. Hence, the Logical condition:(row==1) or (col==1) or (row==7)

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1*				
row=3	3,1*				
row=4	4,1*				
row=5	5,1*				
row=6	6,1*				
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

OR

	col=1	col=2	col=3	col=4	col=5
row=1	*	*	*	*	*
row=2	*				
row=3	*				
row=4	*				
row=5	*				
row=6	*				
row=7	*	*	*	*	*

Let's verify the following logical condition of alphabet C in 7 rows and 5 columns given as:
(row==1) or (col==1) or (row==7)

VerifyAlphabetC.py

```
TOTAL_ROWS = 7
TOTAL_COLUMNS = 5
```

```
# condition is a boolean variable for storing result of logical conditions.

for row in range(1, TOTAL_ROWS + 1):

    for col in range(1, TOTAL_COLUMNS + 1):

        condition =(row == 1) or (col == 1) or (row == 7)

        # space(s) are used just to show formatted output.

        if condition:

            print("*",end="")# single space

        else:

            print("",end="")# double space

print()# Move to the next row.

# Printing of characters will begin in the next new line.
```

Output

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

ExerciseYBD: Alphabet D

We need to put * characters in the first row and the last row. So matching logical conditions will be (row==1) and (row==7).

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

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2*	2,3	2,4	2,5*
row=3	3,1	3,2*	3,3	3,4	3,5*
row=4	4,1	4,2*	4,3	4,4	4,5*
row=5	5,1	5,2*	5,3	5,4	5,5*
row=6	6,1	6,2*	6,3	6,4	6,5*
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2*	2,3	2,4	2,5*
row=3	3,1	3,2*	3,3	3,4	3,5*
row=4	4,1	4,2*	4,3	4,4	4,5*
row=5	5,1	5,2*	5,3	5,4	5,5*
row=6	6,1	6,2*	6,3	6,4	6,5*
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

Now putting * characters in the second column and the last column will complete the alphabet D. The logical conditions columns will be (**col==2**) and (**col==5**). Total logical conditions becomes (**row==1**) or (**row==7**) or (**col==2**) or (**col==5**)

Removing all the (row, col) value pair that doesn't have * characters.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2*	2,3	2,4	2,5*
row=3	3,1	3,2*	3,3	3,4	3,5*
row=4	4,1	4,2*	4,3	4,4	4,5*
row=5	5,1	5,2*	5,3	5,4	5,5*
row=6	6,1	6,2*	6,3	6,4	6,5*
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

OR

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

Let's verify the following logical condition of alphabet D in 7 rows and 5 columns given as:
(row==1) or (row==7) or (col==2) or (col==5)

VerifyAlphabetD.py

```
TOTAL_ROWS = 7
TOTAL_COLUMNS = 5

# condition is a boolean variable for storing result of logical conditions.

for row in range(1, TOTAL_ROWS + 1):

    for col in range(1, TOTAL_COLUMNS + 1):

        condition =( row == 1 ) or ( row == 7 )

        condition = condition or ( col == 2 ) or ( col == 5 )

        # space(s) are used just to show formatted output.

        if condition:

            print("*",end="")# single space

        else:

            print("",end="")# double space

    print()# Move to the next row.

    # Printing of characters will begin in the next new line.
```

Output

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

Exercise YBE: Alphabet E

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2	2,3	2,4	2,5
row=3	3,1	3,2	3,3	3,4	3,5
row=4	4,1	4,2	4,3	4,4	4,5
row=5	5,1	5,2	5,3	5,4	5,5
row=6	6,1	6,2	6,3	6,4	6,5
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

We need to put * characters in the first row and the last row.

So, matching logical conditions will be **(row==1) and (row==7)**.

Also, putting * characters in the second column will complete the alphabet E.

On adding logical condition (col==2), the total logical conditions becomes

(row==1) or (row==7) or (col==2)

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2*	2,3	2,4	2,5
row=3	3,1	3,2*	3,3	3,4	3,5
row=4	4,1	4,2*	4,3	4,4	4,5
row=5	5,1	5,2*	5,3	5,4	5,5
row=6	6,1	6,2*	6,3	6,4	6,5
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

Adding one more condition

(row==4 and col>=2) to the previous conditions.

The AND logical condition **(and col>=2)** is because we don't want to put * character in the box with value pair **(4, 1)**.

The AND logical condition is applied to restrict selecting data, and our data are **(row, col)** value pairs, the boxes in general.

The total logical conditions becomes **(row==1) or (row==7) or (col==2) or (row ==4 and col>=2)**

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2*	2,3	2,4	2,5
row=3	3,1	3,2*	3,3	3,4	3,5
row=4	4,1	4,2*	4,3*	4,4*	4,5*
row=5	5,1	5,2*	5,3	5,4	5,5
row=6	6,1	6,2*	6,3	6,4	6,5
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

OR

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

Let's verify the following logical condition of alphabet E in 7 rows and 5 columns given as:
(row==1) or (row==7) or (col==2) or (row ==4 and col>=2)

VerifyAlphabetE.py

```
TOTAL_ROWS = 7
TOTAL_COLUMNS = 5

# condition is a boolean variable for storing result of logical conditions.

for row in range(1, TOTAL_ROWS + 1):

    for col in range(1, TOTAL_COLUMNS + 1):

        condition =(row == 1) or (row == 7)
        condition = condition or (col == 2) or (row == 4 and col >= 2)

        # space(s) are used just to show formatted output.

        if condition:

            print("*",end="")
# single space

        else:

            print(" ",end="")
# double space

print()# Move to the next row.
# Printing of characters will begin in the next new line.
```

Output

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

ExerciseYBF: Alphabet F

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2*	2,3	2,4	2,5
row=3	3,1	3,2*	3,3	3,4	3,5
row=4	4,1	4,2*	4,3	4,4	4,5
row=5	5,1	5,2*	5,3	5,4	5,5
row=6	6,1	6,2*	6,3	6,4	6,5
row=7	7,1	7,2*	7,3	7,4	7,5

We need to put * characters in the first row and the second column.

So, matching logical conditions will be **(row==1) and (col==2)**. The logical conditions for now becomes;

(row==1) or (col==2).

Adding one more condition **(row ==4 and col>=2)** to the previous conditions. The AND logical condition **(and col>=2)** is because we don't want to put * character in the box with value pair **(4, 1)**.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2*	2,3	2,4	2,5
row=3	3,1	3,2*	3,3	3,4	3,5
row=4	4,1	4,2*	4,3*	4,4*	4,5*
row=5	5,1	5,2*	5,3	5,4	5,5
row=6	6,1	6,2*	6,3	6,4	6,5
row=7	7,1	7,2*	7,3	7,4	7,5

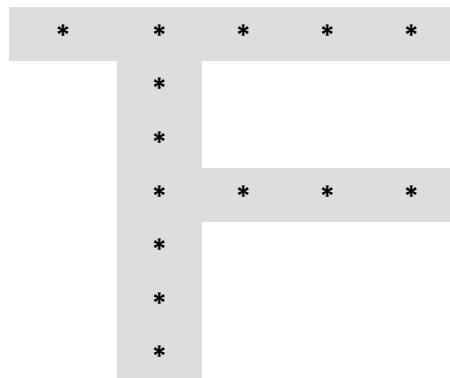
Now the collection of * characters that makes alphabet F can be given by the logical conditions, which are as follows:

**(row==1) or (col==2) or
(row ==4 and col>=2)**.

Removing all the (row, col) value pair that doesn't have * characters.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2		2,2*			
row=3		3,2*			
row=4		4,2*	4,3*	4,4*	4,5*
row=5		5,2*			
row=6		6,2*			
row=7		7,2*			

OR



Let's verify the following logical condition of alphabet F in 7 rows and 5 columns given as:
(row==1) or (col==2) or (row ==4 and col>=2)

VerifyAlphabetF.py

```
TOTAL_ROWS = 7
TOTAL_COLUMNS = 5

# condition is a boolean variable for storing result of logical conditions.

for row in range(1, TOTAL_ROWS + 1):

    for col in range(1, TOTAL_COLUMNS + 1):

        condition =(row == 1) or (col == 2)
        condition = condition or (row == 4 and col >= 2)

        # space(s) are used just to show formatted output.

        if condition:

            print("*",end="")
        else:

            print(" ",end=" ")

    print()# Move to the next row.

    # Printing of characters will begin in the next new line.
```

Output

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

Exercise YBG: Alphabet G

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2	2,3	2,4	2,5
row=3	3,1	3,2	3,3	3,4	3,5
row=4	4,1	4,2	4,3	4,4	4,5
row=5	5,1	5,2	5,3	5,4	5,5
row=6	6,1	6,2	6,3	6,4	6,5
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

We need to put * characters in the first row and the last row. So, matching logical conditions will be (**row==1**) and (**row==7**).

Also, we need to put * characters in the first column. So, the matching logical condition will be (**col==1**).

The logical conditions become
(**row==1**) or (**row==7**) or (**col==1**)

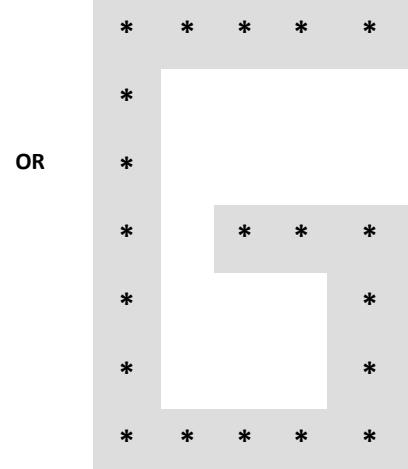
Adding one more condition (**row == 4 and col>=3**) to the previous condition to exclude * character in the box with value pair (**4, 2**). We don't want * character in the box with value pairs (**2, 5**) and (**3, 5**). Once again, adding one more condition (**col == 5 and row >=5**) to the previous condition.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1*	2,2	2,3	2,4	2,5
row=3	3,1*	3,2	3,3	3,4	3,5
row=4	4,1*	4,2	4,3	4,4	4,5
row=5	5,1*	5,2	5,3	5,4	5,5
row=6	6,1*	6,2	6,3	6,4	6,5
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1*	2,2	2,3	2,4	2,5
row=3	3,1*	3,2	3,3	3,4	3,5
row=4	4,1*	4,2	4,3*	4,4*	4,5*
row=5	5,1*	5,2	5,3	5,4	5,5*
row=6	6,1*	6,2	6,3	6,4	6,5*
row=7	7,1*	7,2*	7,3*	7,4*	7,5*

Now the complete collection of * characters along with logical conditions can be shown as follows (**row==1**) or (**row==7**) or (**col==1**) or (**row ==4 and col>=3**) or (**col==5 and row>=5**). Removing all the (row, col) value pair that doesn't have * characters.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1*				
row=3	3,1*				
row=4	4,1*		4,3*	4,4*	4,5*
row=5	5,1*				5,5*
row=6	6,1*				6,5*
row=7	7,1*	7,2*	7,3*	7,4*	7,5*



Let's verify the following logical condition of alphabet G in 7 rows and 5 columns given as:
(row==1) or (row==7) or (col==1) or (row ==4 and col>=3) or (col==5 and row>=5)

VerifyAlphabetG.py

```
TOTAL_ROWS = 7
TOTAL_COLUMNS = 5

# condition is a boolean variable for storing result of logical conditions.

for row in range(1, TOTAL_ROWS + 1):

    for col in range(1, TOTAL_COLUMNS + 1):

        condition =(row == 1) or (row == 7) or (col == 1)
        condition = condition or (row == 4 and col >= 3)
        condition = condition or (col == 5 and row >= 5)

        # space(s) are used just to show formatted output.

        if condition:

            print("*",end="")
        else:

            print(" ",end=" ")

    print()# Move to the next row.

    # Printing of characters will begin in the next new line.
```

Output

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

ExerciseYBJ: Alphabet J

Let fill * characters in the first row & the middle column. So, matching logical conditions will be **(row==1) or (col==3)**. We have used the logical OR (or), conditional operator, to add more * characters.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2	2,3*	2,4	2,5
row=3	3,1	3,2	3,3*	3,4	3,5
row=4	4,1	4,2	4,3*	4,4	4,5
row=5	5,1	5,2	5,3*	5,4	5,5
row=6	6,1	6,2	6,3*	6,4	6,5
row=7	7,1	7,2	7,3*	7,4	7,5

Since we have to restrict column count to 3 at row=7, we have to use conditional operator **logical AND (and col <= 3)**. Adding logical condition **(row==7 and col<=3)** to previous conditions using logical OR (or) conditional operator will give * character set can be shown in the box. The logical condition becomes;

**(row==1) or (col==3) or
(row==7 and col<=3)**

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2	2,3*	2,4	2,5
row=3	3,1	3,2	3,3*	3,4	3,5
row=4	4,1	4,2	4,3*	4,4	4,5
row=5	5,1	5,2	5,3*	5,4	5,5
row=6	6,1	6,2	6,3*	6,4	6,5
row=7	7,1*	7,2*	7,3*	7,4	7,5

Again, adding a condition to previous **(col==1 and row>=4)**. The selected * character set with a thick border represents condition **(col==1 and row>=4)**.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2	2,1	2,2	2,3*	2,4	2,5
row=3	3,1	3,2	3,3*	3,4	3,5
row=4	4,1*	4,2	4,3*	4,4	4,5
row=5	5,1*	5,2	5,3*	5,4	5,5
row=6	6,1*	6,2	6,3*	6,4	6,5
	7,1*	7,2*	7,3*	7,4	7,5

Please Pay Attention

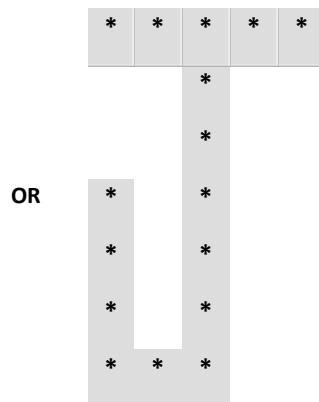
Using logical AND, we have restricted or limited the placing of * character at col=1. The condition is **(col==1 and row>=4)**.

If the condition were col==1 only, then the complete col=1 would have been filled with * characters.

The additional condition **and row>=4** in the expression **(col==1 and row>=4)** actually limited the * characters at column 1 started from row 4 till row 7.

Removing the (row, col) value pair which doesn't have * characters.

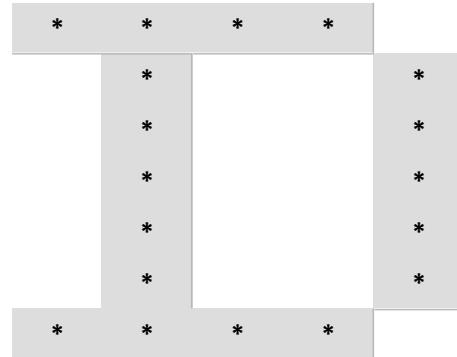
	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2*	1,3*	1,4*	1,5*
row=2			2,3*		
row=3			3,3*		
row=4	4,1*		4,3*		
row=5	5,1*		5,3*		
row=6	6,1*		6,3*		
row=7	7,1*	7,2*	7,3*		



Logical condition:(row==1) or (col==3) or (row==7 and col<=3) or (col==1 and row>=4).

Problem YBD: Write the program for the following style of alphabet J and D.

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



ExerciseYBK: Alphabet K

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2	1,3	1,4	1,5
row=2	2,1*	2,2	2,3	2,4	2,5
row=3	3,1*	3,2	3,3	3,4	3,5
row=4	4,1*	4,2	4,3	4,4	4,5
row=5	5,1*	5,2	5,3	5,4	5,5
row=6	6,1*	6,2	6,3	6,4	6,5
row=7	7,1*	7,2	7,3	7,4	7,5

We need to put * characters in the first row. So, matching logical conditions will be **(row==1)**. We need to find a relationship among highlighted boxes as it suggests one of the arms of the alphabet K.

The highlighted boxes are represented by **(row, col)** value pairs as **(4,1),(3,2),(2,3), (1,4)**.

If we pay attention, we can observe that if we add row value/number to column value/number, we will get value 5 consistently.

$$(4,1),(3,2),(2,3),(1,4) \rightarrow (4+1),(3+2),(2+3),(1+4) \rightarrow (5),(5),(5),(5).$$

We need to verify if this holds True for other **row+col** value combinations or not. Let's check by adding each **row+col** value for every box represented by **(row, col)** value pair. All **row+col** values are as follows and highlighted where **row+col=5**.

$$\begin{aligned} &(1+1=2)(1+2=3)(1+3=4) \textbf{(1+4=5)}(1+5=6) \\ &(2+1=3)(2+2=4) \textbf{(2+3=5)}(2+4=6)(2+5=7) \\ &(3+1=4) \textbf{(3+2=5)}(3+3=6)(3+4=7)(3+5=8) \\ &\textbf{(4+1=5)}(4+2=6)(4+3=7)(4+4=8)(4+5=9) \\ &(5+1=6)(5+2=7)(5+3=8)(5+4=9)(5+5=10) \\ &(6+1=7)(6+2=8)(6+3=9)(6+4=10)(6+5=11) \\ &(7+1=8)(7+2=9)(7+3=10)(7+4=11)(7+5=12) \end{aligned}$$

Nowhere **row + col = 5** holds True in other than our considered boxes. So, our guess now becomes valid, which uniquely identifies or selects the considered value pairs. This how we make a selection of general data by applying conditions. In this case, boxes with **(row, col)** combinations are selected. Hence our next logical condition becomes **(row+col == 5)**.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2	1,3	1,4*	1,5
row=2	2,1*	2,2	2,3*	2,4	2,5
row=3	3,1*	3,2*	3,3	3,4	3,5
row=4	4,1*	4,2	4,3	4,4	4,5
row=5	5,1*	5,2	5,3	5,4	5,5
row=6	6,1*	6,2	6,3	6,4	6,5
row=7	7,1*	7,2	7,3	7,4	7,5

So logical condition becomes

$$\textbf{(col == 1) or (row+col==5)}.$$

Similarly, we need to find the relationship between **(4, 1),(5, 2),(6, 3),(7, 4)** boxes, the other arm of the alphabet K shown by highlighted cell border for row value more than 4.

Previously, the addition of **(row +col)** worked, and this time subtraction may work. Let's study this case

$$(4,1),(5,2),(6,3),(7,4) \rightarrow (4-1),(5-2),(6-3),(7-4) \rightarrow (3),(3),(3),(3).$$

It's giving consistent results, and we need to verify if any other **(row, col)** combination too follows this rule of selection or not. If we find any other **(row, col)**, we need to find some other relationship formula. The selection rule or logical condition should uniquely select **(4,1),(5,2),(6,3),(7,4)**. Let's check by subtracting each **row - col** value. All **(row - col)** values are shown as follows, and highlighted ones follow the rule **(row - col=3)**.

$$\begin{aligned}
 &(1-1=0)(1-2=-1)(1-3=-2)(1-4=-3)(1-5=-4) \\
 &(2-1=1)(2-2=0)(2-3=-1)(2-4=-2)(2-5=-3) \\
 &(3-1=2)(3-2=1)(3-3=0)(3-4=-1)(3-5=-2) \\
 &\mathbf{(4-1=3)}(4-2=2)(4-3=1)(4-4=0)(4-5=-1) \\
 &(5-1=4)\mathbf{(5-2=3)}(5-3=2)(5-4=1)(5-5=0) \\
 &(6-1=5)(6-2=4)\mathbf{(6-3=3)}(6-4=2)(6-5=1) \\
 &(7-1=6)(7-2=5)(7-3=4)\mathbf{(7-4=3)}(7-5=2)
 \end{aligned}$$

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2	1,3	1,4*	1,5
row=2	2,1*	2,2	2,3*	2,4	2,5
row=3	3,1*	3,2*	3,3	3,4	3,5
row=4	4,1*	4,2	4,3	4,4	4,5
row=5	5,1*	5,2*	5,3	5,4	5,5
row=6	6,1*	6,2	6,3*	6,4	6,5
row=7	7,1*	7,2	7,3	7,4*	7,5

Nowhere **row - col = 3** holds True in other than our considered value pairs. So, our guess now becomes valid to select these boxes under consideration.

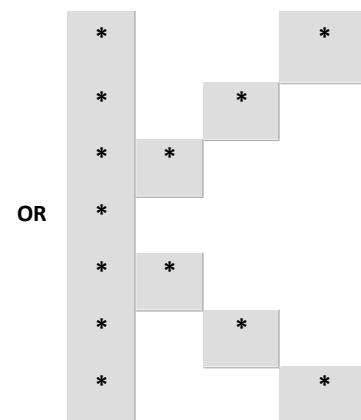
Hence our next logical condition becomes

(row - col == 3). So final logical condition becomes

(col==1) or (row+col==5) or (row - col == 3).

Removing the **(row, col)** value pair where * character is not present, we will get as follows

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*			1,4*	
row=2	2,1*		2,3*		
row=3	3,1*	3,2*			
row=4	4,1*				
row=5	5,1*	5,2*			
row=6	6,1*		6,3*		
row=7	7,1*			7,4*	



We have seen in this case addition and subtraction relationships among the row, and column values worked. Sometimes other arithmetic relationship like division or remainder among row and column value holds True to select the boxes means **(row, col)** value pairs. The logical condition can be implemented in many different ways. Different alternative ways of giving conditions for the same set of a selection of **(row, col)** value pairs could be possible.

Let's verify the following logical condition of alphabet K in 7 rows and 5 columns given as:
(col ==1) or (row + col == 5) or (row - col == 3)

VerifyAlphabetK.py

```
TOTAL_ROWS = 7
TOTAL_COLUMNS = 5

for row in range(1, TOTAL_ROWS + 1):
    for col in range(1, TOTAL_COLUMNS + 1):

        # condition is a boolean variable for storing result of logical conditions.
        condition =( col == 1) or ( row + col == 5 )
        condition = condition or ( row - col == 3 )

        # space(s) are used just to show formatted output.

        if condition:
            print("*",end="")
        else:
            print(" ",end=" ")

    print()# Move to the next row.

    # Printing of characters will begin in the next new line.
```

Output

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

Exercise YBM: Alphabet M needs to be fit in a 7X5 box.

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

	col=1	col=2	col=3	col=4	col=5
row=1	1,1	1,2	1,3	1,4	1,5
row=2	2,1	2,2	2,3	2,4	2,5
row=3	3,1	3,2	3,3	3,4	3,5
row=4	4,1	4,2	4,3	4,4	4,5
row=5	5,1	5,2	5,3	5,4	5,5
row=6	6,1	6,2	6,3	6,4	6,5
row=7	7,1	7,2	7,3	7,4	7,5

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2	1,3	1,4	1,5*
row=2	2,1*	2,2	2,3	2,4	2,5*
row=3	3,1*	3,2	3,3	3,4	3,5*
row=4	4,1*	4,2	4,3	4,4	4,5*
row=5	5,1*	5,2	5,3	5,4	5,5*
row=6	6,1*	6,2	6,3	6,4	6,5*
row=7	7,1*	7,2	7,3	7,4	7,5*

We need to find a relationship among highlighted boxes which are represented by **(row, col)** value pairs as **(1, 1),(2, 2),(3, 3).**

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2	1,3	1,4	1,5*
row=2	2,1*	2,2	2,3	2,4	2,5*
row=3	3,1*	3,2	3,3	3,4	3,5*
row=4	4,1*	4,2	4,3	4,4	4,5*
row=5	5,1*	5,2	5,3	5,4	5,5*
row=6	6,1*	6,2	6,3	6,4	6,5*
row=7	7,1*	7,2	7,3	7,4	7,5*

Let start with putting * characters in the first column & the last column.

So, matching logical conditions will be: **(col==1) or (col==5).**

Let's try to understand the relationship that uniquely selects these specific sets of row and column value pairs.

It's easy to observe that row value/number and column value/number are the same. So, we can express our selection of boxes by logical condition **(row==col).**

The logical condition **(row==col)** will select (4, 4) and (5, 5) also.

Box (5, 5) has already been selected by our previous condition.

We don't want the box **(4, 4)** to be get selected in our case. We have to select till value pair **(3, 3)** starting from the value pair **(1, 1)**. We have to use the logical AND operator to apply the restriction in the selection of boxed and logical condition then can be given as follows

(row==col) and (row <=3 and col<=3)

These selections of boxes till now can be expressed by the logical condition

(col==1) or (col==5) or ((row==col) and (row <=3 and col<=3))

Again, we need to find a relationship among given highlighted boxes in order to uniquely select them out all the given boxes and which are represented by (row, col) value pairs as **(3, 3),(2, 4),(1, 5)**.

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2	1,3	1,4	1,5*
row=2	2,1*	2,2*	2,3	2,4	2,5*
row=3	3,1*	3,2	3,3*	3,4	3,5*
row=4	4,1*	4,2	4,3	4,4	4,5*
row=5	5,1*	5,2	5,3	5,4	5,5*
row=6	6,1*	6,2	6,3	6,4	6,5*
row=7	7,1*	7,2	7,3	7,4	7,5*

Let's try to understand the relationship that uniquely selects these specific set of row and column value pairs which are as follows

→(3,3),(2,4),(1,5) or

→(3+3),(2+4),(1+5) or

→(6),(6),(6)

If we pay a little bit of attention, we can observe that if we add row value/number to column value/number, we will get value 6 consistently. We need to verify if this holds True for other **row+col** value combinations or not. Let's check by adding each **row+col** value for every box represented by **(row, col)** value pair. All **row+col values** are as follows, and highlighted ones were that follows **row+col=6**.

	col=1	col=2	col=3	col=4	col=5
row=1	1+1=2	1+2=3	1+3=4	1+4=5	1+5=6
row=2	2+1=3	2+2=4	2+3=5	2+4=6	2+5=7
row=3	3+1=4	3+2=5	3+3=6	3+4=7	3+5=8
row=4	4+1=5	4+2=6	4+3=7	4+4=8	4+5=9
row=5	5+1=6	5+2=7	5+3=8	5+4=9	5+5=10
row=6	6+1=7	6+2=8	6+3=9	6+4=10	6+5=11
row=7	7+1=8	7+2=9	7+3=10	7+4=11	7+5=12

	col=1	col=2	col=3	col=4	col=5
row=1	1,1*	1,2	1,3	1,4	1,5*
row=2	2,1*	2,2*	2,3	2,4*	2,5*
row=3	3,1*	3,2	3,3*	3,4	3,5*
row=4	4,1*	4,2	4,3	4,4	4,5*
row=5	5,1*	5,2	5,3	5,4	5,5*
row=6	6,1*	6,2	6,3	6,4	6,5*
row=7	7,1*	7,2	7,3	7,4	7,5*

Now, **row+col=6** also holds True for the value pair **(4, 2)**, and we don't want to put character * over there. We have to restrict it to value pair **(3, 3)**. The overall logical condition specifically for this can be given as **(row+col ==6) and (row<=3 and col>=3)**.

Removing the other (row, col) value pairs, and we get the alphabet M.

	col=1	col=2	col=3	col=4	col=5	
row=1	1,1*				1,5*	*
row=2	2,1*	2,2*		2,4*	2,5*	*
row=3	3,1*		3,3*		3,5*	*
row=4	4,1*				4,5*	*
row=5	5,1*				5,5*	*
row=6	6,1*				6,5*	*
row=7	7,1*				7,5*	*

OR

*		*	*	*	*	*
---	--	---	---	---	---	---

The complete selection of boxes now can be expressed by the logical condition

(col==1) or (col==5) or

((row==col) and(row <=3 and col<=3)) or

((row+col ==6) and(row<=3 and col>=3))

In the logical condition **((row==col) and(row <=3 and col<=3))**, due to logical condition **(row==col)** the part of the condition given by **(row <=3 and col<=3)** is either **row<=3 or col<=3** is enough.

This could be further given by

((row==col) and row <=3) or ((row == col) and col<=3).

(row, col) value pair	row value	col value	Description
(1, 5)	1	5	
(2, 4)	2	4	
(3, 3)	3	3	
(4, 2)	4	2	
(5, 1)	5	1	In the logical condition ((row+col ==6) and(row<=3 and col>=3)) , every (row, col) value pair in (row+col ==6) are given by (5, 1),(4, 2),(3, 3),(2, 4),(1, 5) .

Every row value is unique within their group of values and is similar to col values. Hence, considering either row value or col value is sufficient. It is unnecessary to consider both row and col values to apply restriction conditions as given by **and (row <=3 and col>=3)**.

So, part of the total logical condition which is given by **(row <=3 and col>=3)**, considering either **row<=3 or col>=3** is enough. This could be further given by

((row+col ==6) and row <=3) or ((row+col ==6) and col>=3).

We can rewrite the actual logical condition as

(col==1) or (col==5) or

((row==col) and row <=3) or

((row+col ==6) and row<=3)

Please notice the restriction condition **and row <=3** are the same for both logical condition expressions.

We can again rewrite logical condition in a more compact form and can be given as follows:

(col == 1) or (col == 5) or

(((row == col) or (row + col == 6)) and row <= 3)

Let's verify the following logical condition of alphabet M in 7 rows and 5 columns given as:

(col == 1) or (col == 5) or
(((row == col) or (row + col == 6)) and row <= 3)

VerifyAlphabetM.py

```
TOTAL_ROWS = 7
TOTAL_COLUMNS = 5

# condition is a boolean variable for storing result of logical conditions.

for row in range(1, TOTAL_ROWS + 1):

    for col in range(1, TOTAL_COLUMNS + 1):

        condition =(col == 1) or (col == 5)
        condition1 =((row == col) or (row + col == 6))
        condition1 = condition1 and row <= 3
        condition = condition or condition1

        # space(s) are used just to show formatted output.

        if condition:
            print("*",end="")
        else:
            print(" ",end=" ")

    print()# Move to the next row.

    # Printing of characters will begin in the next new line.
```

Output

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

Correct the given logical conditions.

Write a program to update the given logical conditions to match the given output.

ProblemYBM1

(col==1) or (col==5) or
 ((row==col) and (row <=3 and col<=3)) or
 ((row+col ==6) and (row<=3 and col>=3))

	col=1	col=2	col=3	col=4	col=5
row=1	0				
row=2	0	0			
row=3	0		0		
row=4	0	0			
row=5	0				
row=6	0				
row=7	0				

ProblemYBM2

(col==1) or (col==5) or
 ((row==col) and (row <=3 and col<=3)) or
 ((row+col ==6) and (row<=3 and col>=3))

	col=1	col=2	col=3	col=4	col=5
row=1	0				0
row=2	0	0		0	0
row=3	0		0		0
row=4	0	0		0	0
row=5	0				0
row=6	0				0
row=7	0				0

ProblemYBM3

(col==1) or (col==5) or
 ((row==col) and (row <=3 and col<=3)) or
 ((row+col ==6) and (row<=3 and col>=3))

	col=1	col=2	col=3	col=4	col=5
row=1	0				0
row=2		0		0	
row=3			00		
row=4		0	00	0	
row=5	0		00		0
row=6			00		
row=7			00		

ProblemYBM4

(col==1) or (col==5) or
 ((row==col) and (row <=3 and col<=3)) or
 ((row+col ==6) and (row<=3 and col>=3))

	col=1	col=2	col=3	col=4	col=5
row=1	\$\$		0		\$\$
row=2		\$\$	0	\$\$	
row=3	0	0	\$\$	0	0
row=4		\$\$	0	\$\$	
row=5	\$\$		0		\$\$
row=6			0		
row=7			0		

ProblemYBM5

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

	col=1	col=2	col=3	col=4	col=5
row=1			1,3*		
row=2			2,3*	2,4*	
row=3	3,1*		3,3*		3,5*
row=4		4,2*	4,3*	4,4*	
row=5			5,3*		
row=6		6,2*	6,3*	6,4*	
row=7	7,1*		7,3*		7,5*
row=8			8,3*	8,4*	
row=9			9,3*		

ProblemYBN

	col=1	col=2	col=3	col=4	col=5
row=1	*				*
row=2	*	*			*
row=3	*	*	*		*
row=4	*	*	*	*	*
row=5	*		*	*	*
row=6	*			*	*
row=7	*				*

ProblemYBP

	col=1	col=2	col=3	col=4	col=5
row=1		*	*	*	*
row=2		*			*
row=3		*			*
row=4		*	*	*	*
row=5		*			
row=6		*			
row=7		*			

ProblemYBR

	col=1	col=2	col=3	col=4	col=5
row=1	*	*	*	*	
row=2		*		*	
row=3	*			*	
row=4		*	*	*	
row=5	*	*			
row=6	*			*	
row=7	*				*

ProblemYBS

	col=1	col=2	col=3	col=4	col=5
row=1	*	*	*	*	*
row=2	*	*			
row=3	*	*			
row=4	*	*		*	*
row=5					*
row=6	*	*	*	*	*
row=7	*	*	*	*	*

Task Y

	col=1	col=2	col=3	col=4	col=5
row=1	XX				XX
row=2	XX	XX		XX	XX
row=3		XX	XXX	XX	
row=4			\//\		
row=5			or		
row=6			or		
row=7			or		

Task 9

	col=1	col=2	col=3	col=4	col=5
row=1			99	99	
row=2			99		99
row=3			99		99
row=4			99	99	99
row=5				99	
row=6			99		
row=7		99			

Task 5

	col=1	col=2	col=3	col=4	col=5
row=1	5	5	5	5	5
row=2	5	5			
row=3	5	5	5		
row=4	5			5	
row=5				5	
row=6				5	
row=7	5	5	5		

Task 8

	col=1	col=2	col=3	col=4	col=5
row=1	8	or		8	8
row=2	8	or		or	8
row=3	8	or		or	8
row=4	@	@		@	@
row=5	or	8		8	or
row=6	or	8		8	or
row=7	or	8		8	or

Task 5

	col=1	col=2	col=3	col=4	col=5
row=1	5	5	5	5	5
row=2	5				
row=3	5	5	5	5	
row=4				5	5
row=5				5	5
row=6	5			5	5
row=7		5	5	5	

Task 6

	col=1	col=2	col=3	col=4	col=5
row=1				6	
row=2			6		
row=3		6	6		
row=4	6	6	6	6	
row=5	6			6	
row=6	6			6	
row=7		6	6		

Task Y

	col=1	col=2	col=3	col=4	col=5
row=1	XX				XX
row=2	XX	XX		XX	XX
row=3		XX	XXX	XX	
row=4			\//\		
row=5			or		
row=6			or		
row=7			or		

Task 9

	col=1	col=2	col=3	col=4	col=5
row=1			99	99	
row=2			99		99
row=3			99		99
row=4			99	99	99
row=5				99	
row=6			99		
row=7		99			

Task 5

	col=1	col=2	col=3	col=4	col=5
row=1	5	5	5	5	5
row=2	5	5			
row=3	5	5	5		
row=4	5			5	
row=5				5	
row=6				5	
row=7	5	5	5		

Task 8

	col=1	col=2	col=3	col=4	col=5
row=1	8	or		8	8
row=2	8	or		or	8
row=3	8	or		or	8
row=4	@	@		@	@
row=5	or	8		8	or
row=6	or	8		8	or
row=7	or	8		8	or

Task 5

	col=1	col=2	col=3	col=4	col=5
row=1	5	5	5	5	5
row=2	5				
row=3	5	5	5	5	
row=4				5	5
row=5				5	5
row=6	5			5	5
row=7		5	5	5	

Task 6

	col=1	col=2	col=3	col=4	col=5
row=1				6	
row=2			6		
row=3		6	6		
row=4	6	6	6	6	
row=5	6			6	
row=6	6			6	
row=7		6	6		

Task 7

	col=1	col=2	col=3	col=4	col=5
row=1	7	7	7	7	7
row=2					7
row=3					7
row=4					7
row=5					7
row=6					7
row=7					7

Task LT

	col=1	col=2	col=3	col=4	col=5
row=1				#	
row=2			#	#	
row=3			#	#	
row=4	#		#		
row=5			#	#	
row=6			#	#	
row=7				#	

Task 2

	col=1	col=2	col=3	col=4	col=5
row=1			#	#	
row=2		#	#	#	#
row=3	#				#
row=4					#
row=5				#	
row=6			#		
row=7	#	#	#	#	#

Task 4

	col=1	col=2	col=3	col=4	col=5
row=1				#	
row=2				#	#
row=3			#	#	#
row=4	#		#		#
row=5	#		#	#	#
row=6					#
row=7					#

Task Z

	col=1	col=2	col=3	col=4	col=5
row=1	z	z	z	z	z
row=2	#	#	#	#	
row=3			#		
row=4		#			
row=5	#	#	#	#	#
row=6	z	z	z	z	z
row=7					

Task 1

	col=1	col=2	col=3	col=4	col=5
row=1			#		
row=2			#	#	
row=3	#		#		
row=4				#	
row=5				#	
row=6				#	
row=7	#	#	#	#	#

Learning by experiments and guesswork

The usual and common way to learn programming is just by randomly experimenting with it. This takes much more time to learn programming as it required much practice. No doubt it has advantages, especially if we want to explore the API/in-built libraries or explore the features of a programming language. This particular learning style by guessing for beginners may act like a killer of interest in the programming subject itself. This way, it takes more time to learn programming, and yes, for some people, it is an interesting way. **However, everyone can't guess every subject area, and there is no sure formula for guesswork for everyone to utilize.** It's difficult for a person to experiment with programming who couldn't make a good guess or even awkward guessing. The imagination regarding programming doesn't flow for many of them, and the person's mind goes blank. So it should not be a surprise if a person is good in many other activities and lacks patience in learning programming. This is not specific to learning programming as a subject; it could happen with learning any other subject.

The ground reality is that if imagination and thought process doesn't flow for a person trying to learn something, it just doesn't go into the head. Learning programming by guessing is a switch off to many of them. It doesn't reflect any standard way of learning. Instead, a systematic approach to learning programming based on simple math and common sense makes learners better thinkers. That approach dealt in depth at different levels of difficulties to properly learn programming logic is covered in the next section, **programming patterns**.

In order to do programming in patterns, the person has to know for-loop and applying logical conditions well. Looping is the hardest. A nested loop challenges us to think logically in two different directions simultaneously to output a pattern, and any pattern is a 2-dimensional simple picture. These type of activities stimulates logical thinking. During learning, getting multiple and different thoughts are common. Let's try to think about confusion or doubts from a different perspective. It creates a kind of suspense and makes learning exciting. Confusions and doubts are good for learning programming, and it is an indication that learner's thought process and imagination becomes active. It does happen many times that clearing one's doubt opens the door to explore knowledge in areas. So, doubt basically works like a surprise knowledge capsule. A learner needs to keep working in order to try to resolve that specific doubt or confusion. So, whenever we have any doubt, assume we have got one more chance to improve our intelligence.

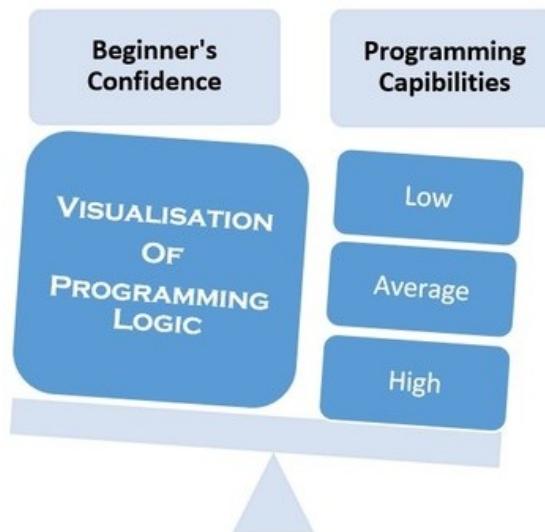
Obviously, learning will not last longer if a systematic approach does not support it. A person who is good at developing programming logic encounters lesser obstacles whenever required to deal with harder subjects of programming. So good programming logic is the key, and this book is all about developing programming logic skills, and it doesn't matter if someone initially was a big zero in it. The content in this book surely helps a person to become interested in programming. The exercises are designed in a way to suit any new learner in increasing order of difficulty naturally. Some exercises are specially included only to surprise, forcing the learner to think out of the box.

Programming Patterns

Any pattern is simply a 2-dimensional picture. We deal with 2-dimensional types of output most of the time, whether it's on the desktop/web/mobile application. So is the reason to select simple picture-based problems to address common programming logic issues. Using a programming language, if someone is good at outputting results in two-dimensional, we can expect that person to take care of the three-dimensional programming aspect, such as game programming.

The main idea behind selecting pictorial pattern-based exercises is that it clearly validates the programmer's logic. The mismatching output forces learner to think for an appropriate reason. This way of repetitively working on programming logic to get the matching output greatly helps perfect the visualization of programming logic. It is one type of reverse engineering, and we will see how it works to improve programming skills. These exercise makes learner alert, involved, self-motivated and sure teaches good amount of patience with ease. This set of exercises does create curiosity a lot. The programmer's imagination should work in a direction to trap the programming logic. If this happens, then programming will be an interesting job. In software development workplaces, one programmer has to understand other programmers' programs. This is also true; a programmer can learn good programming from other programmers. In all cases, some mastery over the visualization of programming logic will definitely help understand other programmer programs. So, practicing visualization of programming logic should be the prime focus for any beginner programmer. Looping is the only hardest thing in a program. Mastering the flow of control in nested loops is enough to boost the confidence of beginner programmers.

In a nutshell, a beginner's confidence in programming capabilities has a direct relation with a low/average/high level of visualization of programming logic.



What we are going to learn is a simple math-based approach to find the actual program logic. This technique requires some level of practice, and the more we practice more it will refine our imagination to trap the programming logic.

If you can imagine, you can do it.

- Enzo Ferrari

Practicing nested loops along with a correct sense of applying logical conditions can immensely help build confidence in a novice programmer.

In general, programmers are always expected to be good at programming logic. If a programmer has that confidence, further advanced technical learning will be taken with much confidence. Every programmer might wish to have that level of imaginative sense so that programming logic should get click naturally, matching the demand of the problem to be solved. It will be just wishful thinking to expect clarity about the visualization of programming logic without proper practice. Solid groundwork requires a reasonable level of clarity about visualization of programming logic and acquiring talent in programming.

The most important protocol that makes visualization of programming logic work correctly is to understand the intelligence involved in the flow of data. The trick to getting this intelligence is to express data flow in one or more logical expressions using our version of the observation table known as the logic trace table. The logical expression is made up of one or more variables and operators. Finally, programming logic is made up of one or more logical expressions. We can easily learn this art of making logical expressions through appropriate practice mentioned in this section.

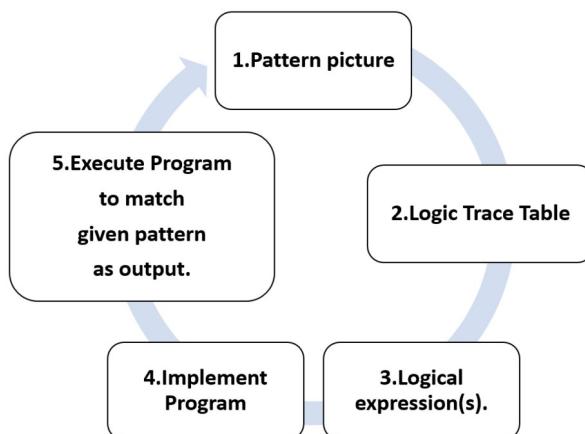
Our approach will be to extract logical expression(s) by observing the output of the program. This thought process is captured in the following approach: nothing but logic-building exercises of one of its kind to develop interest and build confidence in programming. The main steps of the approach are depicted in the following picture. The programmer should be able to visualize the logic clearly even before writing the program. This quality is possible only when the programmer is known to be able to visualize clearly how any repeating control structure works or looping in general. Simple picture-based exercises help greatly in that regard which is basically based on a nested loop. Given is the complete framework to be followed for improving logic-building skills.

The steps need to follow are in a specific order which is given as follows

Step Activity

1. Select the **pattern** picture.
2. Create the **Logic Trace Table** by seeing the pattern picture.
3. Extract the **logical expression(s)** from Logic Trace Table.
4. Implementing the **program** using logical expression(s).
5. Execute the program and check the **program's output to match the selected pattern picture**.

Note: If step 5 goes wrong, need to repeat from step 2 to step 5.



If a person has one quality that is patience to stick to work ethics, then no one can stop that person from becoming talented in that field.

Logic trace table: Brain behind programming logic

Do not deny the classical approach, simply as a reaction, or you will have created another pattern and trapped yourself there.

- Bruce Lee

Conceptually logic trace table is just one type of observation table. All major observations are put into this table so that we should be able to relate the different pieces of information. You can create your own one, or it may depend upon your needs. In our case, the rows and columns of the logic trace table selected with the motive to trace programming logic. We can use an observation table-based approach to solving other types of problems, not just related to patterns.

The main concern is getting involved in perfecting novice programmer's visualization of logic through the usage of logic trace tables and ultimately installing confidence in programming. Furthermore, clarity and work ethics helps to achieve our small goals. Small achievements raise our confidence level to achieve the next level of difficult challenges. So, clarity and work ethics are the foundation for every event of achievement to happen.

What purpose does the logic trace table solve?

Simplicity is the key to brilliance.

- Bruce Lee

It's a simple trick to train novice programmer's minds on logical thinking. Its main purpose is to apply thinking in a more structured manner to get programming logic. It is a simple table structure meant to derive the logical expression. The table structure basically gives clarity about the relationship among the variable parameters.

Clarity is power.

-The more clear you are about EXACTLY what it is that you want, the more your brain knows how to get there.

- Unknown

In my opinion, without taking the help of table structure, most novice programmers will find it difficult to think clearly in terms of building programming logic. Clear thinking to find logical expression is a process that can be improved with a little bit of practice. Unless and until a person is able to put his/her understanding in a simplified way, it means that person had not really understood the concept.

What is the importance of logical expression?

Programming logic is the brain behind a program. It is made up of logical expression(s). So, it is a logical expression at the very core level - when executed - helps in decision-making. Most of the time, it is applied in the condition of looping, in decision structures like **if-condition**, **if-elif condition**.

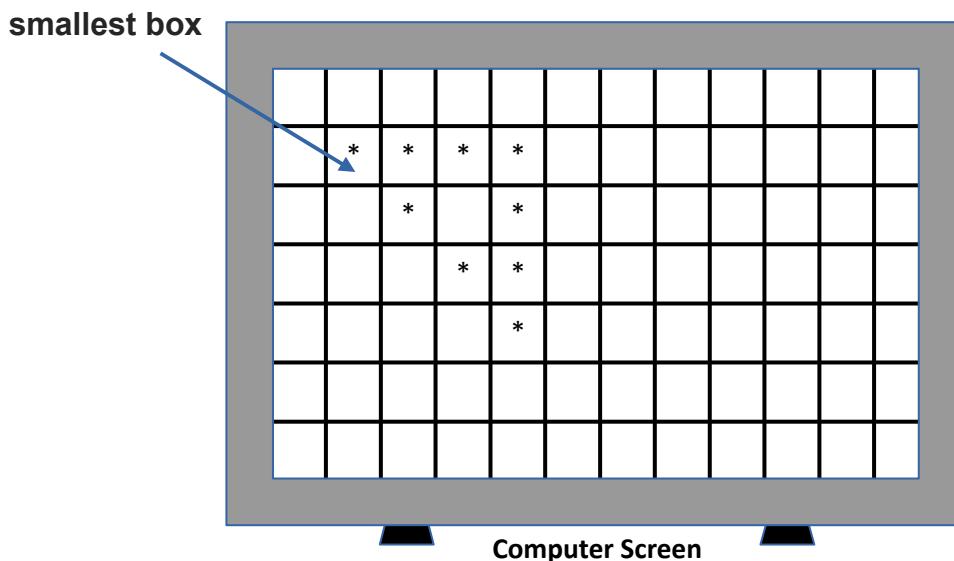
PART:2

PROGRAMMING

IN ACTION

Outputting Simple Picture on Computer Screen

The key point is to understand that the computer screen is nothing more than a logical collection of small boxes where individual small boxes can output individual keyboard characters. These boxes are arranged in such a way to give a clear idea about rows and columns. Following is the conceptual diagram of a computer screen showing some rows and columns of a simple picture outputted into the computer screen. Each keyboard character is in one of the single smallest boxes.



Before we begin programming with simple picture-based exercises, we must first learn to allocate row numbers and column numbers conceptually. This will give justice to identify each smallest box with a singular keyboard character uniquely. The **(row number, column number)** numeric combination will imply a single keyboard character. Mentioning **(row number, column number)** numeric combination is simply a reason to locate any keyboard character on the computer screen correctly. It is quite analogous to locating a city on the geographical map using **(longitude, latitude)** combination. Hence, our version of the single smallest box maps to a single keyboard character on the computer screen and equivalently maps to a unique **(row number, column number)** numeric pair. In this way, the collection of **(row number, column number)** numeric pairs can be used to represent a simple picture completely.

Single Smallest Box → Single keyboard character →(row number, column number) numeric pair.

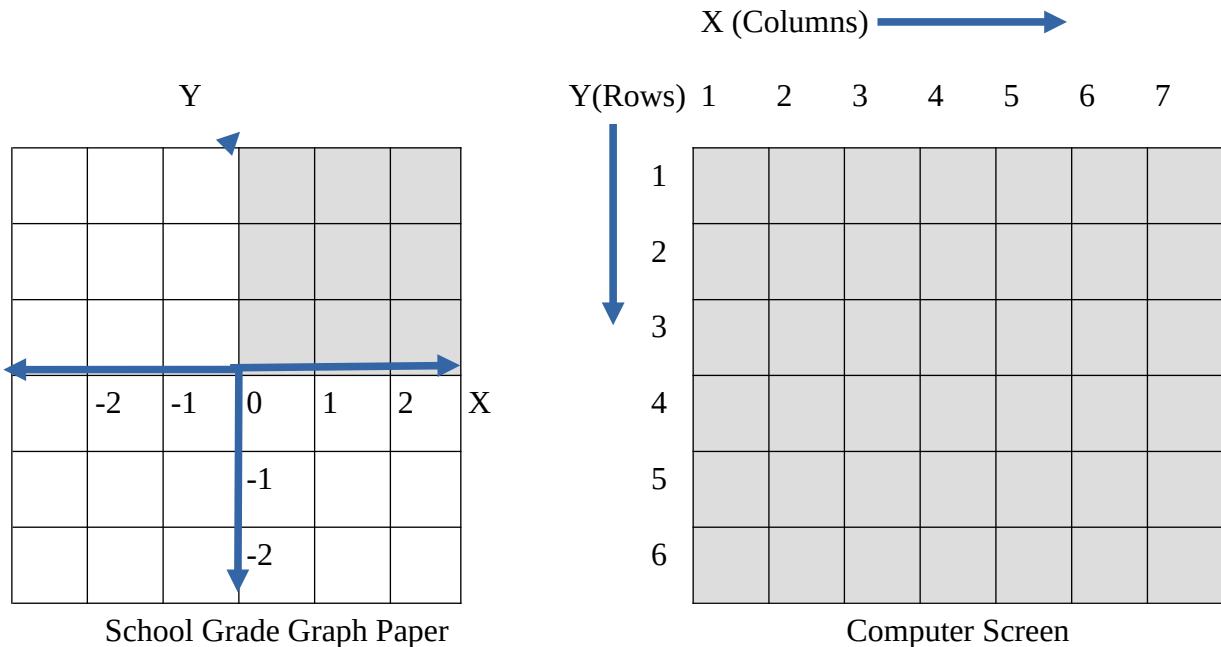
A diagram showing a grid of small boxes on a computer screen. The grid is 7 rows by 12 columns. Row 1 contains 12 empty boxes. Row 2 contains 4 boxes with an asterisk (*) in columns 2, 3, 4, and 5. Row 3 contains 2 boxes with an asterisk (*) in columns 3 and 4. Row 4 contains 2 boxes with an asterisk (*) in columns 6 and 7. Row 5 contains 1 box with an asterisk (*) in column 7. Rows 6 and 7 are entirely empty. To the left of the grid, a vertical blue arrow points downwards and is labeled "Rows". Above the grid, a horizontal blue arrow points to the right and is labeled "Columns". The grid is labeled "Computer Screen" at the bottom.

Rows	1	2	3	4	5	6	7	8	9	10	11	12
1												
2		*	*	*	*							
3			*		*							
4				*	*							
5					*							
6												
7												

Computer Screen

Alternately, we can think of the computer screen as a sophisticated version of graph paper or grid paper. Each smallest box of the computer screen is a coordinate, a unique combination of two numbers, an "X (Columns)"(horizontal) and a "Y(Rows)"(vertical)- that determines the location of a keyboard character in computer screen space.

It is our programming job; how can a simple picture-based pattern appear at these smallest box coordinates represented by (**row number, column number**) numeric pairs.



As discussed previously, we can represent any 2-dimensional simple picture in terms of rows and columns on the computer screen. So, such simple pictures made up of keyboard characters can be completely caged into a collection of small boxes. Each keyboard character of a considered simple picture can have a unique (**row number, column number**) numeric pair. Hence, the collection of such numeric pairs can totally represent a complete simple picture.

The collection of related unique (row number, column number) numeric pairs of this simple picture can be given as
 $(2,2), (2,3), (2,4), (2,5)$
 $(3,3), (3,5), (4,4), (4,5),$
 $(5,5)$

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1					
Row=2		$*(2,2)$	$*(2,3)$	$*(2,4)$	$*(2,5)$
Row=3			$*(3,3)$		$*(3,5)$
Row=4				$*(4,4)$	$*(4,5)$
Row=5					$*(5,5)$

Cursor Movement

A moving position indicator, shown on a computer monitor where the next move or operation will occur. We will use this concept to explain better how each character of a pattern got generated on the computer screen or console screen of the programming IDE. In our case, keyboard characters are to be outputted using inbuilt libraries onto the computer screen at specific locations. So, the cursor moves to the next position on the screen, and then the printing of the character will be taken care of by inbuilt methods.

Pattern 1A

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. Thus, the address in each smallest box on the display screen will logically become a unique (row, column) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread across the screen.



Data Representation

Although logically, we can exactly locate the position of each printable character on the display

Row, Col	Col = 1	Col = 2	Col = 3	Col = 4	Col = 5
Row = 1	* (1, 1)				
Row = 2	* (2, 1)	*(2, 2)			
Row = 3	* (3, 1)	*(3, 2)	*(3, 3)		
Row = 4	* (4, 1)	*(4, 2)	*(4, 3)	*(4, 4)	
Row = 5	* (5, 1)	*(5, 2)	*(5, 3)	*(5, 4)	*(5, 5)

The pair (4, 3) represents a single character * at the assumed position of 4th row and 3rd column on the display screen. The set of all value pairs (row = value, col = value) now represent this particular pattern.

screen by knowing its uniquely identifiable (row, column) numeric paired values.

Analysis of output

Computer dominantly works on very precise calculations. One interesting way to practice programming is to try converting manual steps of writing a given pattern into programming instructions. We can use paper or a spreadsheet. We can mark row numbers and column numbers to identify position indicators to analyze a pattern correctly logically.

	Col = 1	Col = 2	Col = 3	Col = 4	Col = 5	WRITING ACTION
ROW	Move to Row = 1	1 → *				Write * 1 time
MOVEMENT	Move to Row = 2	1 → *	2 → *			Write * 2 times
ACTION	Move to Row = 3	1 → *	2 → *	3 → *		Write * 3 times
	Move to Row = 4	1 → *	2 → *	3 → *	4 → *	Write * 4 times
	Move to Row = 5	1 → *	2 → *	3 → *	4 → *	5 → * Write * 5 times

Decomposition

We can decompose our repetitive actions of printing pattern into the following categories

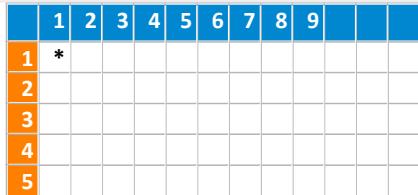
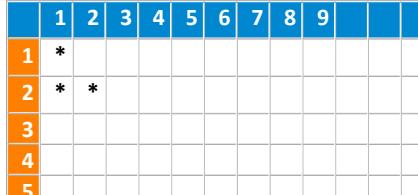
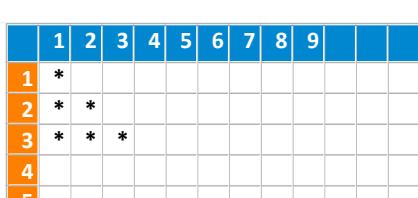
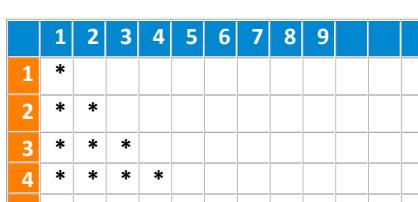
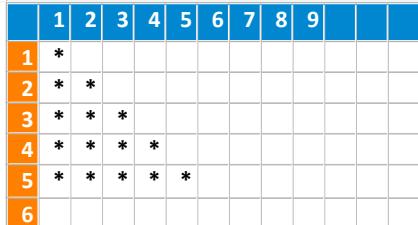
Repetitive Actions

a) **Row Level:** Repetitively moving down, marking the next row number sequence-wise.

b) **Column Level:** Repetitively writing * characters at the column level.

The given pattern is a 2-dimensional simple picture such that it can be expressed in terms of rows and columns. Our next step will be to transform these manual, repetitive actions into a working program. We will use * characters in our program to print it on the computer screen. The working program should behave in the following way.

Steps for generation of pattern on the screen.

		COLUMN LEVEL	Computer Screen
R O W L E V E L	row = 1	1) Print “**” <u>once</u> in the same 1 st row with col = 1. 2) Move to row number 2.	
	row = 2	1) Print “**” <u>twice</u> in the same 2 nd row with col = 1, 2. 2) Move to row number 3.	
	row = 3	1) Print “**” <u>thrice</u> in the same 3 rd row with col = 1, 2, 3. 2) Move to row number 4.	
	row = 4	1) Print “**” <u>four times</u> in the same 4 th row with col = 1, 2, 3, 4. 2) Move to row number 5.	
	row = 5	1) Print “**” <u>five times</u> in the 5 th row with col = 1, 2, 3, 4, 5 2) Move to row number 6.	

Let's try to mimic the steps by writing a program using given in-built methods.

Purpose	Method Name
In order to shift the program's control to the <u>next row</u> .	<code>print()</code>
To move the program's control in the same row for the next column position (same row , new column) and then print the required character. Thus, each column movement on the screen will be a single character-sized jump from the left to the right direction in the <u>same row</u> .	<code>print("*", end="")</code>

We can easily observe that the row value increases by 1 multiple times. This is the same feature of the **for-loop** control structure. Hence, we can apply **for-loop** on row values.

Pattern1A1If.py

```
for row in range(1, 5 + 1):
    # Use of if-condition to print * in the columns or in the same row
    if (row == 1):
        print("*", end="# col = 1")

    if (row == 2):
        print("*", end="# col = 1")
        print("*", end="# col = 2")

    if (row == 3):
        print("*", end="# col = 1")
        print("*", end="# col = 2")
        print("*", end="# col = 3")

    if (row == 4):
        print("*", end="# col = 1")
        print("*", end="# col = 2")
        print("*", end="# col = 3")
        print("*", end="# col = 4")

    if (row == 5):
        print("*", end="# col = 1")
        print("*", end="# col = 2")
        print("*", end="# col = 3")
        print("*", end="# col = 4")
        print("*", end="# col = 5")

    print()# Move to the next row
```

Fixed Output Implementation

*	Whenever we execute this program, we will always get a fixed output in 5 rows.
**	We cannot output bigger or smaller patterns for different rows unless we manually change the given for-loop and add/subtract if-conditions .
***	Implementing generalized programming logic essentially means we should be able to get the customized output without modifying the actual logic of a program.

How to get generalized programming logic

One suitable way to get generalized programming logic is to use our own version of the logic trace table. Basically, a logic trace table is a concept, and it can be applied in general to get the logic of a program. Obviously, its structure will get differ accordingly depends upon the type of problem to be solved. We are learning how to apply this concept of logic trace table. One of the interesting ways to do so is to implement console-based programs of the type pattern-recognition-based problems of various shapes. Getting the program's logic is easy if we clearly understand how the changes in the logic trace table will impact the implementation of the program under consideration. Let's try to understand the principles on which it works.

Principles of Logic trace Table/Observation Table

It's a set of ways for interacting with the program's logic that hasn't been actually explored yet. It helps in doing that. The final generalized programming logic is by nature made up of one more generalized logical expression. So, we need to trace out generalized logical expressions from the logic trace table. The way to get generalized logical expression is to follow a few of the given facts;

- ✓ The order of execution of variables in a given program. In our case, in general, the working program consists of outer-loop and inner-loop. Since we know that outer-loop executes before nested-loop, the **variable: row** will execute before the **variable: col**.

GENERAL APPROACH TO PRINTING PATTERN

Row level Repetitive Action - Outer Loop

Column level Repetitive Action - Nested Loop

Move To Next Row

- ✓ The logic trace table is a snapshot of how different variables store different values during program execution. For example, suppose two variables follow the same numeric pattern during the execution of the program. In that case, the variable which executes first will act as a replacement for the numeric pattern followed by the other variable. In this particular way, we need to try to replace a set of hard-coded numerical values with the available variable(s) of a given program. Hence, it is of utmost importance to check if any numeric pattern matching exists between the columns of the logic trace table.
- ✓ After completing numeric pattern matching and its replacement using the available variable(s), we need to check the consistency of an expression in a particular column. If it holds True, then it is a valid generalized logical expression. A generalized logical expression may consist of one or many variables.
- ✓ We need to check if there exists the same numeric value in a particular column throughout. If this holds True, then it must be a generalized numeric expression, a singular numerical value in the whole column.

So, the bottom line is we need to update columns of a logic trace table till we get consistent numeric or logical expression(s) for all the rows. We will see how to apply these points practically to build the logic of a program. This is an attempt to design the program's logic by applying **logic trace table principles on the columns of the given table to get** suitable numeric/logical expressions. We can start with creating our first version of the logic trace table using the **PatternA1f.py** program.

COLUMN LEVEL REPETITIVE ACTION

Abstracting out details of **PatternA1If.py** program to create a logic trace table. The given table decomposes the given pattern into a relationship of **one-to-many** between row and column values.

INPUT	ROW LEVEL	COLUMN LEVEL
TOTAL ROWS = 5	REPETITIVE ACTION	REPETITIVE ACTION
*	row = 1	col = 1
**	row = 2	col = 1, 2
***	row = 3	col = 1, 2, 3
****	row = 4	col = 1, 2, 3, 4
*****	row = 5	col = 1, 2, 3, 4, 5

To uniquely identify each printing character of a given pattern, let's consider variables **row** and **col**. Choosing variables **row** and **col** to precisely mark printing character's row and column number position respectively in a given pattern. Every value of the **variable: row** can be mapped to a set of **variable: col** values at the column level. The **variable: col** values maintain a common difference of value 1 if we move from *starting value* to the *ending value*. Wait for a second. Doesn't it very much sound like another case of **for-loop** construction?

ROW LEVEL	COLUMN LEVEL	
ROW VALUE	COLUMN VALUES	COMMON DIFFERENCE
row = 1	col = 1	Start value = 1 End value = 1 1 = 1
row = 2	col = 1, 2	Start value = 1 End value = 2 2-1=1
row = 3	col = 1, 2, 3	Start value = 1 End value = 3 2-1=1, 3-2=1
row = 4	col = 1, 2, 3, 4	Start value = 1 End value = 4 2-1=1, 3-2=1, 4-3=1
row = 5	col = 1, 2, 3, 4, 5	Start value = 1 End value = 5 2-1=1, 3-2=1, 4-3=1, 5-4=1

By considering **variable: row** and **variable: col**, we can further refine the logic trace table with the intention of building **for-loop** against each value of row number. The **for-loop** will be basically a nested-loop for each row's value.

INPUT	ROW LEVEL FOR LOOP	COLUMN LEVEL FOR LOOP	
TOTAL ROWS = 5	Variable: row	Variable: col	
PRINT PATTERN	start = 1 end = 5	start	end
*	row = 1	col = 1	col = 1
**	row = 2	col = 1	col = 2
***	row = 3	col = 1	col = 3
****	row = 4	col = 1	col = 4
*****	row = 5	col = 1	col = 5

Based on the recently updated logic trace table, we can update the program's logic as follows.

PatternA1NestedLoop.py

```
for row in range(1, 5 + 1):
    # Use of if-condition to print * in the columns or in the same row

    if (row == 1):
        for col in range(1, 1 + 1):
            print("*", end="")# columns : start = 1 | end = 1

    if (row == 2):
        for col in range(1, 2 + 1):
            print("*", end="")# columns : start = 1 | end = 2

    if (row == 3):
        for col in range(1, 3 + 1):
            print("*", end="")# columns : start = 1 | end = 3

    if (row == 4):
        for col in range(1, 4 + 1):
            print("*", end="")# columns : start = 1 | end = 4

    if (row == 5):
        for col in range(1, 5 + 1):
            print("*", end="")# columns : start = 1 | end = 5

    print()# Move to the next row
```

Output

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

Our program becomes better than before by using nested loops. For each row's value, we had successfully formulated a single nested loop. Hence, the count of nested loops will go up or down based on the number of rows used. The next step in generalization will be to convert many nested-loops into one nested-loop. Such conversion to a single nested loop suggests that the number of rows inputted to the program should not impact our solution. We will see all these in action while applying the principles of the logic trace table. Primarily, our goal will be to get consistent numeric or logical expression(s) in the columns of the logic trace table. We will use these obtained consistent numeric or logical expression(s) results to design our program. Derived expressions are only considered correct when it gives correct output; otherwise, we need to re-work the logic trace table to find our own mistake(s). This is the “**computational thinking**” way to learn the design of a program’s logic compared to the trial-error method.

We can find a column where a numeric value is consistent.



INPUT VARIABLE	ROW LEVEL FOR LOOP		COLUMN LEVEL FOR LOOP	
totalRows = 5	Variable: row		Variable: col	
PRPATTERN	start = 1	end = 5	start	end
*	row = 1		col = 1	col = 1
**	row = 2		col = 1	col = 2
***	row = 3		col = 1	col = 3
****	row = 4		col = 1	col = 4
*****	row = 5		col = 1	col = 5

The column **start** contains the data **col = 1**, which is the same for every row. Hence, column **start** is already being in a generalized state, and hence we got the consistent logical expression in order to **start** the column level for-loop. The **column level for-loop** must require a singular starting value and a singular end value.

INPUT VARIABLE	ROW LEVEL FOR LOOP		COLUMN LEVEL FOR LOOP	
totalRows = 5	Variable: row		Variable: col	
PRPATTERN	start = 1	end = 5	start	end
*	row = 1		col = 1	col = 1
**	row = 2			col = 2
***	row = 3		consistent	col = 3
****	row = 4		numerical	col = 4
*****	row = 5		expression	col = 5

Clearly, the column **end** with a singular value is not our case. The other remained possibility is to derive a logical expression that should be consistent for this column. We can observe numeric pattern variations of the **variable: col** and the **variable: row**. The column **end** matches exactly with numeric pattern variations of the **variable: row**, shown in the above logic trace table.

We can find a column where a logical expression is consistent.



INPUT VARIABLE	ROW LEVEL For Loop		COLUMN LEVEL For Loop	
totalRows = 5	Variable: row		Variable: col	
PRPATTERN	start = 1	end = 5	start	end
*	row = 1		col = 1	col = row
**	row = 2			col = row
***	row = 3			col = row
****	row = 4			col = row
*****	row = 5			col = row

It means column level for-loop with **variable: col** starts at value 1 and ends at the most recent value taken by the **variable: row**. Hence, we got the consistent logical expression (**col = row**) for the column **end**.

INPUT VARIABLE	ROW LEVEL For Loop		COLUMN LEVEL For Loop	
totalRows = 5	Variable: row		Variable: col	
PRPATTERN	start = 1	end = 5	start	end
*	row = 1		col = 1	col = row
**	row = 2			
***	row = 3			
****	row = 4			
*****	row = 5			

So, during the execution of the program, row level for-loop with **variable: row** takes the value 1 to 5. Then column level for-loop with **variable: col** will take values between 1 and the current value of the **variable: row**.

Mathematically, it can be given as

$$1 \leqslant \text{col} \leqslant \text{row}$$

Where **row = 1 to 5**

The row level for-loop will get started at the value 1, and it will be stopped and controlled by a changing value of the **variable: row** during the program's execution. At each row's value, the column-level **for-loop** should stop looping and is controlled by the logical expression (**col <= row**). The consistent logical expression (**col <= row**) will be the reason to produce dynamic output at the column level for different values of **variable: row**. Extracting the consistent logical expression is a way

to generalize and to get a customized solution. When we are able to establish a relationship between variables, then we can have a chance to automate the process of computing.

We can now clearly define the action of column level ***for-loop***.

ELEMENTS OF COLUMN LEVEL FOR LOOP			
Variable: col		INCREMENTAL ORDER	ACTION
start	end	step-size	Print character
1	row	1	*

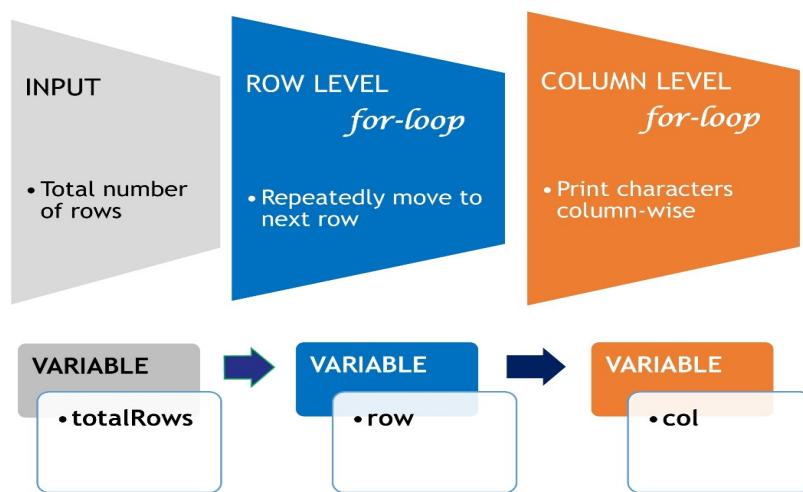
COLUMN LEVEL FOR LOOP OF INCREMENTAL ORDER

Column level Repetitive Action

```
for col in range(1, end + 1):
    Print character *
```

ROW LEVEL REPETITIVE ACTION

We have used a logic trace table as a tool to get the generalized solution. The logic trace table is the snap-shot of how different variables **totalRows**, **row**, and **col** store different values and in what matching pattern during the program's execution.



ORDER OF EXECUTION OF VARIABLES IN A PROGRAM.

The following logic trace table is basically ***for-loop*** in action at row level. The only action expected from **row level for-loop** in our case is to shift the program's control to the next row. It should be done dynamically based on the **total number of rows** inputted by the **variable: totalRows**.

Our target is to get singular numeric/logical expressions for the **start** and **end** of a **row level for-loop**.

We can find a column where a
numeric value is consistent.



ROW LEVEL FOR LOOP			
INPUT VARIABLE	Variable: row		INCREMENTAL ORDER
TOTAL Rows	start	end	step-size = 1
totalRows = 3	1	3	Move To Next Row 3 times
totalRows = 5	1	5	Move To Next Row 5 times
totalRows = 7	1	7	Move To Next Row 7 times
totalRows = 10	1	10	Move To Next Row 10 times

This **for-loop** looping **starts with value 1** and is the same for every different input given by the total number of rows. Hence, it was already being in a generalized state and marked down and updating the given logic trace table with a single numeric expression value 1. Moreover, the numeric pattern which is shown in **shaded columns** of the given table is matching exactly. During the program's execution, the variable: **totalRows** is the first one to get executed, and other columns that are marked shaded followed the same data trend. Hence, **variable: totalRows** is a good replacement for the numeric pattern that exists in a different column and followed the numeric pattern exactly. This will establish the relationship between the input **variable: totalRows** and **row-level for-loop variable: row**.

It means that during the program's execution, when **variable: totalRows** contains the value **3**, then the **row level for-loop** should **end** after **3** repetitions.

For **5** repetitions of **row level for-loop**, the program's control moves to the next rows **3** times or **variable: totalRows** should contain value **5**.

For **7** repetitions of **row level for-loop**, the program's control moves to the next rows **7** times or **variable: totalRows** should contain value **7**.

For **10** repetitions of **row level for-loop**, the program's control moves to the next rows **10** times or **variable: totalRows** should contain value **10**.

ROW LEVEL FOR LOOP			
INPUT	Variable: row		INCREMENTAL ORDER
TOTAL Rows	start	end	STEP-SIZE = 1
totalRows = 3	1	3	Move To Next Row 3 times
totalRows = 5		5	Move To Next Row 5 times
totalRows = 7	CONSISTENT	7	Move To Next Row 7 times
totalRows = 10	NUMERIC EXPRESSION	10	Move To Next Row 10 times

After numeric pattern matching and applying replacement using **variable: totalRows**, the logic trace table looks like the following.

We can find a column where a logical expression is consistent.



ROW LEVEL FOR LOOP					
INPUT		Variable: row		INCREMENTAL ORDER	
TOTAL Rows		start	end	STEP-SIZE = 1	
totalRows = 3	1	totalRows	Move To Next Row	totalRows	times
totalRows = 5		totalRows	Move To Next Row	totalRows	times
totalRows = 7		totalRows	Move To Next Row	totalRows	times
totalRows = 10		totalRows	Move To Next Row	totalRows	times

The set of data in the border marked columns shows a consistent logical expression after updating the previously logic trace table with a single consistent logical expression. The following logic tracing table suggests that maximum possible repetition of **row level for-loop** and moving program's control to the next row can be controlled by the value stored in the **variable: totalRows**. Now, the stored value of **totalRows** will act as the maximum boundary value for the **row level for-loop** repetitions. This is how it formulates a flexible solution to resize the pattern's length based on the input value of the **variable: totalRows**. Whenever a row-level repetitive action becomes generalized, it can print a pattern of matching length using the total number of rows inputted to the program.

ROW LEVEL FOR LOOP					
INPUT		Variable: row		INCREMENTAL ORDER	
TOTAL Rows		start	end	STEP-SIZE = 1	
totalRows = 3	1	totalRows	Move To Next Row	totalRows	times
totalRows = 5					
totalRows = 7		CONSISTENT		CONSISTENT	
totalRows = 10		LOGICAL EXPRESSION		LOGICAL EXPRESSION	

We can now clearly define the action of column level **for-loop**.

ELEMENTS OF ROW LEVEL FOR LOOP			
Variable: row		INCREMENTAL ORDER	ACTION
start	end	step-size	Statements
1	totalRows	1	1# COLUMN LEVEL For Loop 2# Move To Next Row

ROW LEVEL For Loop Of Incremental Order

Row level Repetitive Action

```
totalRows = 5  
step-size = 1  
for row in range(1, totalRows + 1, step-size):
```

Column level Repetitive Action

[Move To Next Row](#)

POINTS TO PONDER:

THE WAY TO A CUSTOMIZED SOLUTION IS TO FIND CONSISTENT NUMERICAL/LOGICAL EXPRESSION

- The way to generalized/customized solutions is to pass through the stage of abstraction. The key to abstraction in our case is to obtain the consistent numeric or logical expression by matching data collectively among the columns of the logic trace table. We refer to this activity as pattern matching. Let's apply pattern matching to find the program's logic.
- A numeric pattern in a logic trace table's column is a foot-prof variable obtained during program execution. Observe the overlapping of numeric patterns. Replacing the matching numeric pattern by available variable(s) targeting a consistent logical expression.
- The order of variable execution is very important to apply numeric pattern matching between the columns of the logic trace table. This is the basic step to get generalized logical expressions and finally leads to the complete generalization of a program.

Whenever the program does not take care of moving the program's control from one row to another, then, for example, someone may end up printing a straight line instead of printing a triangle shaped-pattern.

CASE A. WHEN THE PROGRAM'S CONTROL MOVE TO THE NEXT LINE CORRECTLY

```
* Move to next line  
* * Move to next line  
* * * Move to next line  
* * * * Move to next line  
* * * *
```

CASE B. WHEN THE PROGRAM'S CONTROL DOES NOT MOVE TO THE NEXT LINE

```
* Move to next line - Skip this step.  
* * Move to next line - Skip this step.  
* * * Move to next line - Skip this step.  
* * * * Move to next line - Skip this step.  
* * * *
```

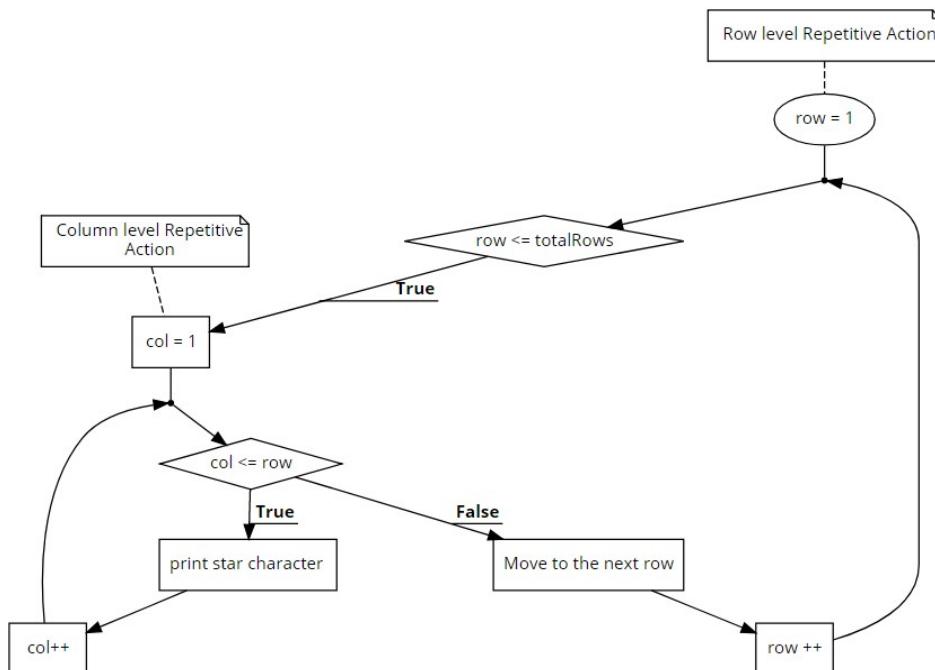
PROGRAM'S OUTPUT—THE PATTERN GOES FLAT.

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

Most beginner programmers experience such fun-filled surprises while learning to code. The Beauty of learning to code in such a way is that code failures do not put learners on a depressing note. It should stimulate a higher level of interest and motivates to do it better next time. The real challenges a learner will face are while handling **column level for-loop**. It requires learners to find a suitable consistent logical expression.

FLOW CHART

We got everything related to **for-loop**.



Let's execute the following program.

Pattern1A_1.py

```
totalRows = 5 # number of rows to display
# Row level Repetitive Action:
# Action1. Executes Column level Repetitive Action:
# Action2. Move cursor to next row.
for row in range(1, totalRows + 1):
    # Column level Repetitive Action
    for col in range(1, row + 1):
        # Action1.Move cursor in the same row.
        # Action2.print * character
        print("*", end = "")

    print()# Move cursor to the next row
```

Output

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

Now, test for a different number of lines; let it be 10. It should work correctly, so to confirm that it generalizes our solution.

Pattern1A_1.py

```
totalRows = 10 # number of rows to display
# Row level Repetitive Action:
# Action1. Executes Column level Repetitive Action:
# Action2. Move cursor to next row.
for row in range(1, totalRows + 1):
    # Column level Repetitive Action
    for col in range(1, row + 1):
        # Action1.Move cursor in the same row.
        # Action2.print * character
        print("*", end = "")

    print()# Move cursor to the next row
```

Output

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

```
*****
```

```
*****
```

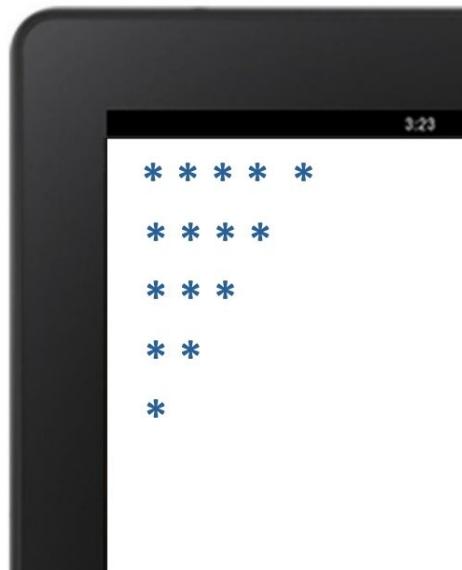
```
*****
```

```
*****
```

```
*****
```

Pattern1B

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. Thus, the address in each smallest box on the display screen will logically become a unique (row, column) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread across the screen.



Data Representation

Although logically, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (row, column) numeric paired values.

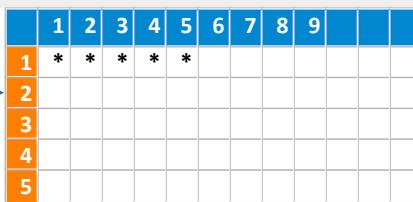
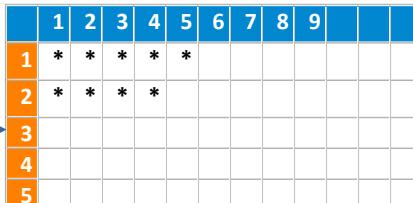
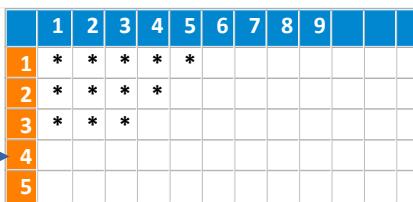
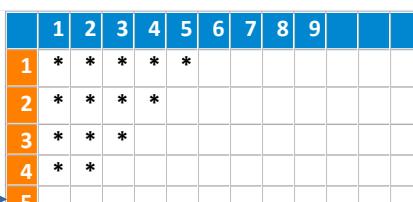
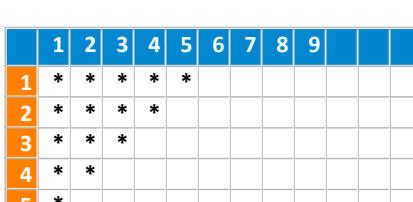
Row, Col	Col = 1	Col = 2	Col = 3	Col = 4	Col = 5
Row = 1	* (1, 1)	* (1, 2)	*(1, 3)	* (1, 4)	* (1, 5)
Row = 2	* (2, 1)	* (2, 2)	* (2, 3)	* (2, 4)	
Row = 3	* (3, 1)	* (3, 2)	* (3, 3)		
Row = 4	* (4, 1)	* (4, 2)			
Row = 5	* (5, 1)				

The pair (2, 4) represents a single character * at the assumed position of 2nd row and 4th column on the display screen. The set of all value pairs (row = value, col = value) now represent this particular pattern.

Analysis of output

Computer dominantly works on very precise calculations. One interesting way to practice programming is to try converting manual steps of writing a given pattern into programming instructions. We can use paper or a spreadsheet. To analyze a pattern correctly, we can mark row numbers and column numbers to identify position-indicator logically.

	Col = 1	Col = 2	Col = 3	Col = 4	Col = 5	WRITING ACTION
ROW	Move to Row = 1	1 → *	2 → *	3 → *	4 → *	5 → * Write * 5 times
MOVEMENT	Move to Row = 2	1 → *	2 → *	3 → *	4 → *	Write * 4 times
ACTION	Move to Row = 3	1 → *	2 → *	3 → *		Write * 3 times
	Move to Row = 4	1 → *	2 → *			Write * 2 times
	Move to Row = 5	1 → *				Write * 1 time

Steps for pattern generation (Total Rows = 5)		Computer Screen
For row = 1	Print "*" <u>five</u> times in the same 1 st row and then move to row number 2.	
For row = 2	Print "*" <u>four</u> times in the same 2 nd row and then move to row number 3.	
For row = 3	Print "*" <u>thrice</u> in the same 3 rd row and then move to row number 4.	
For row = 4	Print "*" <u>twice</u> in the same 4 th row and then move to row number 5.	
For row = 5	Print "*" <u>once</u> in the 5 th row and then move to row number 6.	

Let's convert steps into a program.

Use of the in-built method	
In order to move the cursor to the <u>next row</u>	<code>print()</code>
Move the cursor to the next (row, column) position and then Print the star "*" character. Thus, each movement on the computer screen will be a single keyboard character-sized jump from the left to the right direction in the <u>same row</u> .	<code>print("*", end="")</code>

These are two kinds of statements that will get repeated many times in the program.

`print() & print("*", end="")`

Hence, we can use **for-loop** for the same.

PatternB1If.py

```
for row in range(1, 5 + 1):
    # Use of if-condition to print * in the columns or in the same row
    if (row == 1):
        print("*", end="")# col = 1
        print("*", end="")# col = 2
        print("*", end="")# col = 3
        print("*", end="")# col = 4
        print("*", end="")# col = 5

    if (row == 2):
        print("*", end="")# col = 1
        print("*", end="")# col = 2
        print("*", end="")# col = 3
        print("*", end="")# col = 4

    if (row == 3):
        print("*", end="")# col = 1
        print("*", end="")# col = 2
        print("*", end="")# col = 3

    if (row == 4):
        print("*", end="")# col = 1
        print("*", end="")# col = 2

    if (row == 5):
        print("*", end="")# col = 1

print()# Move to the next row
```

Fixed Output Implementation

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

WHENEVER WE EXECUTE THIS PROGRAM, WE WILL ALWAYS GET A FIXED OUTPUT IN 5 ROWS. WE CANNOT OUTPUT BIGGER OR SMALLER PATTERNS FOR DIFFERENT ROWS UNLESS WE MANUALLY CHANGE THE GIVEN **FOR-LOOP** AND ADD/SUBTRACT **IF-CONDITIONS**. IMPLEMENTING GENERALIZED PROGRAMMING LOGIC ESSENTIALLY MEANS WE SHOULD BE ABLE TO GET A CUSTOMIZED OUTPUT WITHOUT MODIFYING THE ACTUAL LOGIC OF A PROGRAM.

COLUMN LEVEL REPETITIVE ACTION

Abstracting out details of **PatternB1If.py** program to create a logic trace table. The given table decomposes the given pattern into a relationship of **one-to-many** between row and column values.

INPUT	ROW LEVEL	COLUMN LEVEL
TOTAL ROWS = 5	REPETITIVE ACTION	REPETITIVE ACTION
*****	row = 1	col = 1, 2, 3, 4, 5
****	row = 2	col = 1, 2, 3, 4
***	row = 3	col = 1, 2, 3
**	row = 4	col = 1, 2
*	row = 5	col = 1

For the purpose of uniquely identifying each printing character of a given pattern, let's consider variables **row** and **col**. Choosing variables **row** and **col** to precisely mark printing character's row and column number position respectively in a given pattern. Every value of the **variable: row** can be mapped to a set of **variable: col** values at the column level. The **variable: col** values maintain a common difference of value 1 if we move from starting value to the ending value. Wait for a second. Doesn't it very much sound like another case of **for-loop** construction?

ROW LEVEL		
ROW VALUE	COLUMN VALUES	COMMON DIFFERENCE
row = 1	col = 1, 2, 3, 4, 5	Start value = 1 End value = 5 $2-1=1, 3-2=1, 4-3=1, 5-4=1$
row = 2	col = 1, 2, 3, 4	Start value = 1 End value = 4 $2-1=1, 3-2=1, 4-3=1$
row = 3	col = 1, 2, 3	Start value = 1 End value = 3 $2-1=1, 3-2=1$
row = 4	col = 1, 2	Start value = 1 End value = 2 $2-1=1$
row = 5	col = 1	Start value = 1 End value = 1 $1 = 1$

By considering **variable: row** and **variable: col**, we can further refine the logic trace table with the intention of building **for-loop** against each value of row number. The **for-loop** will be basically a nested-loop for each row's value.

INPUT	ROW LEVEL FOR LOOP	COLUMN LEVEL FOR LOOP	
TOTAL ROWS = 5	Variable: row	Variable: col	
PRPATTERN	start = 1 end = 5	start	end
*****	row = 1	col = 1	col = 5
****	row = 2	col = 1	col = 4
***	row = 3	col = 1	col = 3
**	row = 4	col = 1	col = 2
*	row = 5	col = 1	col = 1

Based on the recently updated logic trace table, we can update the program's logic given as follows.

PatternB1NestedLoop.py

```
for row in range(1, 5 + 1):
# Use of if-condition to print * in the columns or in the same row

    if (row == 1):

        for col in range(1, 5 + 1):

            print("*", end="")# columns : start = 1 | end = 5

    if (row == 2):

        for col in range(1, 4 + 1):

            print("*", end="")# columns : start = 1 | end = 4

    if (row == 3):

        for col in range(1, 3 + 1):

            print("*", end="")# columns : start = 1 | end = 3

    if (row == 4):

        for col in range(1, 2 + 1):

            print("*", end="")# columns : start = 1 | end = 2

    if (row == 5):

        for col in range(1, 1 + 1):

            print("*", end="")# columns : start = 1 | end = 1

print()# Move to the next row
```

Output

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

Our program becomes better than before by using nested loops. For each row's value, we had successfully formulated a single nested loop. Hence, the count of nested loops will go up or down based on the number of rows used. The next step in generalization will be to convert many nested loops into one nested loop. It cannot be done unless and until we get consistent numeric or logical expression(s) in the columns of the logic trace table against all row values.

We can find a column where a numeric value is consistent.



INPUT	ROW LEVEL FOR LOOP		COLUMN LEVEL FOR LOOP	
TOTAL ROWS = 5	Variable: row		Variable: col	
PRPATTERN	start = 1	end = 5	start	end
*****	row = 1		col = 1	col = 5
****	row = 2		col = 1	col = 4
***	row = 3		col = 1	col = 3
**	row = 4		col = 1	col = 2
*	row = 5		col = 1	col = 1

Any **for-loop** must require a singular starting value and a singular end value. The column **end** contains the data **col = 1**, which is the same for every row. Hence, the column **end** is already being in a generalized state. We got the consistent numerical expression to end the column level for-loop.

INPUT	ROW LEVEL FOR LOOP		COLUMN LEVEL FOR LOOP	
TOTAL ROWS = 5	Variable: row		Variable: col	
PRPATTERN	start = 1	end = 5	start	end
*****	row = 1		col = 1	col = 5
****	row = 2			col = 4
***	row = 3		consistent numerical expression	col = 3
**	row = 4			col = 2
*	row = 5			col = 1

Target to achieve

We need to find a unique expression for the column **start** so that it should generate that much row sized pattern if we increase the number of rows.

Approach to solution

We need to generalize the non-repeating numeric pattern of the column **variable: start** using the available variables. We can use the **variable: totalRows**. As an input, we had been assigned the value 5, the **total number of rows**. Let's try to derive the consistent logical expression for the column **variable: start**.

ROW LEVEL FOR LOOP	COLUMN LEVEL FOR LOOP	
Variable: row	Variable: col	
start = 1 end = 5	end	start
row = 1	col = 5 = totalRows = $(\text{totalRows}+1)-1$	col = 1
row = 2	col = 4 = totalRows-1 = $(\text{totalRows}+1)-2$	
row = 3	col = 3 = totalRows-2 = $(\text{totalRows}+1)-3$	
row = 4	col = 2 = totalRows-3 = $(\text{totalRows}+1)-4$	
row = 5	col = 1 = totalRows-4 = $(\text{totalRows}+1)-5$	

The Logical expression in the column **variable: start** is partially consistent. We can observe, it has a non-repeating numeric pattern partially.

ROW LEVEL FOR LOOP	COLUMN LEVEL FOR LOOP	
Variable: row	Variable: col	
start = 1 end = 5	end	start
row = 1	col = 5 = $(\text{totalRows} + 1) -$	1
row = 2	col = 4 = $(\text{totalRows} + 1) -$	2
row = 3	col = 3 = $(\text{totalRows} + 1) -$	3
row = 4	col = 2 = $(\text{totalRows} + 1) -$	4
row = 5	col = 1 = $(\text{totalRows} + 1) -$	5

The partially non-repeating numeric pattern has the same numeric value pattern as the **variable: row**.

ROW LEVEL FOR LOOP	COLUMN LEVEL FOR LOOP	
Variable: row	Variable: col	
start = 1 end = 5	end	start
row = 1	col = 5 = $(\text{totalRows} + 1) -$	1
row = 2	col = 4 = $(\text{totalRows} + 1) -$	2
row = 3	col = 3 = $(\text{totalRows} + 1) -$	3
row = 4	col = 2 = $(\text{totalRows} + 1) -$	4
row = 5	col = 1 = $(\text{totalRows} + 1) -$	5

Further, we can express it in terms of **variable: totalRows** & **variable: row**.

ROW LEVEL FOR LOOP		COLUMN LEVEL FOR LOOP		
Variable: row		Variable: col		
start = 1	end = 5	end		start
row = 1		col = 5 = (totalRows + 1) -	row	col = 1
row = 2		col = 4 = (totalRows + 1) -	row	
row = 3		col = 3 = (totalRows + 1) -	row	
row = 4		col = 2 = (totalRows + 1) -	row	
row = 5		col = 1 = (totalRows + 1) -	row	

Now, **variable: start** has the same logical expression and is consistent for every row value.

ROW LEVEL FOR LOOP		COLUMN LEVEL FOR LOOP		
Variable: row		Variable: col		
start = 1	end = 5	end		start
row = 1		col = 5		
row = 2		col = 4		
row = 3		col = 3 (totalRows + 1) - row		col=1
row = 4		col = 2		
row = 5		col = 1		

Mathematically, it can be given as

$$1 \leq col \leq (\text{totalRows} + 1) - row$$

Where **row = 1 to 5**

The consistent logical expression (**totalRows + 1**) is the reason to produce dynamic output at the column level for different values of **variable: row**. Extracting the consistent logical expression is a way to generalize and to get a customized solution. When we are able to establish a relationship between variables to get the consistent logical expression, then we can have a **True** chance to automate the process of computing. We can clearly define the looping action of column level **for-loop**.

COLUMN LEVEL FOR LOOP OF INCREMENTAL ORDER

Column level Repetitive Action

```
for( col = 1 col <=(totalRows + 1)- row col = col - 1)
    Print character *
```

ELEMENTS OF COLUMN LEVEL FOR LOOP

Variable: col		DECREMENTING ORDER	ACTION
start	end	step-size	Print character
1	(totalRows + 1) - row	1	*

ROW LEVEL REPETITIVE ACTION

The only action expected from **row level for-loop** in our case is to shift the program's control to the next row. It should be done based on the **total number of rows** inputted by the **variable: totalRows**. It means that during the program's execution, when **variable: totalRows** contains the value **3**, then the **row level for-loop** should **end** after **3** repetitions.

For **5** repetitions of **row level for-loop**, the program's control moves to the next rows **3** times or **variable: totalRows** should contain value **5**.

For **7** repetitions of **row level for-loop**, the program's control moves to the next rows **7** times or **variable: totalRows** should contain value **7**.

For **10** repetitions of **row level for-loop**, the program's control moves to the next rows **10** times or **variable: totalRows** should contain value **10**.

ROW LEVEL FOR LOOP					
INPUT	Variable: row		INCREMENTAL ORDER		
TOTAL Rows	start	end	STEP-SIZE = 1		
totalRows = 3	1	3	Move To Next Row	3	times
totalRows = 5		5	Move To Next Row	5	times
totalRows = 7	CONSISTENT	7	Move To Next Row	7	times
totalRows = 10	NUMERIC EXPRESSION	10	Move To Next Row	10	times

We can find a column where a logical expression is consistent.



ROW LEVEL FOR LOOP					
INPUT	Variable: row		INCREMENTAL ORDER		
TOTAL Rows	start	end	STEP-SIZE = 1		
totalRows = 3	1	totalRows	Move To Next Row	totalRows	times
totalRows = 5		totalRows	Move To Next Row	totalRows	times
totalRows = 7		totalRows	Move To Next Row	totalRows	times
totalRows = 10		totalRows	Move To Next Row	totalRows	times

Clearly, the stored value of **totalRows** will act as the upper bound for the **row level for-loop** repetitions. It formulates a flexible solution to resize the length of the pattern in the run-time based on the given input value of the **variable: totalRows**.

We can now abstractly define the complete action of row level ***for-loop***.

ELEMENTS OF ROW LEVEL FOR LOOP			
Variable: row		INCREMENTAL ORDER	ACTION
start	end	step-size	Statements
1	totalRows	1	1# COLUMN LEVEL For Loop 2# Move To Next Row

ROW LEVEL FOR LOOP OF INCREMENTAL ORDER

Row level Repetitive Action

```

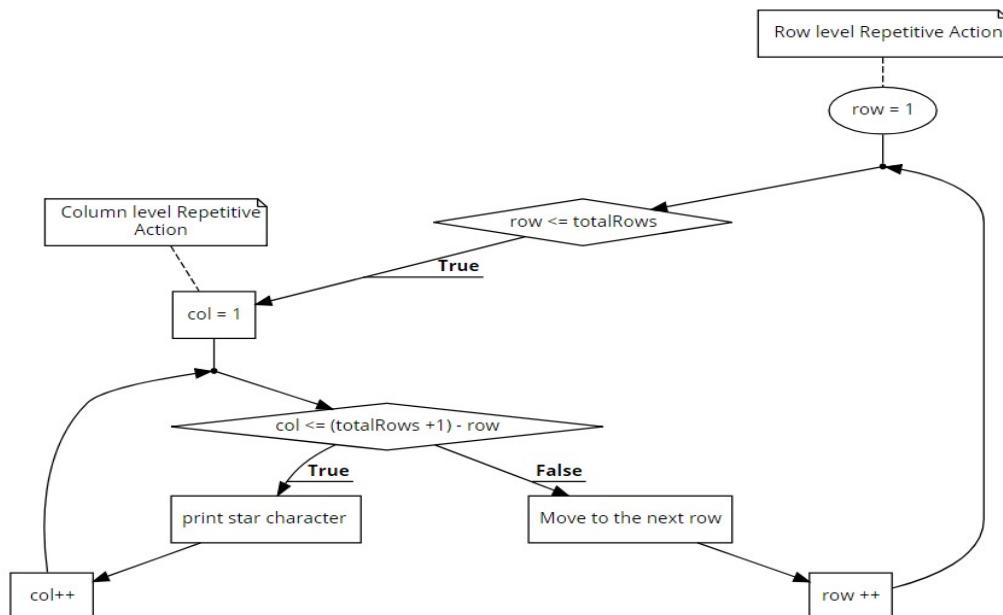
for row in range(1, totalRows + 1):
    1.# Column level Repetitive Action :
        for col in range(1, (totalRows + 1 - row)+ 1):
            print * character

    2.Move to next row

```

FLOW CHART

We got everything related to ***for-loop***.



Let's execute the following program.

PatternB1.py

```
totalRows = 5 # number of rows to display

    # Row level Repetitive Action :

    # Action1. Executes Column level Repetitive Action:

    # Action2. Move to next row to print it further.

for row in range(1,totalRows + 1):

    # Column level Repetitive Action :

    for col in range(1,(totalRows + 1 - row)+ 1):

        # print * character in the same row

        print("*",end="")



print()# Move to next row
```

Output: totalRows = 5

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

```

Output: totalRows = 10

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

Alternative solution2

From the previous solution, we got the following **Column level Repetitive Action** details as follows:

Let's try to simplify the expression `(totalRows +1) - row`.

ROW LEVEL FOR LOOP	COLUMN LEVEL FOR LOOP		
Variable: row	Variable: col		
start = 1 end = 5	end		start
row = 1	col = 5	= (totalRows + 1) -	row col = 1
row = 2	col = 4	= (totalRows + 1) -	row
row = 3	col = 3	= (totalRows + 1) -	row
row = 4	col = 2	= (totalRows + 1) -	row
row = 5	col = 1	= (totalRows + 1) -	row

Subtracting 1 from row and column **end**, since its logical expression is impacted by the **variable: row**.

ROW LEVEL FOR LOOP	COLUMN LEVEL FOR LOOP		
Variable: row	Variable: col		
start = 1 end = 5	end		start
row = 1 - 1	col = 5	= (totalRows + 1) -	row - 1 col = 1
row = 2 - 1	col = 4	= (totalRows + 1) -	row - 1
row = 3 - 1	col = 3	= (totalRows + 1) -	row - 1
row = 4 - 1	col = 2	= (totalRows + 1) -	row - 1
row = 5 - 1	col = 1	= (totalRows + 1) -	row - 1

ROW LEVEL FOR LOOP	COLUMN LEVEL FOR LOOP		
Variable: row	Variable: col		
start = 1 end =(totalRows - 1)	end		start
row = 0	col = 5	= totalRows -	row col = 1
row = 1	col = 4	= totalRows -	row
row = 2	col = 3	= totalRows -	row
row = 3	col = 2	= totalRows -	row
row = 4 =(totalRows - 1)	col = 1	= totalRows -	row

Now, **column: end** has the same expression, and it is consistent for every value of the **variable: row**. We can define column level **for-loop** while considering every row value.

ELEMENTS OF COLUMN LEVEL FOR LOOP			
Variable: col		DECREMENTING ORDER	ACTION
start	end	step-size	Print character
1	(totalRows - row)	1	*

Outer **for-loop** starts at row = 0 & will repeat till row <=(**totalRows** – 1) or row < **totalRows**.

ELEMENTS OF ROW LEVEL FOR LOOP			
Variable: row		INCREMENTAL ORDER	ACTION
start	end	step-size	Statements
row = 0	row < totalRows	1	1# COLUMN LEVEL For Loop 2# Move To Next Row

The new solution changes are in the following program.

PatternB2.py

```
totalRows = 5 # number of rows to display

    # Row level Repetitive Action :

    # Action1.Executes Column level Repetitive Action

    # Action2.Move cursor to next row.

for row in range(0, totalRows ):

    # Column level Repetitive Action :

    for col in range(1,( totalRows - row )+ 1):

        # print * character in the same row

        print("*",end="")



    print()# Move to the next row
```

Output: totalRows = 5

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

```

Output: totalRows = 10

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

Please verify the output for the different number of rows; let it be 10.

Alternative solution3

The next observation table is the one among several logic trace tables we summarized in the very beginning trying to find a solution to this pattern problem.

INPUT	ROW LEVEL FOR LOOP		COLUMN LEVEL FOR LOOP	
TOTAL ROWS = 5	Variable: row		Variable: col	
PRPATTERN	start = 1	end = 5	start	end
*****	row = 1		col = 1	col = 5
****	row = 2			col = 4
***	row = 3		consistent	col = 3
**	row = 4		numerical	col = 2
*	row = 5		expression	col = 1

If we observe the numeric pattern for rows (1, 2, 3, 4, and 5), it is the reverse of the numeric pattern of the column: end (5, 4, 3, 2, and 1). If we just reverse the counter of the row variable, it will match exactly the values of the column max_val (5, 4, 3, 2, and 1). Instead of using the incrementing for-loop for the row variable, we need to replace it with the decrementing for-loop for the row variable. Reversing the flow of values for the **variable: row**. It can be given as follows:

INPUT	ROW LEVEL FOR LOOP		COLUMN LEVEL FOR LOOP	
TOTAL ROWS = 5	Variable: row		Variable: col	
PRPATTERN	start = 5	end = 1	start	end
*****	row = 5		col = 1	col = 5
****	row = 4			col = 4
***	row = 3		consistent	col = 3
**	row = 2		numerical	col = 2
*	row = 1		expression	col = 1

Now the numeric pattern of variable row and **column: end** matches exactly. So, **column: end** can be expressed in the **variable: row**. So, **column: end** is consistent for every value of the **row**. Now, we are in a good position since the **for-loop** requires one minimum value or a logical expression and one maximum value or a logical expression.

INPUT	ROW LEVEL FOR LOOP		COLUMN LEVEL FOR LOOP	
TOTAL ROWS = 5	Variable: row		Variable: col	
PRPATTERN	start = 5	end = 1	start	end
*****	row = 5			row
****	row = 4			row
***	row = 3		col = 1	row
**	row = 2			row
*	row = 1			row

We can now define column level incrementing **for-loop**.

ELEMENTS OF COLUMN LEVEL FOR LOOP			
Variable: col		INCREMENTING ORDER	ACTION
start	end	step-size	Print character
col = 1	col <= row	1	*

We can now define row level **decrementing for-loop**.

It starts at the current value of the variable: **totalRows** and repeatedly decreases till it becomes 1. Please verify results with the mentioned updated program.

ELEMENTS OF ROW LEVEL FOR LOOP			
Variable: row		DECREMENTAL ORDER	ACTION
start	end	step-size	Statements
row = totalRows	row >= 1	1	1# COLUMN LEVEL For Loop 2# Move To Next Row

PatternB3.py

```
totalRows = 10 # 10 rows to display

    # Row level Repetitive Action :

    # Action1. Executes Column level Repetitive Action:

    # Action2. Move cursor to next row.

for row in range(totalRows + 1, 1,-1):

    # Column level Repetitive Action :

    for col in range(row, 1,-1):

        # print * character in the same row

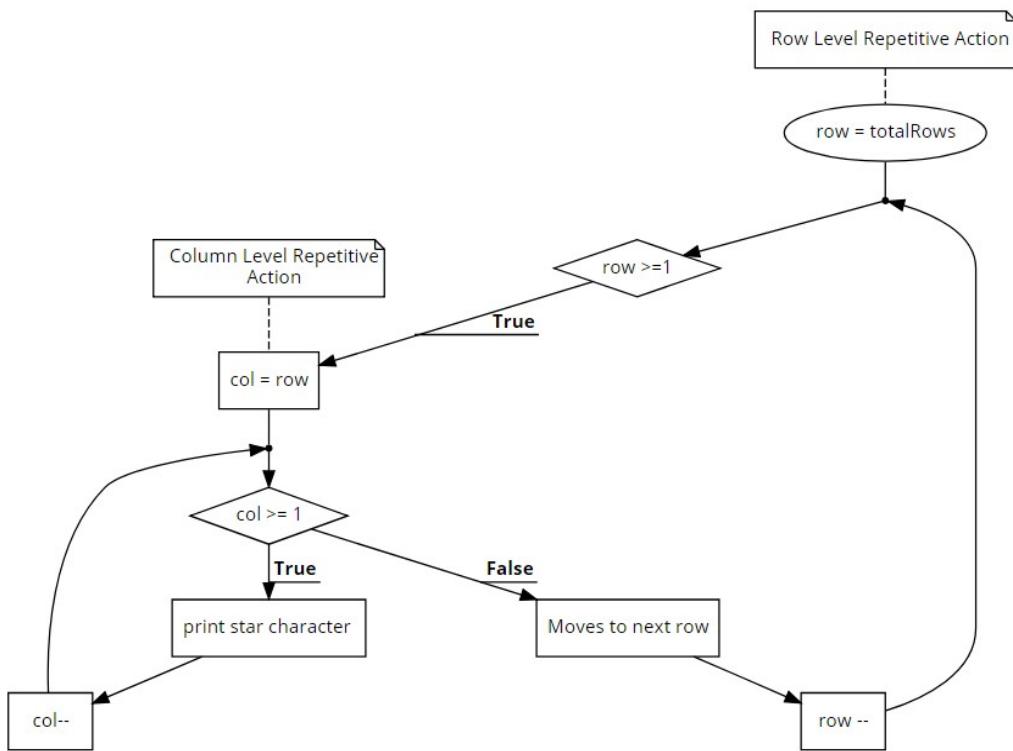
        print("*", end="")

    print()# Move to the next row
```

Output

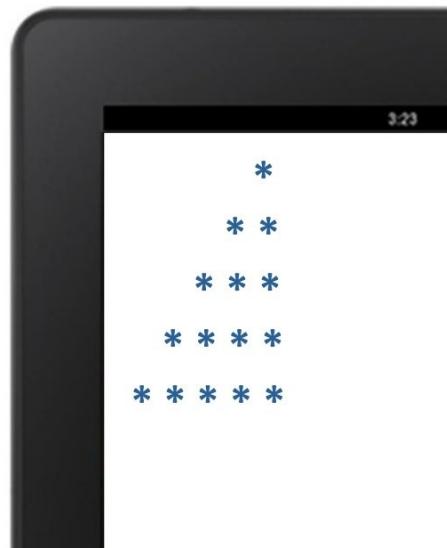
```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*
*
```

Flow Chart Problem: The given following flow-chart has every information related to the outer and nested **FOR-LOOP**. PLEASE WRITE A PROGRAM WHICH SUITS THE GIVEN FLOW-CHART.



Pattern 1C

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. Thus, the address in each smallest box on the display screen will logically become a unique (row, column) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread across the screen.



Data Representation

Although logically, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (row, column) numeric paired values.

Row, Col	Col = 1	Col = 2	Col = 3	Col = 4	Col = 5
Row = 1					* (1, 5)
Row = 2				* (2, 4)	* (2, 5)
Row = 3			* (3, 3)	* (3, 4)	* (3, 5)
Row = 4		* (4, 2)	* (4, 3)	* (4, 4)	* (4, 5)
Row = 5	* (5, 1)	* (5, 2)	* (5, 3)	* (5, 4)	* (5, 5)

The pair (4, 5) represents a single character * at the assumed position of 4th row and 5th column on the display screen. The set of all value pairs (row = value, col = value) now represent this particular pattern.

Analysis of output

Computer dominantly works on very precise calculations. One interesting way to practice programming is to try converting manual steps of writing a given pattern into programming instructions. We can use paper or a spreadsheet. To analyze a pattern correctly, we can mark row numbers and column numbers to identify position-indicator logically.

	Col = 1	Col = 2	Col = 3	Col = 4	Col = 5	WRITING ACTION
ROW	Move to Row = 1					1 →* 4 spaces Write * 1 time
MOVEMENT	Move to Row = 2					1 →* 2 →* 3 spaces Write * 2 times
ACTION	Move to Row = 3					1 →* 2 →* 3 →* 2 spaces Write * 3 times
	Move to Row = 4			1 →* 2 →* 3 →* 4 →* 1 space		Write * 4 times
	Move to Row = 5	1 →*	2 →*	3 →*	4 →*	5 →* 0 space Write * 5 times

Space is also a character like any other keyboard character. If a word or single non-spacing character is displayed on the screen after a gap, it means one or many space characters have to be printed before printing any other non-spacing character(s).

Steps for pattern generation		Computer Screen
For row = 1	Print space “” <u>4 times</u> in the same 1 st row. Print star “*” <u>once</u> in the same 1 st row and then move to the next row number 2.	
For row = 2	Print space “” <u>3 times</u> in the same 2 nd row. Print star “*” <u>2 times</u> in the same 2 nd row and then move to the next row number 3.	
For row = 3	Print space “” <u>2 times</u> in the same 3 rd row. Print star “*” <u>3 times</u> in the same 3 rd row and then move to the next row number 4.	
For row = 4	Print space “” <u>1 time</u> in the same 4 th row. Print star “*” <u>4 times</u> in the same 4 th row and then move to the next row number 5.	
For row = 5	Print space “” <u>0 times</u> in the same 5 th row. Print star “*” <u>5 times</u> in the same 5 th row and then move to the next row number 5.	

Let's convert these steps into a program using the following inbuilt methods.

To move the cursor to the <u>next row</u> .	<code>print()</code>
Print space “” character in the same row.	<code>print(" ", end = "")</code>
Print star “*” character in the same row.	<code>print("*", end = "")</code>

In order to print space “” or “*” character, each movement on the computer screen will be a single keyboard character-sized jump from left to the right direction in the same row—these three types of programming statements required to get repeat many times using looping. Writing a program to calculate the row and column values to Print characters correctly on the computer screen will be the main reason for improving logical thinking.

COLUMN LEVEL REPETITIVE ACTION

Let's consider;

row → variable type of integer to store integer value one at a time and represents row number between 1 and 5.

col → variable type of integer to store integer value one at a time and represents column number between 1 and 5.

spaces → variable type of integer to store integer value one at a time and represents space character at column number between 1 and 5.

totalRows → variable type of integer to store integer value for the total number of rows.

Logic Trace Tables

(ROW, COL) VALUE PAIR DEPICTS THE POSITION OF "*" CHARACTER ON THE DISPLAY SCREEN				
				row=1, col=5
			row=2, col=4	row=2, col=5
		row=3, col=3	row=3, col=4	row=3, col=5
	row=4, col=2	row=4, col=3	row=4, col=4	row=4, col=5
row=5, col=1	row=5, col=2	row=5, col=3	row=5, col=4	row=5, col=5

ROW LEVEL REPETITIVE ACTION				PATTERN
Variable: row	Column Level Repetitive Action			PATTERN
	Variable: spaces	Variable: col	Character count	
1	1,2,3,4	5	4 Spaces + 1 star	*
2	1,2,3	4, 5	3 Spaces + 2 stars	**
3	1,2	3, 4, 5	2 Spaces + 3 stars	***
4	1	2, 3, 4, 5	1 Spaces + 4 stars	****
5	0(no space)	1, 2, 3, 4, 5	0 Spaces + 5 stars	*****

Space ("") and star("*) characters are repeated many times. So, based on the count of the space ("") and star(*) characters, we can use **for-loop** for each. We can put our initial observation in somewhat more clear form as given in the following table.

ROW LEVEL REPETITIVE ACTION					PATTERN
Variable: row	Column Level Repetitive Action				PATTERN
	Variable: spaces		Variable: col		
	min_val	max_val	min_val	max_val	
1	1	4	1	1	*
2	1	3	1	2	**
3	1	2	1	3	***
4	1	1	1	4	****
5	0	0	1	5	*****

Approach to solution

The variables are **row** & **totalRows=5**.

We need to generalize the non-repeating numeric pattern of the column **max_val**. The numeric pattern of the **variable col's column max_val** matches exactly every value of the **row**. So, all the numeric patterns of **variable col's column max_val** can be expressed in row variable.

Let's try to use variable **totalRows** to generalize the variable **spaces' column max_val**.

Row level Repetitive Action					
Variable: row	Column level Repetitive Action				
	Variable: spaces		Variable: col		
	min_val	default	min_val	min_val	max_val
1	1	4	totalRows - 1	1	row
2	1	3	totalRows - 2	1	row
3	1	2	totalRows - 3	1	row
4	1	1	totalRows - 4	1	row
5	0	0	totalRows - 4	1	row

Part of variable **spaces** column **max_val's** expression follows the same numeric value pattern as a **row**. Further, we can express it in terms of variables **totalRows & row**.

Row level Repetitive Action					
Variable: row	Column level Repetitive Action				
	Variable: spaces		Variable: col		
	min_val	default	min_val	min_val	max_val
1	1	4	totalRows - row	1	row
2	1	3	totalRows - row	1	row
3	1	2	totalRows - row	1	row
4	1	1	totalRows - row	1	row
5	0	0	totalRows - row	1	row

We generalized the non-repeating numeric pattern of the column **max_val** into an expression.

We can now define column level **for-loop**, which is printing spaces for every row value.

ELEMENTS OF COLUMN LEVEL FOR LOOP			
Variable: col		INCREMENTING ORDER	ACTION
min_val	max_val	step-size	Print character
spaces = 1	spaces <= totalRows — row	1	space

We can now define column level **for-loop**, which is printing (*) characters for every row value.

ELEMENTS OF COLUMN LEVEL FOR LOOP			
Variable: col		INCREMENTING ORDER	ACTION
min_val	max_val	step-size	Print character
col = 1	col <= row	1	*

ROW LEVEL REPETITIVE ACTION

The same following statements had been repeated 5 times(total number of rows).

Column level for-loop for printing space characters # inner for-loop

Column level for-loop for printing star characters # inner for-loop

print()# Move to the next row

We have assigned a unique value to each row starting from 1 to 5. So, it's an easy take now.

The minimum and maximum value can be given for the **for-loop** as

min_val = 1 & max_val = 5 or totalRows

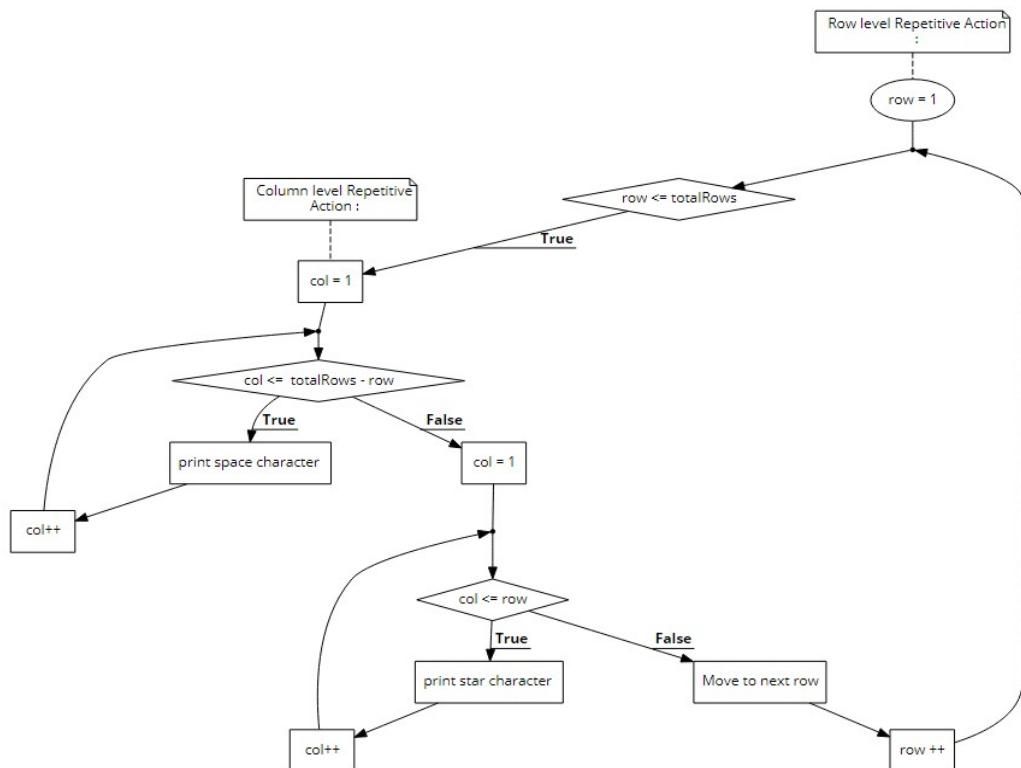
Let's consider;

row → variable type of integer to store integer value one at a time and represents row number between 1 and 5(**totalRows**). The **totalRows** has initially assigned the value 5.

Outer **for-loop** starts at row=1 & will repeat till row <= **totalRows**.

We got everything related to **for-loop** for row level repetition.

ELEMENTS OF ROW LEVEL FOR LOOP				
Variable: row		INCREMENTAL ORDER	ACTION	
start	end	step-size	Statements	
<code>row = 1</code>	<code>row = totalRows</code>	<code>1</code>	<code>1#</code>	COLUMN LEVEL For Loop for printing spaces
			<code>2#</code>	COLUMN LEVEL For Loop for printing *
			<code>3#</code>	Move To Next Row



Let's write the complete programming logic and execute the program.

PatternC1.py

```
totalRows = 5 # number of rows to display

    # Row level Repetitive Action :
    # Action1.Executes Column level Repetitive Action
    # Action2.Move cursor to next row.

for row in range(1, totalRows + 1):

    # Column level Repetitive Action :
    for sp in range(1,( totalRows - row )+ 1 ):

        # Action1.Move cursor in the same row
        # Action2.print space character
        print("", end = "")# extra space for formatting

    for col in range(1, row + 1):

        # Action1.Move cursor in the same row
        # Action2.print star * character
        print("*", end = "")# extra space for formatting

    print()# Move to the next row
```

Output

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

Alternative solution2

The available variables are **row & totalRows=5**. The variable:col = 1, 2, 3, 4, 5, takes the value 1 to 5. The overall observation is that at each row, either it's printing space characters or star (*) characters continuously. Based on the type of printing the characters, the columns are grouped into two parts either by printing space "" character(s) or star("*) character(s).

Row level Repetitive Action				Expected Output
Variable: row	Column level Repetitive Action			
	Variable: spaces	Variable: col	Character count	
1	1,2,3,4	5	4 Spaces + 1 star = 5 characters	*
2	1,2,3	4, 5	3 Spaces + 2 stars = 5 characters	**
3	1,2	3, 4, 5	2 Spaces + 3 stars = 5 characters	***
4	1	2, 3, 4, 5	1 Spaces + 4 stars = 5 characters	****
5	0(no space)	1, 2, 3, 4, 5	0 Spaces + 5 stars = 5 characters	*****

The printing of space(s) comes first before the Print star("*) character(s). When the care of the condition till it Print space character(s), then the remaining are the star("*) character(s). If we closely observe, it's cutting the values of variable col into two halves. One part goes to the printing of space character(s), and the other goes to the printing of star character(s).

Row level Repetitive Action			Expected Output
Variable: row	Column level Repetitive Action		
	Variable: spaces	Variable: col	

	min_val	max_val	min_val	max_val	
1	1	4	5	5	*
2	1	3	4	5	**
3	1	2	3	5	***
4	1	1	2	5	****
5	0	0	1	5	*****

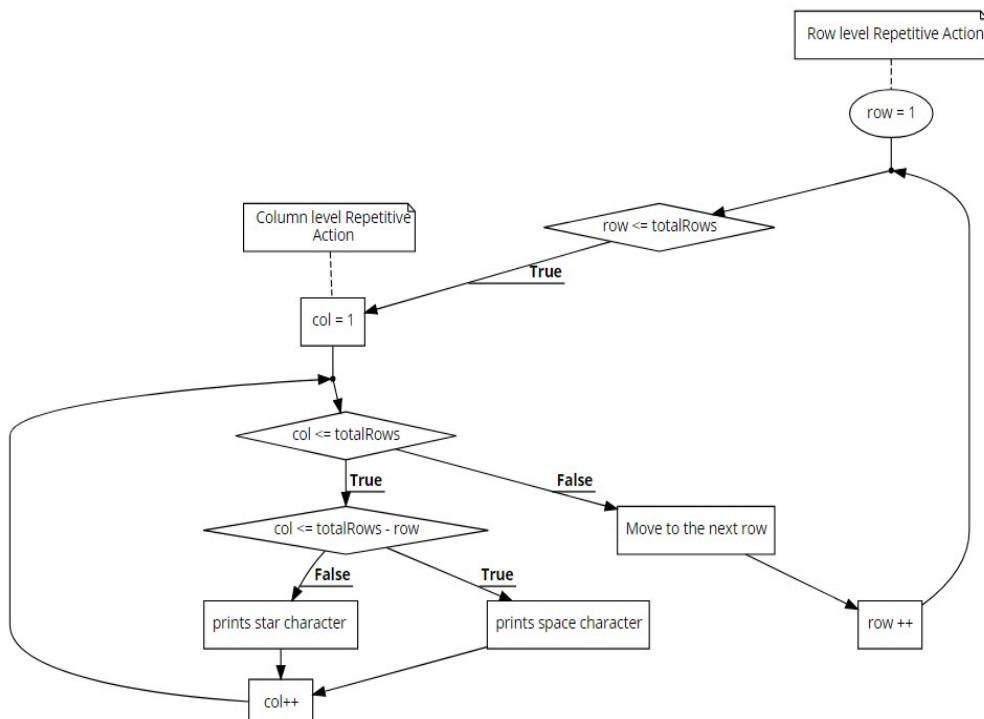
Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: spaces		Variable: col		
	min_val	max_val	min_val	max_val	
1	1	$4 = 5 - 1 = \text{totalRows} - 1$	5	5	*
2	1	$3 = 5 - 2 = \text{totalRows} - 2$	4	5	**
3	1	$2 = 5 - 3 = \text{totalRows} - 3$	3	5	***
4	1	$1 = 5 - 4 = \text{totalRows} - 4$	2	5	****
5	0	$0 = 5 - 5 = \text{totalRows} - 5$	1	5	*****

We partially got a generalized logical expression for printing spaces from the previous logic trace table. The column **max_val**'s numeric pattern of **variable: spaces** matches with the numeric occurrences of **variable: row** and hence got the logical condition (**totalRows - row**) as shown below.

Row level Repetitive Action					
Variable: row	Column level Repetitive Action				
	Variable: spaces		Variable: col	min_val	max_val
	min_val	max_val			
1	1	totalRows - 1 = totalRows - row		5	5
2	1	totalRows - 2 = totalRows - row		4	5
3	1	totalRows - 3 = totalRows - row		3	5
4	1	totalRows - 4 = totalRows - row		2	5
5	0	totalRows - 5 = totalRows - row		1	5

The other **for-loop** runs between 1 and (**totalRows = 5**) for **variable: col**. The logical expression (**totalRows - row**) gives the maximum count starting from count 1 to print that many space characters. So, we can use **if-else** for conditional filtering and printing character (space or star).

Print character space when the condition is True for every value of the variable: col				
col=1	col=2	col=3	col=4	col=5
If (condition is True)				If (col<= totalRows - row) is True
Print space character				Print space character
Else				Else
Print star character				Print star character



The updated program with if-else statement will follows next.

PatternC2.py

```
totalRows = 5 # number of rows to display

# Row level Repetitive Action :

# Action1. Executes Column level Repetitive Action:

# Action2.Move cursor to next row.

for row in range(1, totalRows + 1):

    # Column level Repetitive Action :

    # Action1. Print characters column-wise or in the same row.

    for col in range(1, totalRows + 1):

        if (col <=(totalRows - row)):

            print("", end = "")# Action2.prints space character

        else:

            print("*", end="")# Action3.prints star * character

    print()# move control of program to the next row

# to print characters in a new line.
```

Output: totalRows = 5

```
*
```



```
**
```



```
***
```



```
****
```



```
*****
```



```
*****
```



```
*****
```



```
*****
```



```
*****
```



```
*****
```

Output: totalRows = 10

```
*
```



```
**
```



```
***
```



```
****
```



```
*****
```



```
*****
```



```
*****
```



```
*****
```



```
*****
```



```
*****
```

Now, test for different number of lines, let it be 10

Alternative solution3

From the previous solution, we have checked the **if-condition** to be True to Print spaces, and in the else part of the **if-statement**, we have printed the star("*") characters. In this solution, we will reverse the situation. When **if-condition** becomes True, we will print a star("*") character, and in the else part of the **if-else statement**, we will print spaces.

Print character star when the condition is True for every value of the variable: col				
col=1	col=2	col=3	col=4	col=5
If (condition is True)		If (?) is True		
Print star character		Print star character		
Else		Else		
Print space character		Print space character		

Let's try to re-learn from the given observation table that we had used in the previous solution.

Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: spaces		Variable: col		
	min_val	max_val	min_val	max_val	
1	1	4	5	5	*
2	1	3	4	5	**
3	1	2	3	5	***
4	1	1	2	5	****
5	0	0	1	5	*****

We know that totalRows = 5.

The column **min_val** for printing star character(s) can be given as;

Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: spaces		Variable: col		
	min_val	max_val	min_val	max_val	
1	1	4	5 - 0	5 = totalRows	*
2	1	3	5 - 1	5 = totalRows	**
3	1	2	5 - 2	5 = totalRows	***
4	1	1	5 - 3	5 = totalRows	****
5	0	0	5 - 4	5 = totalRows	*****

If we can try variable: totalRows to generalize the numeric pattern of the column **min_val**, then it will be $\text{totalRows} - 0, \text{totalRows} - 1, \dots, \text{totalRows}$. This expression nearly follows the numeric pattern of the variable row = 1,2,...,5. The former is having the numeric pattern starting from zero(0) and **variable: row's** numeric pattern starting from one(1). We can start the numeric pattern of **variable: row** from zero(0). In the case of row level movements, whether the range is 1 to 5 or 0 to 4, it only means five iterations.

Let's make use of the variables **totalRows & row** to derive the generalized logical expression for printing character star for the **columns min_val & max_val** given as follows

Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: spaces		Variable: col		
	min_val	max_val	min_val	max_val	
0	1	4	totalRows - row	totalRows	*
1	1	3	totalRows - row	totalRows	**
2	1	2	totalRows - row	totalRows	***
3	1	1	totalRows - row	totalRows	****
4	0	0	totalRows - row	totalRows	*****

The condition (`col >=(totalRows - row)`) gives the minimum limit, a starting point from which it will start printing star("*) character(s) till the maximum value of variable col, which is **totalRows**.

The **if-condition** can be given for in between range of values (`totalRows - row`) and **totalRows** using the comparison and logical operators as:

If (`col >=(totalRows - row)` and `col <= totalRows`) and moreover we already know that

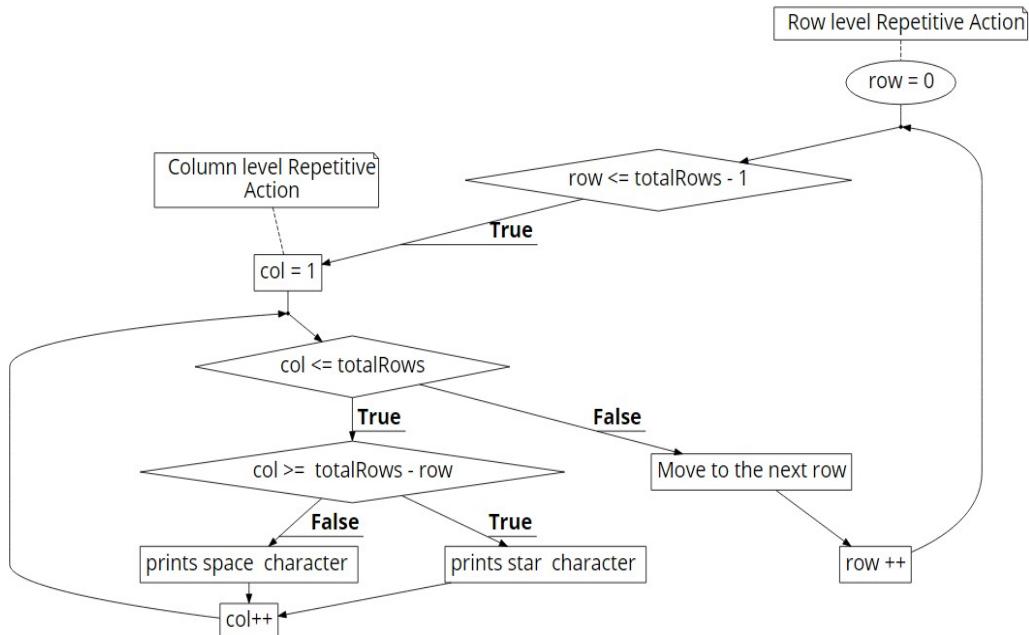
variable:col = 1,2,3,4,5 or 1,2,3,4, **totalRows**

variable: row = 0,1,2,3,4 or 0,1,2,3,4, (**totalRows - 1**).

We can eliminate the logical AND(and) condition(`col <= totalRows`),since this condition is already been taken care in the column level inner for-loop. So, the if-condition will be reduced to

`If (col >=(totalRows — row))`

We can summarize it all in the following flowchart and update it in the program.



Test for the total number of lines equals 7.

PatternC3.py

```
totalRows = 7 # 7 rows to display

# Row level Repetitive Action

# Action1. Executes Column level Repetitive Action:

# Action2. Move cursor to next row.

for row in range( 0, totalRows ):

    # Column level Repetitive Action

    # Action1. Print characters column-wise or in the same row.

    for col in range(1, totalRows + 1):

        if (col >=(totalRows - row)):

            # Action2.prints star * character

            print("*",end="")

        else:

            # Action3.prints space character

            print(" ",end="")

    print()# Move cursor to the next row
```

Output

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

```
*****
```

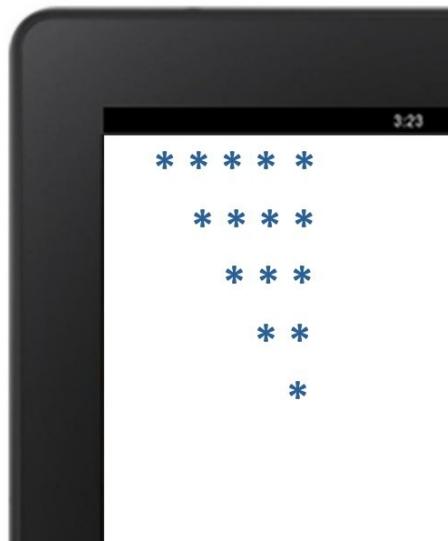
```
*****
```

Problem: Write a program to create prpattern1C using **decrementing for-loop** in the following mentioned two cases.

Case A	Case B
If (condition == True) Print space character Else Print star character	If (condition == True) Print star character Else Print space character

Pattern1D

Our programming job is to print this particular logic-based pattern on one part of the computer screen. **Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes** —the smallest box treated as a resident of just a single character only. Thus, the address in each smallest box on the display screen will logically become a unique (**row, column**) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all over the screen.



Data Representation

Although logically, we can exactly locate the position of each printable character on the display

Row, Col	Col = 1	Col = 2	Col = 3	Col = 4	Col = 5
Row = 1	* (1, 1)	* (1, 2)	* (1, 3)	* (1, 4)	* (1, 5)
Row = 2		* (2, 2)	* (2, 3)	* (2, 4)	* (2, 5)
Row = 3			* (3, 3)	* (3, 4)	* (3, 5)
Row = 4				* (4, 4)	* (4, 5)
Row = 5					* (5, 5)

The pair (4, 5) represents a single character * at the assumed position of 4th row and 5th column on the display screen. The set of all value pairs (row = value, col = value) now represent this particular pattern.

screen by knowing its uniquely identifiable (**row, column**) numeric paired values.

Analysis of output

Computer dominantly works on very precise calculations. One interesting way to practice programming is to try converting manual steps of writing a given pattern into programming instructions. We can use paper or a spreadsheet. To analyze a pattern correctly, we can mark row numbers and column numbers to identify position-indicator logically. **Space is also a character like any other keyboard character. If the spacing is displayed on the screen, it means one or many space characters have to be printed first before printing any other non-blank characters.**

	Col = 1	Col = 2	Col = 3	Col = 4	Col = 5	WRITING ACTION
ROW	Move to Row = 1	1 → *	2 → *	3 → *	4 → *	5 → * 0 space Write * 5 times
MOVEMENT	Move to Row = 2		1 → *	2 → *	3 → *	4 → * 1 space Write * 4 times
ACTION	Move to Row = 3			1 → *	2 → *	3 → * 2 spaces Write * 3 times
	Move to Row = 4				1 → *	2 → * 3 spaces Write * 2 times
	Move to Row = 5					1 → * 4 spaces Write * 1 time

Steps for pattern generation (Total Rows = 5)		Computer Screen
For row = 1	Print "*" <u>five times</u> in the same 1 st row and then move to row number 2.	A 10x10 grid with rows labeled 1 through 10 and columns labeled 1 through 10. Row 1 has five asterisks (*) in the first five columns. Row 2 is empty.
For row = 2	Print space "" <u>1 time</u> in the same 2 nd row. Print "*" <u>four times</u> in the same 2 nd row and then move to row number 3.	The 10x10 grid now shows Row 2 with four asterisks (*) in the first four columns. Row 3 is empty.
For row = 3	Print space "" <u>2 times</u> in the same 3 rd row. Print "*" <u>thrice</u> in the same 3 rd row and then move to row number 4.	The 10x10 grid now shows Row 3 with three asterisks (*) in the first three columns. Row 4 is empty.
For row = 4	Print space "" <u>3 times</u> in the same 4 th row. Print "*" <u>twice</u> in the same 4 th row and then move to row number 5.	The 10x10 grid now shows Row 4 with two asterisks (*) in the first two columns. Row 5 is empty.
For row = 5	Print space "" <u>4 times</u> in the same 5 th row. Print "*" <u>once</u> in the same 5 th row and then move to row number 6.	The 10x10 grid now shows Row 5 with one asterisk (*) in the first column. Row 6 is empty.

Let's convert steps into a program using the following inbuilt methods.

To move to the <u>next row</u>	<code>print()</code>
To move and to print space "" character in the <u>same row</u>	<code>print("", end = "")</code>
To move and to print star "*" character in the <u>same row</u>	<code>print("*", end = "")</code>

In order to print each space "" or "*" character, each movement on the computer screen will be a single keyboard character-sized jump from the left to the right direction in the same row. These three types of programming statements are required to get repeated many times in the program. It should happen in a way so that this particular printing pattern should appear on the screen. The way in our context mainly regards underlying logical expressions. A logical expression is just like a formula that we need to explore our-self. Since looping is meant for repetitive action within a program, we need to use these printing statements and logical expressions within the for-loop. Writing a program to calculate the row and column values to print characters correctly into the computer screen is an interesting exercise for improving logical thinking.

COLUMN LEVEL REPETITIVE ACTION

Let's consider;

row → variable type of integer to store integer value one at a time and represents row number between 1 and 5.

col → variable type of integer to store integer value one at a time and represents column number between 1 and 5.

spaces → variable type of integer to store integer value one at a time and represents space character at column number between 1 and 5.

totalRows → variable type of integer to store integer value for the total number of rows.

Logic Trace Tables

(row, col) value pair depicts the position of "*" character on the display screen

row=1, col=1	row=1, col=2	row=1, col=3	row=1, col=4	row=1, col=5
	row=2, col=2	row=2, col=3	row=2, col=4	row=2, col=5
		row=3, col=3	row=3, col=4	row=3, col=5
			row=4, col=4	row=4, col=5
				row=5, col=5

ROW LEVEL REPETITIVE ACTION				PATTERN
Variable: row	Column Level Repetitive Action			PATTERN
	Variable: spaces	Variable: col	Character count	
1	0(no space)	1, 2, 3, 4, 5	4 Spaces + 1 star	*****
2	1	2, 3, 4, 5	3 Spaces + 2 stars	****
3	1,2	3, 4, 5	2 Spaces + 3 stars	***
4	1,2,3	4, 5	1 Spaces + 4 stars	**
5	1,2,3,4	5	0 Spaces + 5 stars	*

ROW LEVEL REPETITIVE ACTION					PATTERN
Variable: row	Column Level Repetitive Action				PATTERN
	Variable: spaces		Variable: col		
	min_val	max_val	min_val	max_val	
1	0	0	1	5	*****
2	1	1	2	5	****
3	1	2	3	5	***
4	1	3	4	5	**

5

1

4

5

5

*

We know that **totalRows** = 5.

If we observe the numeric pattern for variable row (1, 2, 3, 4, and 5) is matching the numeric pattern of the column **min_val** of printing star character(s).

Our target is to express our program logic in terms of our chosen variables (**row**, **col**, **totalRows**).

Let's use these variables to find generalized expressions for printing star character(s).

ROW LEVEL REPETITIVE ACTION					PATTERN
Variable: row	COLUMN LEVEL REPETITIVE ACTION				
	Variable: spaces		Variable: col		
	min_val	max_val	min_val	max_val	
1	0	0	row	totalRows	*****
2	1	1	row	totalRows	****
3	1	2	row	totalRows	***
4	1	3	row	totalRows	**
5	1	4	row	totalRows	*

The condition (`col >= row`) gives the minimum limit, a starting point from which it will start printing star("*") character(s) till the maximum value of variable col, which is `totalRows`.

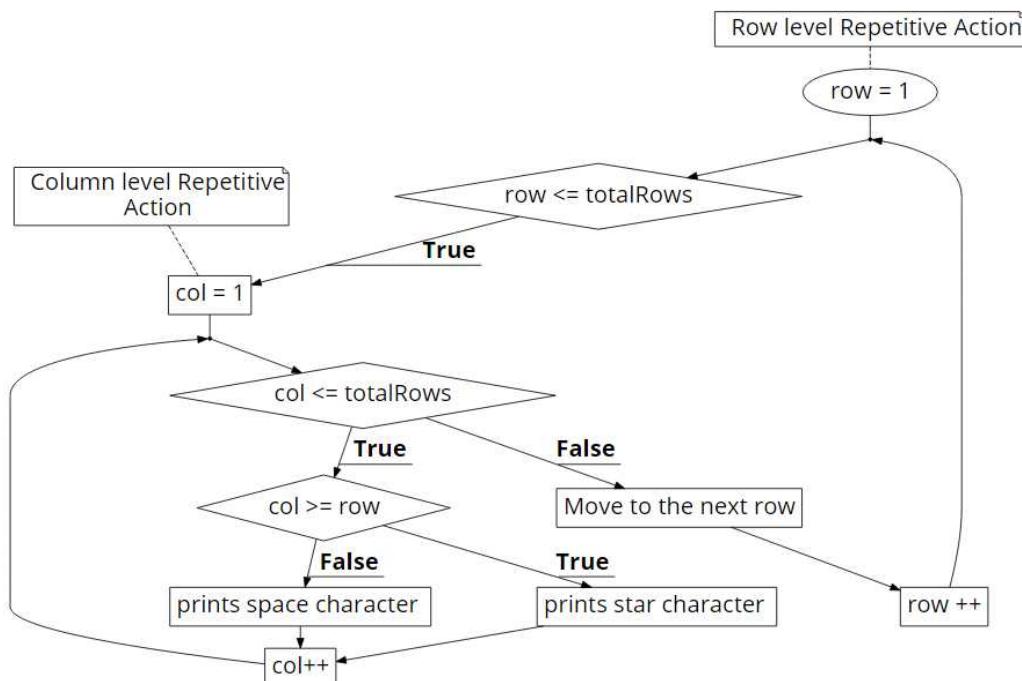
The if-condition can be given for the in-between range of values (`row`) and `totalRows` using the comparison and logical operators are given as follows:

If (`col >= row and col <= totalRows`).

We can eliminate the logical AND(and) condition(`col <= totalRows`)since this condition has already been taken care of in the column level inner for-loop. So, the **if-condition** can be given as;

If (`col >= row`)

We summarized our conclusions in a flow-chart which is given as follows:



The program can be written as follows:

```
PatternD1.py
totalRows = 5 # total number of rows to display

# Row level Repetitive Action :

# Action1. Executes Column level Repetitive Action:

# Action2. Move to next row to set ground for printing characters column-wise.

for row in range(1,totalRows+1):

    # Column level Repetitive Action :

    # Action1. Print characters column-wise or in the same row

    for col in range(1, totalRows + 1):

        if (col >= row):
            # Action2.prints star * character
            print("*",end="")
        else:
            # Action3.prints space character
            print(" ",end="")

    # move control of program to the next row
    # to switch printing of characters in a new line.

    print()
```

Output: totalRows = 5

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

```

Output: totalRows = 10

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

Now, test for a different number of lines. Let it be 10.

Problem D1: Write a program to generate the pattern **Pattern1D** by following the condition expression to print space characters. When the conditional expression is **True**, the program should Print space character and character star("*") when it's **False**.

Prints a character space when the condition is True for every value of the variable: col				
col=1	col=2	col=3	col=4	col=5
		If (condition=(?) is True)		
		Print space character		
Else				
		Print star character		

Problem D2. Write a program to generate the pattern **Pattern1D** by using the information given in the mentioned columns. Do not use the **variable: col**.

columns → Variable: row, Character count, and Total Characters.

The variable totalRows = 5 denotes the number of rows. The program should work for 10 rows also and which should give output by changing the variable **totalRows = 10**.

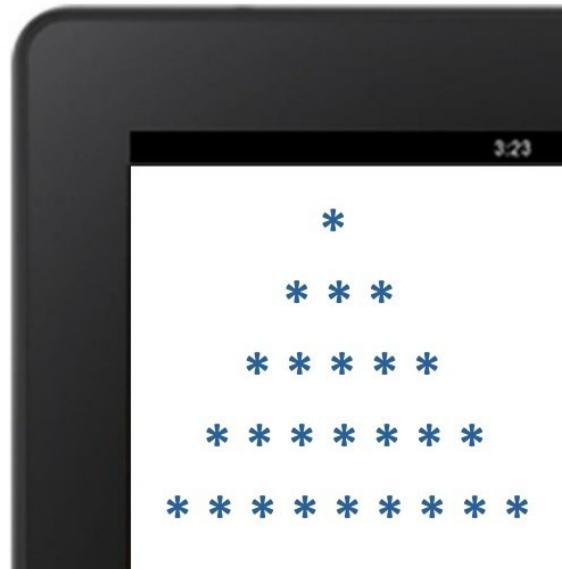
Row Level Repetitive Action					PATTERN
Variable: row	Column Level Repetitive Action				
	Variable: spaces	Variable: col	Character count	Total Characters	
1	0(no space)	1, 2, 3, 4, 5	4 Spaces + 1 star	5	*****
2	1	2, 3, 4, 5	3 Spaces + 2 stars	5	****
3	1,2	3, 4, 5	2 Spaces + 3 stars	5	***
4	1,2,3	4, 5	1 Spaces + 4 stars	5	**
5	1,2,3,4	5	0 Spaces + 5 stars	5	*

Problem D3: Write a program to generate the pattern **Pattern1D** using decrementing for-loop. When the result of conditional expression is **True**, the program should Print space character and character star("*") when it's **False**.

Prints a character space when the condition is True for every value of the variable: col				
col=1	col=2	col=3	col=4	col=5
		If (condition=(?) is True)		
		Print space character		
Else				
		Print star character		

Pattern1E

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique **(row, column)** numeric combination. The computer screen will simply become a unique combination of **(row, column)** numeric value paired boxes that are spread uniformly all over the screen.



Data Representation

Although logically, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable **(row, column)** numeric paired-values. For example; **(2, 4)** means character “*” at **2nd row** and **4th column** in the pattern.

The given set of row and col value pairs tries to mimic the location of each printable character of a given pattern on the device's display screen. The set of all value pairs (row = value, col = value) represent this particular pattern.

Row, Col	Col = 1	Col = 2	Col = 3	Col = 4	Col = 5	Col = 6	Col = 7	Col = 8	Col = 9
Row = 1					* (1, 5)				
Row = 2				* (2, 4)	*(2, 5)		* (2, 6)		
Row = 3			*(3, 3)	* (3, 4)	* (3, 5)		* (3, 6)	* (3, 7)	
Row = 4		* (4, 2)	* (4, 3)	* (4, 4)	* (4, 5)		* (4, 6)	* (4, 7)	* (4, 8)
Row = 5	* (5, 1)	* (5, 2)	* (5, 3)	* (5, 4)	* (5, 5)		* (5, 6)	* (5, 7)	* (5, 8)

Space is also a character like any other character. If a non-space character or a text is displayed on the screen after a blank area or gap, it means single or that many space characters have to be printed before printing any other non-space character(s).

Steps for pattern generation		Computer Screen
For row = 1	Print space “” <u>4 times</u> in the same 1 st row. Print star “*” <u>once</u> in the same 1 st row and then move to the next row number 2.	
For row = 2	Print space “” <u>3 times</u> in the same 2 nd row. Print star “*” <u>3 times</u> in the same 2 nd row and then move to the next row number 3.	
For row = 3	Print space “” <u>2 times</u> in the same 3 rd row. Print star “*” <u>5 times</u> in the same 3 rd row and then move to the next row number 4.	
For row = 4	Print space “” <u>1 time</u> in the same 4 th row. Print star “*” <u>7 times</u> in the same 4 th row and then move to the next row number 5.	
For row = 5	Print space “” <u>0 times</u> in the same 5 th row. Print star “*” <u>9 times</u> in the same 5 th row and then move to the next row number 6.	

Let's convert steps into a program using in-built methods

To move the cursor to the <u>next row</u>	<code>print()</code>
Move the cursor to the next (row, column) position and then Print space “” character in the <u>same row</u> .	<code>print(" ",end="")</code>
Move the cursor to the next (row, column) position and then Print star “*” character in the <u>same row</u> .	<code>print("*",end="")</code>

The above three types of programming statements are expected to get repeated many times in the program. Hence, we can use **for-loop** for the same.

COLUMN LEVEL REPETITIVE ACTION

Let's consider;

row → variable type of integer to store integer value one at a time and represents row number between 0 and 4.

col → variable type of integer to store integer value one at a time and represents column number between 1 and 5.

spaces → variable type of integer to store integer value one at a time and represents space character at column number between 1 and 5.

totalRows → variable type of integer to store integer value for the total number of rows.

(row, col) value pair depicts the position of “*” character on the display screen

			row=1, col=5				
			row=2, col=4	row=2, col=5	row=2, col=6		
		row=3, col=3	row=3, col=4	row=3, col=5	row=3, col=6	row=3, col=7	
	row=4, col=2	row=4, col=3	row=4, col=4	row=4, col=5	row=4, col=6	row=4, col=7	row=4, col=8
row=5, col=1	row=5, col=2	row=5, col=3	row=5, col=4	row=5, col=5	row=5, col=6	row=5, col=7	row=5, col=8

Row level Repetitive Action			Expected Output
Variable: row	Column level Repetitive Action		
	Variable: spaces	Variable: col	
1	1,2,3,4	5	*
2	1,2,3	4, 5,6	***
3	1,2	3, 4, 5,6,7	*****
4	1	2, 3, 4, 5,6,7,8	*****
5	0(no space)	1, 2, 3, 4, 5,6,7,8,9	*****

Space (“”) and star (“*”) characters are repeated many times.

So, we can use **for-loop** for each. We can fill the table as follows:

Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: spaces	Variable: col	min_val	max_val	
1	1	4	5	5	*
2	1	3	4	6	***
3	1	2	3	7	*****
4	1	1	2	8	*****
5	0	0	1	9	*****

We know that **totalRows = 5** (total number of rows). Let's try to use variable **totalRows** to generalize the variable **col**'s columns **min_val** & **max_val**. Also, we need to start the row value from 0 to 4 instead of 1 to 5. This is in order to match the part of the expression of **variable: col**'s columns **min_val** & **max_val**.

Row level Repetitive Action					Expected Output	
Variable: row	Column level Repetitive Action					
	Variable: spaces		Variable: col			
	min_val	max_val	min_val	max_val		
0	1	4	totalRows - 0	0	totalRows + 0	*
1	1	3	totalRows - 1	1	totalRows + 1	***
2	1	2	totalRows - 2	2	totalRows + 2	*****
3	1	1	totalRows - 3	3	totalRows + 3	*****
4	0	0	totalRows - 4	4	totalRows + 4	*****

Using variable **row** to generalize the variable **col**'s columns **min_val** & **max_val**.

Row level Repetitive Action					Expected Output	
Variable: row	Column level Repetitive Action					
	Variable: spaces		Variable: col			
	min_val	max_val	min_val	max_val		
0	1	4	totalRows - row	totalRows + row	*	
1	1	3	totalRows - row	totalRows + row	***	
2	1	2	totalRows - row	totalRows + row	*****	
3	1	1	totalRows - row	totalRows + row	*****	
4	0	0	totalRows - row	totalRows + row	*****	

Row level Repetitive Action					Expected Output	
Variable: row	Column level Repetitive Action					
	Variable: spaces		Variable: col			
	min_val	max_val	min_val	max_val		
0	1	4	totalRows - row	totalRows + row	*	
1	1	3			***	
2	1	2			*****	
3	1	1			*****	
4	0	0			*****	

The condition (**col >=(totalRows - row)**) gives the minimum limit. This specific logical expression will calculate a starting point from which it will start printing star("*) character(s) till the maximum value of the **variable: col**, which will be (**totalRows + row**) calculated.

Then **if-condition** can be given for in between the range of values givens as:

(**totalRows - row**) and (**totalRows + row**).

On applying comparison and logical operators if-condition can be given as:

If (**col >=(totalRows - row) and col <=(totalRows + row)**)

We already know that;

variable: row = 0,1,2,3,4 or 0,1,2,3,4, totalRows - 1

variable:col = 1,2,3,4,5 or 1,2,3,4,(totalRows + row)

Why the maximum value of the variable: col is kept as dynamic?

We know that the printing star("*") character(s) will continue till the maximum value of the variable: **col**, which is (**totalRows + row**).

We are not worried about the spaces when printing of all the star("*") character(s) gets completed in a row. It makes sense to restrict the column level looping where the maximum number of printing of star("*") characters reached.

Why the minimum value of the variable: col is kept as 1?

It makes sense to start the **col=1,2...** from the value 1 since we have to Print space characters before the actual task of printing star("*") characters. It is required to count space characters accurately at the beginning of the column level looping.

We can eliminate the logical AND(and) condition(col <=(totalRows + row)) in the **if-condition** to print the star("*") characters. It's already planned **to consider** this logical AND(and) condition as the **loop-condition** already been taken into the column level inner for-loop. So, the if-condition will now reduce to the following:

If (col >=(totalRows - row))

We can summarize it all as;

Prints a character star when the condition is True for every value of the variable: col

col=1	col=2	col=3	col=4	col=(totalRows + row)
-------	-------	-------	-------	-------	-----------------------

If (condition =(col >=(totalRows - row)) is True)

Print star character

Else

Print space character

We generalized the non-repeating numeric pattern of the column **max_val** into an expression.

We can now define column level **for-loop**, which is the same for every row value.

ELEMENTS OF COLUMN LEVEL FOR LOOP			
Variable: col		INCREMENTAL ORDER	ACTION
start	end	step-size	Print character
col = 1	col <= totalRows + row	1	* or space

ROW LEVEL REPETITIVE ACTION

The following statements had been repeated 5 times(total number of rows).

Column level for-loop for printing star characters # inner for-loop

```
print()# Move to the next row
```

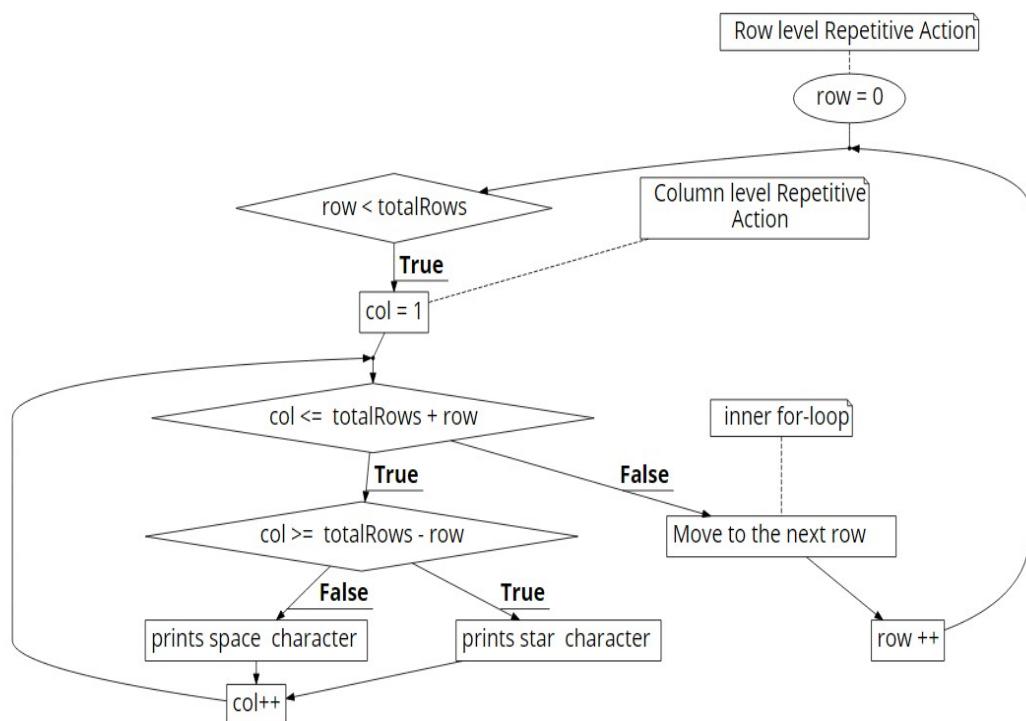
We have assigned a unique value to each row starting from 0 to 4.

Let's consider;

row → variable type of integer to store integer value one at a time and represents row number between 0 and 4(totalRows - 1). The **totalRows** has initially assigned the value 5.

ELEMENTS OF ROW LEVEL FOR LOOP				
Variable: row		INCREMENTAL ORDER	ACTION	
start	end	step-size	Statements	
row = 0	row = totalRows - 1	1	1#	COLUMN LEVEL For Loop for printing space or star(*) characters.
			2#	Move To Next Row

We got everything related to **for-loop** for row level repetition, and a related flow-chart can be shown.



Let's execute the following program.

```
PatternE1.py
totalRows = 5 # total number of rows to display

# Row level Repetitive Action :

# Action1. Executes Column level Repetitive Action:

# Action2. Move control of a program to the next row

#           to set the ground for printing characters column-wise.

for row in range(0,totalRows):

    # Column level Repetitive Action :

    # Action1. Print characters column-wise or in the same row.

    for col in range(1,totalRows + row+1):

        if (col >=(totalRows - row)):

            # Action2.prints star * character

            print("*",end="")

        else:

            # Action3.prints space character

            print(" ",end="")

    # ends inner for-loop

    # move control of a program to the next row

    # in order to switch the printing of characters to the next line.

    print()

# outer for-loop
```

Output

```
*
```

```
***
```

```
*****
```

```
*****
```

```
*****
```

Alternative solution2

From the previous solution, when the **if-condition** is **True**, we will print a star("") character, and in the else part of the **if-statement**, we will Print spaces. Let's start with the observation table that we got from the previous solution.

Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: spaces		Variable: col		
	min_val	max_val	min_val	max_val	
1	1	4	5	5	*
2	1	3	4	6	***
3	1	2	3	7	*****
4	1	1	2	8	*****
5	0	0	1	9	*****

Reversing the counter values of the variable **row**, i.e., **row = 5, 4, 3, 2, 1** so that it will match the column **min_val** of the variable **col**. Moreover, we can use the **variable: totalRows** to generalize the column **max_val** for printing star character(s).

Row level Repetitive Action					
Variable: row	Column level Repetitive Action				
	Variable: spaces		Variable: col (For printing *)		
	min_val	max_val	min_val	max_val	
5	1	4	row	5 = 5 + 0	totalRows + 0
4	1	3	row	6 = 5 + 1	totalRows + 1
3	1	2	row	7 = 5 + 2	totalRows + 2
2	1	1	row	8 = 5 + 3	totalRows + 3
1	0	0	row	9 = 5 + 4	totalRows + 4

Part of **variable: col's column max_val's expression** still has a distinct numeric value at each row.

Row level Repetitive Action					
Variable: row	Column level Repetitive Action				
	Variable: col (For printing *)				
	min_val	max_val			
5	row	5 = 5 + 0		totalRows + 0	totalRows + 5 - 5
4	row	6 = 5 + 1		totalRows + 1	totalRows + 5 - 4
3	row	7 = 5 + 2		totalRows + 2	totalRows + 5 - 3
2	row	8 = 5 + 3		totalRows + 3	totalRows + 5 - 2
1	row	9 = 5 + 4		totalRows + 4	totalRows + 5 - 1

Part of **variable: col's column max_val's expression** still has a distinct numeric value at each row. Again, we can use the **variable: totalRows** to generalize the column **max_val**.

Row level Repetitive Action					
Variable: row	Column level Repetitive Action				
	Variable: col (For printing *)				
	min_val	max_val			
5	row	totalRows + 0	totalRows + 5 - 5	totalRows + totalRows - 5	
4	row	totalRows + 1	totalRows + 5 - 4	totalRows + totalRows - 4	
3	row	totalRows + 2	totalRows + 5 - 3	totalRows + totalRows - 3	
2	row	totalRows + 3	totalRows + 5 - 2	totalRows + totalRows - 2	
1	row	totalRows + 4	totalRows + 5 - 1	totalRows + totalRows - 1	

Part of variable **col's column max_val's expression matches with the row values.**

Row level Repetitive Action					
Variable: row	Column level Repetitive Action				
	Variable: col (For printing *)				
	min_val	max_val			
5	row	totalRows + totalRows - 5	2 * totalRows - row	2 * totalRows - row	
4	row	totalRows + totalRows - 4	2 * totalRows - row		
3	row	totalRows + totalRows - 3	2 * totalRows - row		
2	row	totalRows + totalRows - 2	2 * totalRows - row		
1	row	totalRows + totalRows - 1	2 * totalRows - row		

Row level Repetitive Action					
Variable: row	Column level Repetitive Action				
	Variable: col (For printing *)				
	min_val	max_val			
5					
4					
3		row			
2					
1					

The condition ($col \geq \text{row}$) gives starting point from which **for-loop** will start printing star("") character(s) till the maximum value of variable col which is ($2 * \text{totalRows} - \text{row}$) for every value of row. The if-condition can be given for between the range of values (row) and ($2 * \text{totalRows} - \text{row}$) using the comparison and logical operators as:

If ($\text{col} \geq (\text{row})$ and $\text{col} \leq (2 * \text{totalRows} - \text{row})$)

We already know that **Same expression**

variable col = 1,2,3,4,5 or 1,2,3,4,...,($2 * \text{totalRows} - \text{row}$)

variable row = 5,4,3,2,1 or **totalRows,4,3,2,1**

We can eliminate the logical AND(and) condition(`col <=(2 * totalRows - row)`) in the **if-condition**. This logical AND(and) condition already been taken care as the **loop-condition** in the column level inner for-loop.

So, the if-condition will now reduce to `If (col >= row)`. We can summarize it all as;

Prints a star when the condition is True for every value of variable col

<code>col=1</code>	<code>col=2</code>	<code>col=3</code>	<code>col=4</code>	<code>col=(2 * totalRows - row)</code>
--------------------	--------------------	--------------------	--------------------	-------	--

`If (condition=(col >= row) is True)`

Print star character

Else

Print space character

We generalized the non-repeating numeric pattern of the column **max_val** into an expression.

We can now define column level **for-loop**.

Elements of Column level for-loop

<code>min_val</code>	<code>1</code>
<code>max_val</code>	<code>(2 * totalRows - row)</code>
<code>variable_name</code>	<code>col</code>
<code>statement(s)</code>	<code>print("*",end="")</code>

ROW LEVEL REPETITIVE ACTION

The following statements had been repeated 5 times(total number of rows).

Column level for-loop for printing star characters # inner for-loop

`print()`# Moves to the next row

We have assigned a unique value to each row starting from 5 to 1.

Let's consider;

`row` → variable type of integer to store integer value one at a time and represents row number between 5 and 1. The **totalRows** has initially assigned the value 5.

We got everything related to **for-loop** for row level repetition.

Elements Of Row Level decrementing for-loop

<code>min_val</code>	<code>1</code>
<code>max_val</code>	<code>totalRows</code>
<code>variable_name</code>	<code>row</code>
<code>statement(s)</code>	<p>Column level for-loop for printing <u>star characters</u></p> <p><code>print()</code># Moves to the next row</p>

Let's execute the following program.

```
PatternE2.py
totalRows = 5 # total number of rows to display

# Row level Repetitive Action :

# Action1.Executes Column level Repetitive Action:

# Action2.Move control of a program to the next row

#           to set the ground for printing characters column-wise.

for row in range(totalRows, 0,-1):

    # Column level Repetitive Action :

    # Action1. Print characters column-wise or in the same row

    for col in range(1,(2 * totalRows - row)+ 1):

        if (col >= row):

            # Action2.prints star * character

            print("*",end="")

        else:

            # Action3.prints space character

            print(" ",end="")

    # Ends inner for-loop

    # move control of a program to the next row

    # in order to switch the printing of characters to the next line.

    print()

    # Ends outer for-loop
```

Output

```
*
```

```
***
```

```
*****
```

```
*****
```

```
*****
```

Pattern 1F

Our programming job is to output this pattern on one part of the computer screen, as shown in the following. We can consider the computer screen as a combination of (**row**, **column**) numeric value paired boxes that are spread all over the screen. All box treated as a resident of just a single character only.



Data Representation

Although logically, we can exactly locate the position of each printable character on the display screen by knowing it's uniquely identifiable (**row**, **column**) numeric paired-values.

For example :

(**2, 4**) means character “*” at 2nd row and the 4th column in the pattern.

Row, Col	Col = 1	Col = 2	Col = 3	Col = 4	Col = 5	Col = 6	Col = 7	Col = 8	Col = 9
Row = 1	*(1, 1)	*(1, 2)	*(1, 3)	*(1, 4)	*(1, 5)	*(1, 6)	*(1, 7)	*(1, 8)	*(1, 9)
Row = 2		*(2, 2)	*(2, 3)	*(2, 4)	*(2, 5)	*(2, 6)	*(2, 7)	*(2, 8)	
Row = 3			*(3, 3)	*(3, 4)	*(3, 5)	*(3, 6)	*(3, 7)		
Row = 4				*(4, 4)	*(4, 5)	*(4, 6)			
Row = 5					*(5, 5)				

The actual challenge is to find correct (row, col) positions on the computer screen by writing a program. This will be carried out by applying for-loop along with appropriate logical conditions. This stimulates growth for algorithmic thinking, a fundamental requirement of a confident programmer.

Exercise. Complete the steps for the generation of this logic pattern for every row 1 to 5 by showing output on the computer screen diagram.

Steps for pattern generation(row = 1 to 5)	Computer Screen																																																																													
For row = 1	<table border="1"> <tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td></td></tr> <tr><td></td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>3</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>4</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>5</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>		1	2	3	4	5	6	7	8	9			1											2											3											4											5																				
	1	2	3	4	5	6	7	8	9																																																																					
	1																																																																													
	2																																																																													
	3																																																																													
	4																																																																													
	5																																																																													
For row = 2	<table border="1"> <tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td></td></tr> <tr><td></td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>3</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>4</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>5</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>		1	2	3	4	5	6	7	8	9			1											2											3											4											5																				
	1	2	3	4	5	6	7	8	9																																																																					
	1																																																																													
	2																																																																													
	3																																																																													
	4																																																																													
	5																																																																													
For row = 3	<table border="1"> <tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td></td></tr> <tr><td></td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>3</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>4</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>5</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>		1	2	3	4	5	6	7	8	9			1											2											3											4											5																				
	1	2	3	4	5	6	7	8	9																																																																					
	1																																																																													
	2																																																																													
	3																																																																													
	4																																																																													
	5																																																																													
For row = 4	<table border="1"> <tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td></td></tr> <tr><td></td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>3</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>4</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>5</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>		1	2	3	4	5	6	7	8	9			1											2											3											4											5																				
	1	2	3	4	5	6	7	8	9																																																																					
	1																																																																													
	2																																																																													
	3																																																																													
	4																																																																													
	5																																																																													
For row = 5	<table border="1"> <tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td></td></tr> <tr><td></td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>3</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>4</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>5</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>6</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>		1	2	3	4	5	6	7	8	9			1											2											3											4											5											6									
	1	2	3	4	5	6	7	8	9																																																																					
	1																																																																													
	2																																																																													
	3																																																																													
	4																																																																													
	5																																																																													
	6																																																																													

Let's convert steps into a program using inbuilt methods.

Move to the <u>next row</u> .	<code>print()</code>
Move to the next (row, col) position and print space " " character in the <u>same row</u>	<code>print(" ",end="")</code>
Move to the next (row, col) position and print star "*" character in the <u>same row</u>	<code>print("*",end="")</code>

The task of printing characters to generate this particular pattern is repetitive in nature. To output, these characters on the computer screen repeatedly mentioned method calls bound to get repeat many a time within **for-loop** in a program.

COLUMN LEVEL REPETITIVE ACTION

Let's consider;

row → variable type of integer to store integer value one at a time and represents row number between 0 and 4.

col → variable type of integer to store integer value one at a time and represents column number between 1 and 5.

spaces → variable type of integer to store integer value one at a time and represents space character at column number between 1 and 5.

totalRows → variable type of integer to store integer value for the total number of rows.

Logic Trace Tables

(row, col) value pair depicts the position of "*" character on the display screen									
	row=1, col1	row=1, col2	row=1, col3	row=1, col4	row=1, col5	row=1, col6	row=1, col7	row=1, col8	row=1, col9
space		row=2, col2	row=2, col3	row=2, col4	row=2, col5	row=2, col6	row=2, col7	row=2, col8	
space	space		row=3, col3	row=3, col4	row=3, col5	row=3, col6	row=3, col7		
space	space	space		row=4, col4	row=4, col5	row=4, col6			
space	space	space	space		row=5, col5				

Space ("") and star("*) characters repeated many times.

So, we can use **for-loop** for space("") and star(*) characters.

Row level Repetitive Action			Expected Output
Variable: row	Column level Repetitive Action	Variable: col	
	Variable : spaces		
1	0(no space)	1, 2, 3, 4, 5, 6, 7, 8, 9	*****
2	1	2, 3, 4, 5, 6, 7, 8	*****
3	1,2	3, 4, 5, 6, 7	***
4	1,2,3	4, 5, 6	*
5	1,2,3,4	5	

We can fill the previous table as also follows in order to explore various components of the **for-loop**.

Row level Repetitive Action				Expected Output
Variable: row	Column level Repetitive Action	Variable: col		
	Variable: spaces	min_val	max_val	
1		0	0	*****
2		1	1	*****
3		1	2	***
4		1	3	*
5		1	4	

We know that **totalRows = 5**(**total number of rows**).

Let's try to use variable **totalRows** to generalize the variable **col**'s columns are **min_val & max_val**.

Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: spaces	Variable: col			
	min_val	max_val	min_val	max_val	
1	0	0	totalRows -4	totalRows + 4	*****
2	1	1	totalRows -3	totalRows + 3	*****
3	1	2	totalRows -2	totalRows + 2	****
4	1	3	totalRows -1	totalRows + 1	***
5	1	4	totalRows -0	totalRows + 0	*

Let use decrementing for-loop with range 4 to at the row level repetition.

This is in order to match the part of the expression of variable **col**'s columns **min_val** & **max_val**. Using variable **row** to generalize the variable **col**'s columns **min_val** & **max_val**.

Row level Repetitive Action					
Variable: row	Column level Repetitive Action				
	Variable: spaces	Variable: col			
	min_val	max_val	min_val		max_val
4	0	0	totalRows - row		totalRows + row
3	1	1	totalRows - row		totalRows + row
2	1	2	totalRows - row	totalRows - row	totalRows + row
1	1	3	totalRows - row		totalRows + row
0	1	4	totalRows - row		totalRows + row

The condition ($col \geq (\text{totalRows} - \text{row})$) gives the minimum limit, a starting point from which it will start printing star("*) character(s) till the maximum value of variable col, which is (**totalRows + row**).

The **if-condition** can be given for between the range of values

(**totalRows - row**) and (**totalRows + row**) using the comparison and logical operators as:

If ($col \geq (\text{totalRows} - \text{row})$ and $col \leq (\text{totalRows} + \text{row})$)

We already know that **Same expression**

Variable col = 1,2,3,4,...,9 or 1,2,3,4,...,(**totalRows + row**)

Variable **row** = 4,3,2,1,0 or (**totalRows - 1**),3,2,1,0

We can eliminate the logical AND(and) condition($col \leq (\text{totalRows} + \text{row})$) in the

if-condition to print the star("*) characters. This logical AND(and) condition already been considered as the upper_limit in the **loop-condition** for column level inner for-loop. So, the **if-condition** will now reduce to the following.

If ($col \geq (\text{totalRows} - \text{row})$), it will allow printing star("*) character when the result of **if-condition's expression** is **True** or else will print the space character. We generalized the non-repeating numeric pattern of the column **max_val** into an expression. We can now define column level **for-loop**.

Elements of Column level *for-loop*

min_val	1
max_val	totalRows + row
variable_name	col
statement	print("*",end="")

ROW LEVEL REPETITIVE ACTION

The following statements had been repeated 5 times(total number of rows).

Column level *for-loop* for printing star characters # **inner for-loop**

print()# Move to the next row

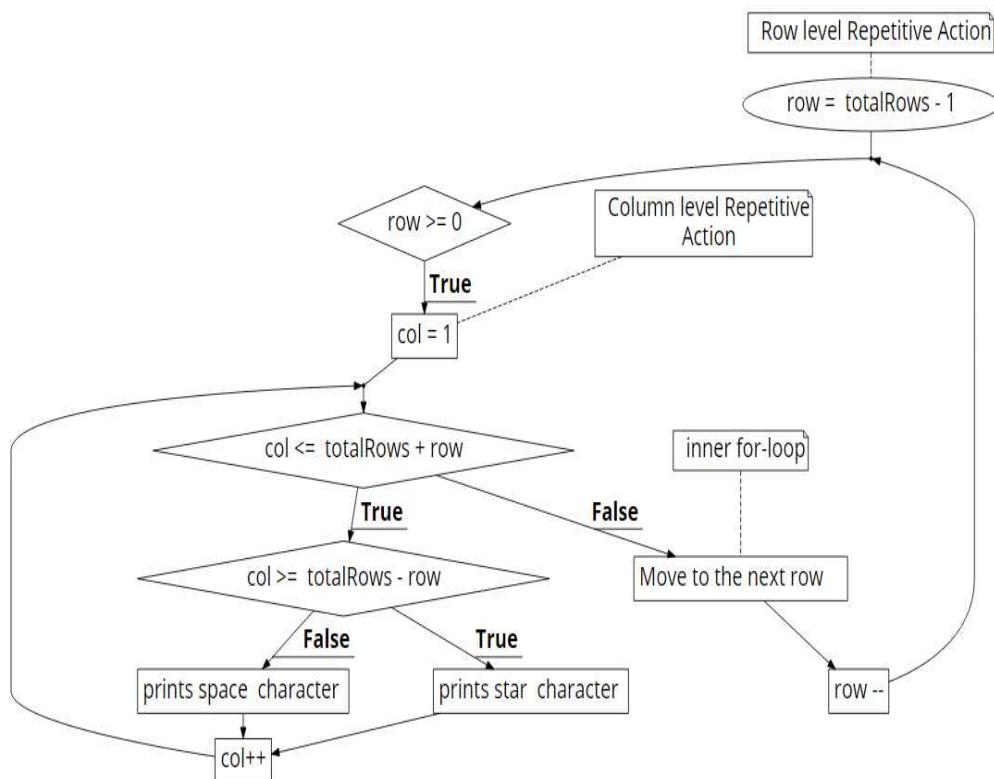
row → variable type of integer to store integer value one at a time and represents row number between 4 or (**totalRows - 1**) and 0. The **totalRows** has initially assigned the value 5.

We got everything related to ***for-loop*** for row level repetition.

Elements Of Row Level decrementing *for-loop*

min_val	0
max_val	totalRows - 1
variable_name	row
statement(s)	Column level <i>for-loop</i> for printing <u>star characters</u> print()# Move to the next row

The flow-chart can be given as follows:



Let's execute the following program.

```
PatternF1.py
totalRows = 5 # total number of rows to display

# Row level Repetitive Action :
# Action1. Executes Column level Repetitive Action:
# Action2. Move control of a program to the next row
#           to set the ground for printing characters column-wise.
for row in range(totalRows, 0,-1):

    # Column level Repetitive Action :
    # Action1. Print characters column-wise or in the same row.
    for col in range(1,totalRows + row):

        if (col >=(totalRows - row+1)):
            # Action2.prints star * character
            print("*",end="")
        else:
            # Action3.prints space character
            print(" ",end="")

    # Ends inner for-loop

    # move control of a program to the next row
    # in order to switch the printing of characters to the next line.
    print()

# Ends outer for-loop
```

Output

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

Alternative solution2

By using incrementing for-loop at the row level repetition, we can give alternative solutions.

Problem F1. Find the different logic and write the program to print the following pattern for 5 rows. Test the logic for 12 rows. Some details are mentioned in the form of a table to start with.

Row level Repetitive Action					Expected Output
Variable: row Column level Repetitive Action					
	Variable: spaces		Variable: col		
	min_val	max_val	min_val	max_val	
1	0	0	1	9	*****
2	1	1	2	8	*****
3	1	2	3	7	***
4	1	3	4	6	**
5	1	4	5	5	*

When the **if-condition** is True program should print a star("*) character, and in the **else** part of the **if-statement** program should Print space character.

Problem F2. Continuation to the previous problem (**Problem F1**) to write the program for the following situation.

When the **if-condition** is True, then the program should print a space("") character, and in the **else** part of the **if-statement** program should print star("*)character.

Pattern 1G

Our programming job is to output this pattern on one part of the computer screen, as shown in the following. We can treat the computer screen as a combination of **(row, column)** numeric value paired boxes that are spread all over the screen. Each box can hold a single character only.



Data Representation

The complete pattern can be easily expressed as a collection of selected **(row, column)** numeric value pairs. The first value represents the row number, and the second value represents a column number.

For example;

(2, 1) means character “*” at **second row** and **first column** in the table.

The real challenge is to find the correct (row, col) positions and to print the required character on the computer screen by writing a program. This will be carried out by applying **for-loop** in combination with appropriate logical conditions. Finding logical conditions in a step-by-step approach stimulates algorithmic thinking, a fundamental requirement to become a confident programmer.

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	*(1,1)				
Row=2	*(2,1)	*(2,2)			
Row=3	*(3,1)	*(3,2)	*(3,3)		
Row=4	*(4,1)	*(4,2)	*(4,3)	*(4,4)	
Row=5	*(5,1)	*(5,2)	*(5,3)	*(5,4)	*(5,5)
Row=6	*(6,1)	*(6,2)	*(6,3)	*(6,4)	
Row=7	*(7,1)	*(7,2)	*(7,3)		
Row=8	*(8,1)	*(8,2)			
Row=9	*(9,1)				

Exercise. Print logic pattern step-by-step from row 1 to 9 by showing output similar to in the computer screen.

Steps for pattern generation(row = 1 to 9) with computer screen diagram.

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

For row = 1

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

For row = 2

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

For row = 3

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

For row = 4

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

For row = 5

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

For row = 6

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

For row = 7

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

For row = 8

	1	2	3	4	5		
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							

For row = 9

	1	2	3	4	5		
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							

For row = 10

Let's convert steps into a program using in-built methods.

Move the cursor to the next row

`print()`

Move the cursor to the next (row, column) position and then print star "*" character in the same row.

`print("*", end="")`

In order to Print star "*" character, each movement on the computer screen will be a single keyboard character-sized jump from the left to the right direction in the same row. Since looping is meant for repetitive action within a program, we need to use these printing statements and the **for-loop**. Writing a program to calculate the row and column values to print characters correctly into the computer screen will be the actual drill for improving logical thinking capabilities.

Let's consider the following variables, which are a type of integer to store integer value one at a time.

row → represents row numbers between 1 and 9.

col → represents column numbers between 1 and 5.

totalRows → represents the input value in the program for the total number of rows.

COLUMN LEVEL REPETITIVE ACTION

We need to explore various components of required **outer and inner for-loops**.

(row, col) value pair depicts the position of “**” character on the display screen

row=1, col=1				
row=2, col=1	row=2, col=2			
row=3, col=1	row=3, col=2	row=3, col=3		
row=4, col=1	row=4, col=2	row=4, col=3	row=4, col=4	
row=5, col=1	row=5, col=2	row=5, col=3	row=5, col=4	row=5, col=5
row=6, col=1	row=6, col=2	row=6, col=3	row=6, col=4	
row=7, col=1	row=7, col=2	row=7, col=3		
row=8, col=1	row=8, col=2			
row=9, col=1				

We required a set of **min_val & max_val** boundary values so that looping shall be performed.

Row level Repetitive Action	Column level Repetitive Action	Expected Output
Variable: row	Variable: col	
1	1	*
2	1,2	**
3	1,2,3	***
4	1,2,3,4	****
5	1,2,3,4,5	*****
6	1,2,3,4	****
7	1,2,3	***
8	1,2	**
9	1	*

Row level Repetitive Action	Column level Repetitive Action		Expected Output
Variable: row	Variable: col		
	min_val	max_val	
1	1	1	*
2	1	2	**
3	1	3	***
4	1	4	****
5	1	5	*****
6	1	4	****
7	1	3	***
8	1	2	**
9	1	1	*

Approach to solution

The variables are row & totalRows = 5. We need to generalize the non-repeating numeric pattern of the column **max_val** using these remaining variables. The numeric pattern (1 to 5) of the column **max_val** matches exactly every value of the row. Hence, it can be expressed in the row variable.

Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: col		min_val	max_val	
1	1	1	1	row	*
2	2	1	2	row	**
3	3	1	3	row	***
4	4	1	4	row	****
5	totalRows	1	5	row	*****
6	6	1	4	totalRows - 1	****
7	7	1	3	totalRows - 2	***
8	8	1	2	totalRows - 3	**
9	totalRows + 4	1	1	totalRows - 4	*

The column **min_val** contains only the data **col = 1**, which means it is the same for every row. Hence, column **min_val** is already being in a generalized state. Hence, we got the consistent numerical expression in order to **start** the column level for-loop. The column **max_val** is partially generalized (row = 1 to 5).

Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: col		min_val	max_val	
1	1		1		*
2	2		2		**
3	3		3	row	***
4	4	1	4		****
5	totalRows		5		*****
6	6		4	totalRows - 1	****
7	7		3	totalRows - 2	***
8	8		2	totalRows - 3	**
9	totalRows + 4		1	totalRows - 4	*

We need to find the generalized expression for the maximum value of the **variable: row** also.

There are still non-repeating numeric values in the column **max_val** of the col variable.

Row level Repetitive Action		Column level Repetitive Action		
Variable: row	Variable: col	min_val	max_val	
1	1			
2	2			
3	3	row	row	row
4	4			
5	totalRows	1		
6	6	totalRows - 1	totalRows - (6 - 5)	totalRows - (row - 5)
7	7	totalRows - 2	totalRows - (7 - 5)	totalRows - (row - 5)
8	8	totalRows - 3	totalRows - (8 - 5)	totalRows - (row - 5)
9	totalRows + 4	totalRows - 4	totalRows - (9 - 5)	totalRows - (row - 5)

Row level Repetitive Action		Column level Repetitive Action		
Variable: row	Variable: col	min_val	max_val	
1	1			
2	2			
3	3			
4	4	1	row	row
5	totalRows			
6	6		totalRows - (row - 5)	totalRows - row + 5
7	7		totalRows - (row - 5)	totalRows - row + 5
8	8		totalRows - (row - 5)	totalRows - row + 5
9	totalRows + totalRows - 1		totalRows - (row - 5)	totalRows - row + 5

There are still non-repeating numeric values in the column **max_val** of the col variable.

Row level Repetitive Action		Column level Repetitive Action	
Variable: row	Variable: col	min_val	max_val
1	1		
2	2		
3	3		row
4	4		row
5	5	1	
6	6	totalRows – row + totalRows	2*totalRows – row
7	7	totalRows – row + totalRows	2*totalRows – row
8	8	totalRows – row + totalRows	2*totalRows – row
9	2 * totalRows – 1	totalRows – row + totalRows	2*totalRows - row

Row level Repetitive Action		Column level Repetitive Action	
Variable: row	Variable: col	min_val	max_val

1	1		
2	2		
3	3	row	row
4	4		
5	totalRows	1	
6	6	2*totalRows - row	
7	7	2*totalRows - row	
8	8	2*totalRows - row	2*totalRows - row
9	2 * totalRows - 1	2*totalRows - row	

The numeric pattern of the column **max_val** of variable **col** matches exactly the values(1 to 5) of the **row**. This specific part of the numeric pattern (1 to 5) can be expressed in the **row** variable.

The condition (**col >=1**) gives a starting point from which it will start printing star("*") character(s) till the value of variable **row** reaches (**totalRows**).

The variable **col** then starts decreases captured in the expression (**2*totalRows - row**).

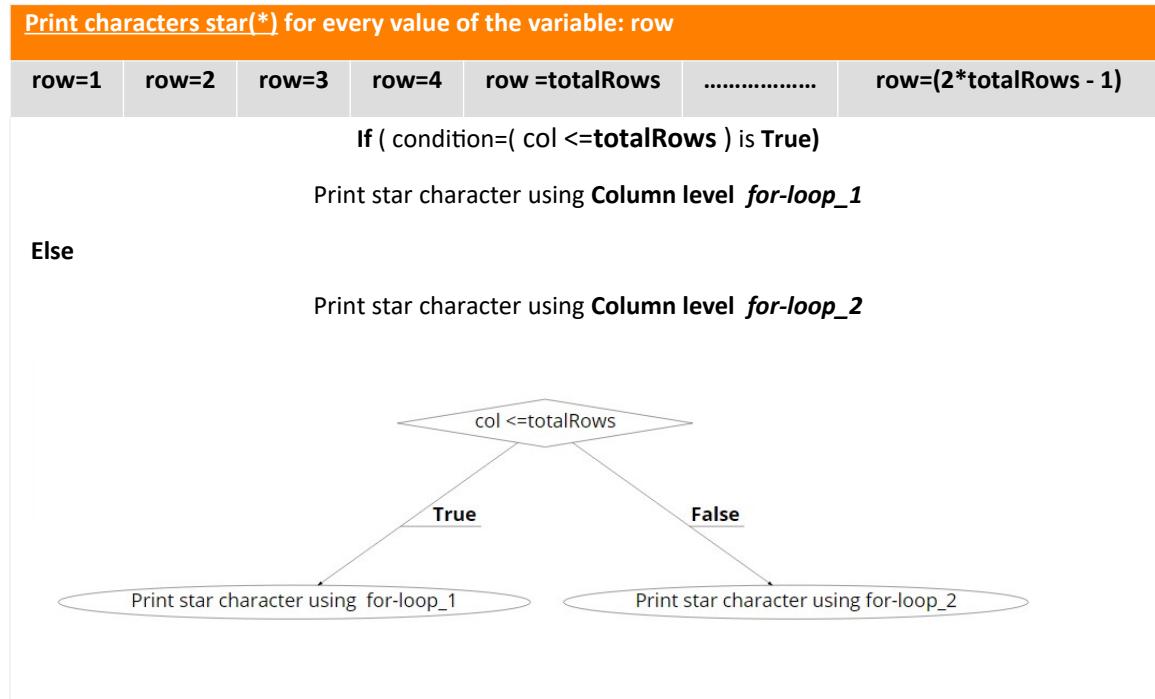
So, in this case, there is not one but two generalized expressions for values of the variable **col**. This indicates that we need to use two separate **for-loop** for the variable **col**.

Let the following be the two versions of column level for-loop for printing star ("*") character(s).

Column level *for-loop_1* for printing star characters # inner for-loop

Column level *for-loop_2* for printing star characters # inner for-loop

We can summarize it as:



We can now define these two column levels ***for-loop***.

Elements of Column level *for-loop_1*

min_val	1
max_val	totalRows
variable_name	col
statement(s)	<code>print("*", end = "")</code>

Elements of Column level *for-loop_2*

min_val	totalRows + 1
max_val	2*totalRows - row
variable_name	col
statement(s)	<code>print("*", end = "")</code>

ROW LEVEL REPETITIVE ACTION

We have assigned unique values to each row starting from 1 to 9.

We can represent value 9 as $(2*5 - 1)$. The variable **totalRows** has initially assigned the value 5.

So, we can generalize the maximum limit of the **row** variable using the variable. Let's consider;

row → variable type of integer to store integer value one at a time and represents row number between 1 and $(2*\text{totalRows} - 1)$.

We have logically divided row level repetition by for-loop into two parts due to two different versions of the column level for-loop.

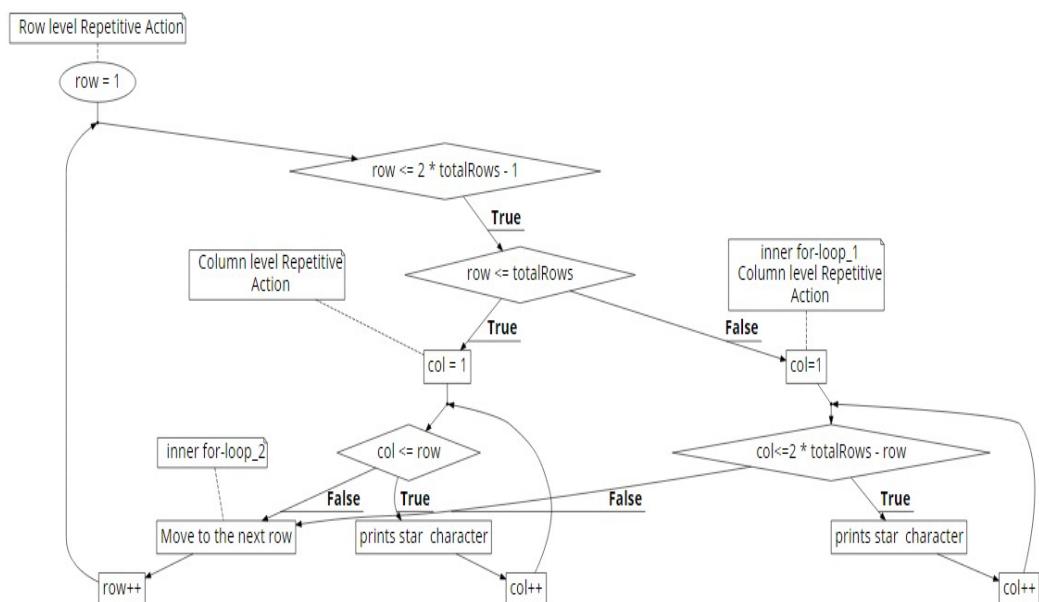
Elements Of Row Level decrementing for-loop

min_val	1
totalRows	totalRows
variable_name	row
statement(s)	Column level for-loop_1 for printing star characters <code>print()# Move to the next row</code>

Elements Of Row Level decrementing for-loop

min_val	totalRows + 1
max_val	2*totalRows - 1
variable_name	row
statement(s)	Column level for-loop_2 for printing star characters <code>print()# Move to the next row</code>

Since we got all the details related to considered for-loops, we can summarize this related information with the help of a flowchart.



Let's execute the following program.

PatternG1.py

```
totalRows = 5 # number of rows to display

# Row level Repetitive Action :
# Action1. Executes Column level Repetitive Action:
# Action2.Move control of a program to the next row
# to set the ground for printing characters column-wise.
for row in range(1,2 * totalRows):

    # Column level Repetitive Action :
    # Print characters column-wise or in the same row
    if (row <= totalRows):
        # inner for-loop_1:prints star * character
        for col in range(1, row + 1):
            print("*",end ="")

        else:
            # inner for-loop_2:prints space character
            for col in range(1,(2 * totalRows - row)+ 1):
                print("*",end ="")

    # move control of a program to the next row
    # in order to switch the printing of characters to the next line.
    print()
```

Output

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

```
****
```

```
***
```

```
**
```

```
*
```

The inner for-loop_1 and inner for-loop_2 differ only in the maximum limit value of the **col** variable.

Therefore, it will be better if we capture the maximum limit of the **col** variable in a different variable.

Then we will be in a position to use only one inner for-loop for the complete column level repetition.

Let's update the previous program using a single inner for-loop.

PatternG2.py

```
totalRows = 5 # number of rows to display
colMax = 0 # defines the maximum limit for variable:col.

# Row level Repetitive Action :
# Action1. Executes Column level Repetitive Action:
# Action2. Move control of a program to the next row so that,
# printing characters should happen in a new-line.
for row in range(1, 2 * totalRows):

    #sets the maximum limit of the inner for-loop
    if (row <= totalRows):
        colMax = row
    else:
        colMax =(2 * totalRows - row)

    # Column level Repetitive Action :
    # Print characters column-wise or in the same row
    for col in range(1, colMax + 1):
        print("*", end="")

    # Ends inner for-loop

    # move control of a program to the next row
    # in order to switch the printing of characters to the next line.
    print()
    # Ends outer for-loop
```

Output

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

```
****
```

```
***
```

```
**
```

```
*
```

Alternative solution2

Let's start with the trace table that we got from the previous solution.

Row level Repetitive Action			Expected Output
Variable: row	Column level Repetitive Action		
	Variable: col		
	min_val		max_val
1	1		1
2	1		2
3	1		3
4	1		4
5	1		5
6	1		4
7	1		3
8	1		2
9	1		1

The remaining variables are `row` & `totalRows = 5`. We can try to express column `max_val` of `col` variable in terms of a numeric pattern of row values.

Row level Repetitive Action			Expected Output		
Variable: row	Column level Repetitive Action				
	Variable: col				
	min_val max_val		I		
1	1	1	row	row	row - 2*0 *
2	2	2	row	row	row - 2*0 **
3	3	3	row	row	row - 2*0 ***
4	4	4	row	row	row - 2*0 ****
5	1	5	row	row	row - 2*0 *****
6	4	6 - 2	row - 2	row - 2*1	row - 2*1 ****
7	3	7 - 4	row - 4	row - 2*2	row - 2*2 ***
8	2	8 - 6	row - 6	row - 2*3	row - 2*3 **
2*5 - 1	1	9 - 8	row - 8	row - 2*4	row - 2*4 *

We can express column `max_val` of `col` variable in terms of row variable, and part of the expression follows a multiple of 2 but for row values greater than 5 or `totalRows`.

It has the condition when to increment the multiple of 2 or not. The condition is handled by conditional statements, mostly by **if-statement** or **if-else-statement** or **if-elif-else**. This we need to handle using such conditional statement by declaring another integer variable, for example;

`count = 0 # counter to increment the multiple of 2.`

We can give an **if-statement** something like the following by incrementing the value of the **variable: count**.

```
# increment count only when row value is greater
# than the total number of rows.
if (row > totalRows):
# increment count by 1.
    count = count + 1
```

Now we can use variable count to generalize the column **max_val** of **col** variable.

Row level Repetitive Action			Expected Output	
Variable: row	Column level Repetitive Action			
	Variable: col			
		min_val max_val		
1	1	row - 2*0	row - 2*count	*
2	2	row - 2*0	row - 2*count	**
3	3	row - 2*0	row - 2*count	***
4	4	row - 2*0	row - 2*count	****
5	1	5	row - 2*0 row - 2 * count	*****
6		4	row - 2*1	****
7		3	row - 2*2	***
8		2	row - 2*3	**
(2* totalRows - 1)		1	row - 2*4	*

Let's implement these observations as a different program.

PatternG3.py

```
totalRows = 5 # number of rows to display
count = 0 # counter to increment the multiple of 2.

# Row level Repetitive Action :
# Action1. Executes Column level Repetitive Action:
# Action2. The control of the program should move to the next row
# so that the printing of characters is on a new line next time.
for row in range(1, 2 * totalRows):

    # increment count only when row value is greater
    # than the total number of rows.
    if (row > totalRows):
        count = count + 1

    # Column level Repetitive Action :
    # Print characters column-wise or in the same row
    for col in range(1, row - 2 * count + 1):
        print("*", end="")

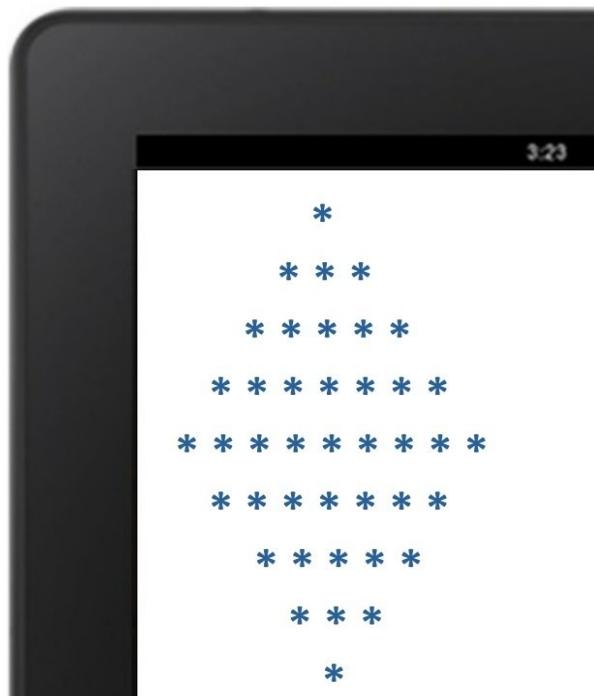
    # Ends inner for-loop

    # move control of a program to the next row
    # in order to switch the printing of characters to the next line.
    print()
# Ends outer for-loop
```

Verify results for a different number of lines. Let it be **totalRows** = 7,8 or 11.

Pattern1H

Our programming job is to output this pattern on one part of the computer screen, as shown in the following. We can treat the computer screen as a combination of **(row, column)** numeric value paired boxes that are spread all over the screen. Each box can hold a single character only.



Data Representation

The complete pattern can be easily expressed as a collection of selected (row, column) numeric value pairs. Those selected (row, column) numeric value pairs are shown in the given table structure. For example,(2, 5) means character “*” at the **second row** and **fifth column** in the table.

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9									
Row=1					*	(1,5)												
Row=2				*	(2,4)	(2 ,5)	*	(2,6)										
Row=3			*	(3,3)	*	(3,4)	*	(3,5)	*	(3,6)	*	(3,7)						
Row=4		*	(4,2)	(4,3)	*	(4,4)	(4,5)	*	(4,6)	(4,7)	*	(4,8)						
Row=5	*	(5,1)	*	(5,2)	*	(5,3)	*	(5,4)	*	(5,5)	*	(5,6)	*	(5,7)	*	(5,8)	*	(5,9)
Row=6		*	(6,2)	(6,3)	*	(6,4)	(6,5)	*	(6,6)	(6,7)	*	(6,8)						
Row=7			*	(7,3)	*	(7,4)	*	(7,5)	*	(7,6)	*	(7,7)						
Row=8				*	(8,4)	(8,5)	*	(8,6)										
Row=9					*	(9,5)												

Exercise. Complete the steps for the generation of this logic pattern for every row 1 to 9 by assuming output on the computer screen diagram.

Steps for pattern generation(row = 1 to 9) with computer screen diagram.

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

For row = 1

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

For row = 2

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

For row = 3

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

For row = 4

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

For row = 5

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

For row = 6

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

For row = 7

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

For row = 8

	1	2	3	4	5		
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							

For row = 9

	1	2	3	4	5		
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							

For row = 10



Let's convert steps into a program using **inbuilt methods**.

Given three types of programming statements are going to be repeated many times in the program. We know that looping is meant for repetitive action within a program. Hence, we need to use these printing statements within the for-loop logically.

Purpose	Method
To move the cursor to the <u>next row</u> .	<code>print()</code>
Move to the next (row, column) position and then print space " " character in the <u>same row</u> .	<code>print(" ", end="")</code>
Move to the next (row, column) position and then print star "*" character in the <u>same row</u> .	<code>print("*", end="")</code>

Writing a program to calculate the row and column values appropriately is the actual challenge in such types of simple picture-based exercises. Such exercises force the brain cells to work harder without losing motivation. We need to practice such exercise till our [visualization about the logic of the program](#) gets sufficiently improved. Without using logic trace tables, learners have achieved the mark most of the time if learners can run such a program in a single chance.

COLUMN LEVEL REPETITIVE ACTION

Let's consider;

row → variable type of integer to store integer value one at a time and represents row number between 1 and 9.

col → variable type of integer to store integer value one at a time and represents column number between 1 and 9.

spaces → variable type of integer to store integer value one at a time and represents space character at column number between 1 and **totalRows**.

totalRows → variable type of integer to store integer value for the total number of rows which is 5 by default.

Logic Trace Tables

(row, col) value pair depicts the position of "*" character on the display screen				
space	space	space	space	row=1, col=5

space	space	space	row=2, col=4	row=2, col=5	row=2, col=6			
space	space		row=3, col=3	row=3, col=4	row=3, col=5	row=3, col=6	row=3, col=7	
space		row=4, col=2	row=4, col=3	row=4, col=4	row=4, col=5	row=4, col=6	row=4, col=7	row=4, col=8
row=5, col=1	row=5, col=2	row=5, col=3	row=5, col=4	row=5, col=5	row=5, col=6	row=5, col=7	row=5, col=8	row=5, col=9
	row=6, col=2	row=6, col=3	row=6, col=4	row=6, col=5	row=6, col=6	row=6, col=7	row=6, col=8	
		row=7, col=3	row=7, col=4	row=7, col=5	row=7, col=6	row=7, col=7		
			row=8, col=4	row=8, col=5	row=8, col=6			
					row=9, col=5			

Approach to solution1

The overall observation is that at each row, either it's printing space characters or star (*) characters in a continuous fashion. The columns in the pattern are divided into two parts, either it's printing space (" ") character(s) or star ("*") characters. These commas separated values in the table represent the column positions where space and star character(s) are actually positioned.

Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: spaces	Variable: col	Character count	Total Characters	
1	1,2,3,4	5	4 Spaces + 1 star	5	*
2	1,2,3	4, 5,6	3 Spaces + 3 stars	6	***
3	1,2	3, 4, 5,6,7	2 Spaces + 5 stars	7	*****
4	1	2, 3, 4, 5,6,7,8,	1 Space + 7 stars	8	*****
5	0	1, 2, 3, 4, 5,6,7,8,9	0 Space + 9 stars	9	*****
6	1	2, 3, 4, 5,6,7,8,	1 Space + 7 stars	8	*****
7	1,2	3, 4, 5,6,7	2 Spaces + 5 stars	7	*****
8	1,2,3	4, 5,6	3 Spaces + 3 stars	6	***
9	1,2,3,4	5	4 Spaces + 1 star	5	*

The printing of space(s) comes first before the print star("*) character(s). If we take care of the condition till it prints space character(s), then the remaining are the star(*) character(s). If we closely observe, it's cutting the values of variable col into two halves. One part goes to the printing of space character(s), and the other goes to the printing of star character(s). We know that the comma-separated values basically represent the column positions that uniquely identify the position of space and star character(s).

Let's take out the minimum value (**min_val**) and maximum value(**max_val**) from these comma-separated-values, which were given for printing space and star character(s).

Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: spaces		Variable: col		
	min_val	max_val	min_val	max_val	
1	1	4	5	5	*
2	1	3	4	6	***
3	1	2	3	7	*****
4	1	1	2	8	*****
5	0	0	1	9	*****
6	1	1	2	8	*****
7	1	2	3	7	*****
8	1	3	4	6	***
5 + 4	1	4	5	5	*

We need to derive the generalized expression from the numeric pattern of minimum value (**min_val**) and maximum value(**max_val**). Those generalized expressions will help us to define the **for-loop** at the row and the column level. We know that the total number of rows given by the variable **totalRows** has a value of 5 by default. So **totalRows = 5**.

The variables **row** and **col** both takes the value 1 to 9, i.e., **row =1,2,3,4,5,..,9** and **col=1,2,3,4,5,..,9**.

Let's focus our attention on printing star character(s) and try to determine the expression in terms of the variable **totalRows** = 5.

Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: col (Printing star characters)				
	min_val		max_val		
1	5	5-0	5	5+0	*
2	4	5-1	6	5+1	***
3	3	5-2	7	5+2	*****
4	2	5-3	8	5+3	*****
5	1	5-4	9	5+4	*****
6	2	5-3	8	5+3	*****
7	3	5-2	7	5+2	*****
8	4	5-1	6	5+1	***
5 + 5 -1	5	5-0	5	5+0	*

Part of the **min_val** and **max_val** expression both follows the same numeric pattern. We can consider this as a separate integer variable, and let's name it as count.

Then the logical expression can be expressed in terms of **totalRows** and count.

Row level Repetitive Action					
Variable: row	Variable: count	Column level Repetitive Action			
		Variable: col (Printing star characters)			
		min_val		max_val	
1	0	5	totalRows - count	5	totalRows + count
2	1	4	totalRows - count	6	totalRows + count
3	2	3	totalRows - count	7	totalRows + count
4	3	2	totalRows - count	8	totalRows + count
5	4	1	totalRows - count	9	totalRows + count
6	3	2	totalRows - count	8	totalRows + count
7	2	3	totalRows - count	7	totalRows + count
8	1	4	totalRows - count	6	totalRows + count
2*totalRows - 1	0	5	totalRows - count	5	totalRows + count

The variable count starts to increase by value 1 from the start of the row=1 till the row=**totalRows** (**total number of rows**) and then starts decreases further by value 1 till (**totalRows** – 1) times.

We can deal with this using an if-else conditional statement.

The condition (**col >=(totalRows – count)**) gives starting point from which **for-loop** will start printing star("*") character(s) till the maximum value of variable col, which is (**totalRows + count**) for every value of the row. The **if-condition** can be given using the comparison and logical operators for the range of values between (**totalRows – count**) and (**totalRows + count**) as:

If (**col >=(totalRows – count)** and **col <=(totalRows + count)**)

We already know that **Same expression**

variable:col = 1,2,3,4,5 or 1,2,3,4,...,(totalRows + count)

variable: row = 1,2,3,...,9 or 1,2,3,...,(2 * totalRows - 1)

We can eliminate the logical AND(and) condition(`col <=(totalRows + count)`) in the **if-condition**. This logical AND(and) condition already been taken care in the **loop-condition** of the column level inner **for-loop**. So, the if-condition will now reduce to the following:

`If (col >= (totalRows - count))`

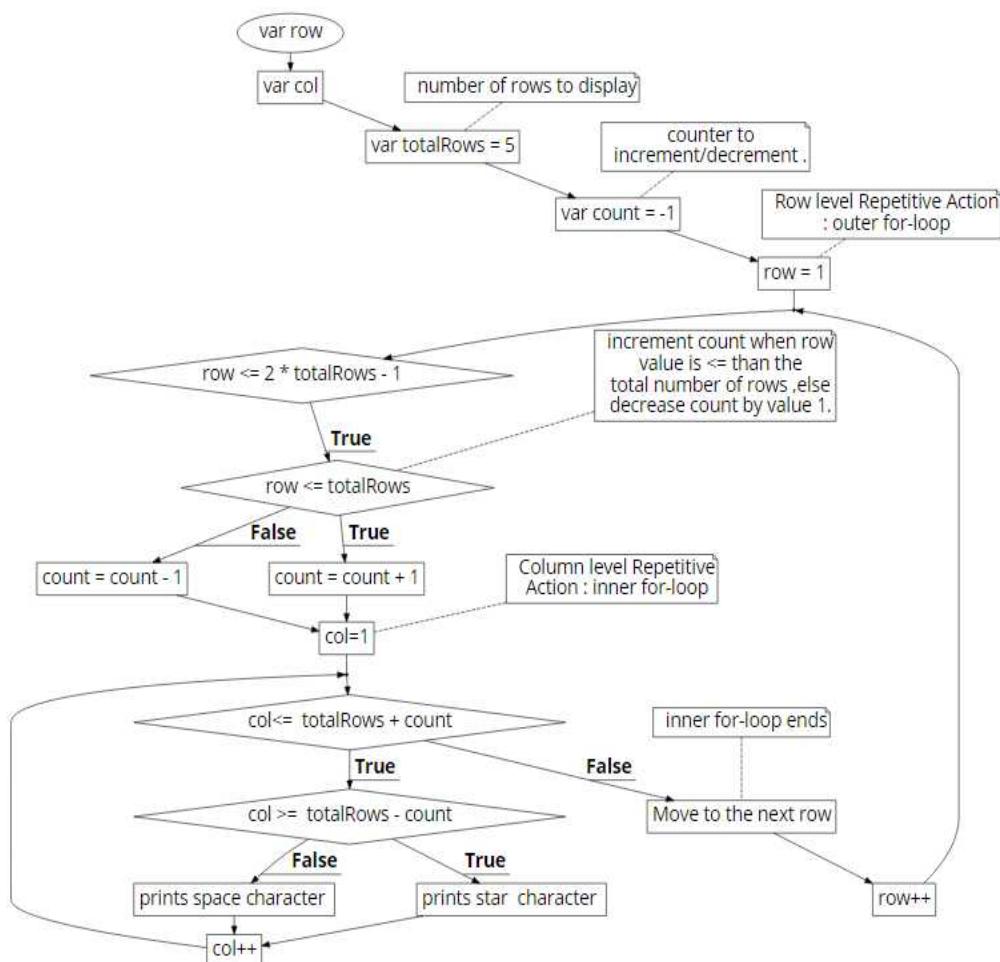
We can generalize the non-repeating numeric pattern of the column **max_val** into an expression. We can define column level **for-loop**.

Elements of Column level for-loop	
min_val	1
max_val	<code>(totalRows + count)</code>
variable_name	col
statement(s)	<code>print("*",end="")</code>

We can define row level **for-loop**.

Elements Of Row Level decrementing for-loop	
min_val	1
max_val	<code>2*totalRows - 1</code>
variable_name	row
statement(s)	Column level for-loop for printing <u>star characters</u> <code>print()# Move to the next row</code>

We can summarize it all in a flowchart as



Let's execute the following program.

PatternH1.py

```
totalRows = 7 # number of rows to display
count = -1 # counter to increment or decrement value.

# Row level Repetitive Action :
# Action1.Executes Column level Repetitive Action:
# Action2.The control of the program should move to the next row
# so that the printing of characters is on a new line next time.
for row in range(1,2 * totalRows):
    # increment count when row value is <= than the
    # total number of rows
    # else decrease count by value 1.
    if (row <= totalRows):
        count = count + 1
    else:
        count = count - 1
    # Column level Repetitive Action :
    # Print characters column-wise or in the same row
    for col in range(1,totalRows + count+1):
        if(col >=(totalRows - count)):
            # prints star * character
            print("*",end="")
        else:
            # prints space character
            print(" ",end="")
    # Ends inner for-loop

    # move control of a program to the next row
    # in order to switch the printing of characters to the next line.
    print()
    # Ends outer for-loop
```

Output

```
*
```

```
***
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*
```

Problem H1. Continuation to the previous program (**PatternH1.py**), modify the program for the following situation.

In the program (**PatternH1.py**), the count variable started with the value -1 given as :

```
count = -1 # counter to increment or decrement
```

Required to match the output (**Pattern1H**) of the modified program (**PatternH1_2.py**) by making the following change:

```
count = 0 # counter to increment or decrement
```

This time program has to initialize the variable count with the value **0** instead of -1, and still, it should give the correct matching output as that of the pattern program (**Pattern1H.py**).

PatternH1_2.py

```
totalRows = 7 # number of rows to display
count = 0 # counter to increment or decrement value.

# Row level Repetitive Action :
# Action1. Executes Column level Repetitive Action:
# Action2. The control of the program should move to the next row
# so that the printing of characters is on a new line next time.

# IMPLEMENT YOUR LOGIC...?
```

Output

```
*
```

```
***
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
***
```

```
*
```

Alternative solution2

Let's start with one of the logic trace tables that we got from the previous solution.

Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: col				
	Printing space character(s)		Printing star character(s)		
	min_val	max_val	min_val	max_val	
1	1	4	5	5	*
2	1	3	4	6	***
3	1	2	3	7	*****
4	1	1	2	8	*****
5	0	0	1	9	*****
6	1	1	2	8	*****
7	1	2	3	7	*****
8	1	3	4	6	***
9	1	4	5	5	*

Let's focus our attention on printing of star character(s) and try to find out the expression in terms of the variables, i.e., row & **totalRows=5**.

Row level Repetitive Action							
Variable: row	Column level Repetitive Action						
	Variable: col (Printing of star (*) characters)						
	min_val				max_val		
	1	5	5-0	totalRows +1 - 1	5	totalRows +0	totalRows +1 - 1
2	4	5-1	totalRows +1 - 2	6	totalRows +1	totalRows +2 - 1	
3	3	5-2	totalRows +1 - 3	7	totalRows +2	totalRows +3 - 1	
4	2	5-3	totalRows +1 - 4	8	totalRows +3	totalRows +4 - 1	
5	1	5-4	totalRows +1 - 5	9	totalRows +4	totalRows +5 - 1	
6	2	2	2	8	totalRows +3	totalRows +5 - 2	
7	3	3	3	7	totalRows +2	totalRows +5 - 3	
8	4	4	4	6	totalRows +1	totalRows +5 - 4	
9	5	5	5	5	totalRows +0	totalRows +5 - 5	

We can use row=1,2,3,4,5 values to express the columns min_val and max_val partially.

Row level Repetitive Action							
Variable: row	Column level Repetitive Action						
	Variable: col (Printing of star (*) characters)						
	min_val				max_val		
	1	5	5-0	totalRows +1 - row	5	totalRows + row - 1	totalRows + row - 1
2	4	5-1	totalRows +1 - row	6	totalRows + row - 1	totalRows + row - 1	
3	3	5-2	totalRows +1 - row	7	totalRows + row - 1	totalRows + row - 1	
4	2	5-3	totalRows +1 - row	8	totalRows + row - 1	totalRows + row - 1	
5	1	5-4	totalRows +1 - row	9	totalRows + row - 1	totalRows + row - 1	
6	2	2	2	8	totalRows +5 - 2	2*totalRows - 2	
7	3	3	3	7	totalRows +5 - 3	2*totalRows - 3	
8	4	4	4	6	totalRows +5 - 4	2*totalRows - 4	
9	5	5	5	5	totalRows +5 - 5	2*totalRows - 5	

Row level Repetitive Action						
Variable: row	Column level Repetitive Action					
	Variable: col (Printing of star(*) characters)					
	min_val			max_val		
1	5	5-0	totalRows +1 - row	5	totalRows + row - 1	totalRows + row - 1
2	4	5-1	totalRows +1 - row	6	totalRows + row - 1	totalRows + row - 1
3	3	5-2	totalRows +1 - row	7	totalRows + row - 1	totalRows + row - 1
4	2	5-3	totalRows +1 - row	8	totalRows + row - 1	totalRows + row - 1
5	1	5-4	totalRows +1 - row	9	totalRows + row - 1	totalRows + row - 1
6	2	2	(6 - 4)	8	2*totalRows -(6 - 4)	2*totalRows - 6 + 4)
7	3	3	(7 - 4)	7	2*totalRows -(7 - 4)	2*totalRows - 7 + 4)
8	4	4	(8 - 4)	6	2*totalRows -(8 - 4)	2*totalRows - 8 + 4)
9	5	5	(9 - 4)	5	2*totalRows -(9 - 4)	2*totalRows - 9 + 4)

We can denote numeric patterns 6,7,8,9 as the variable: row. It will be more generalized and still not all.

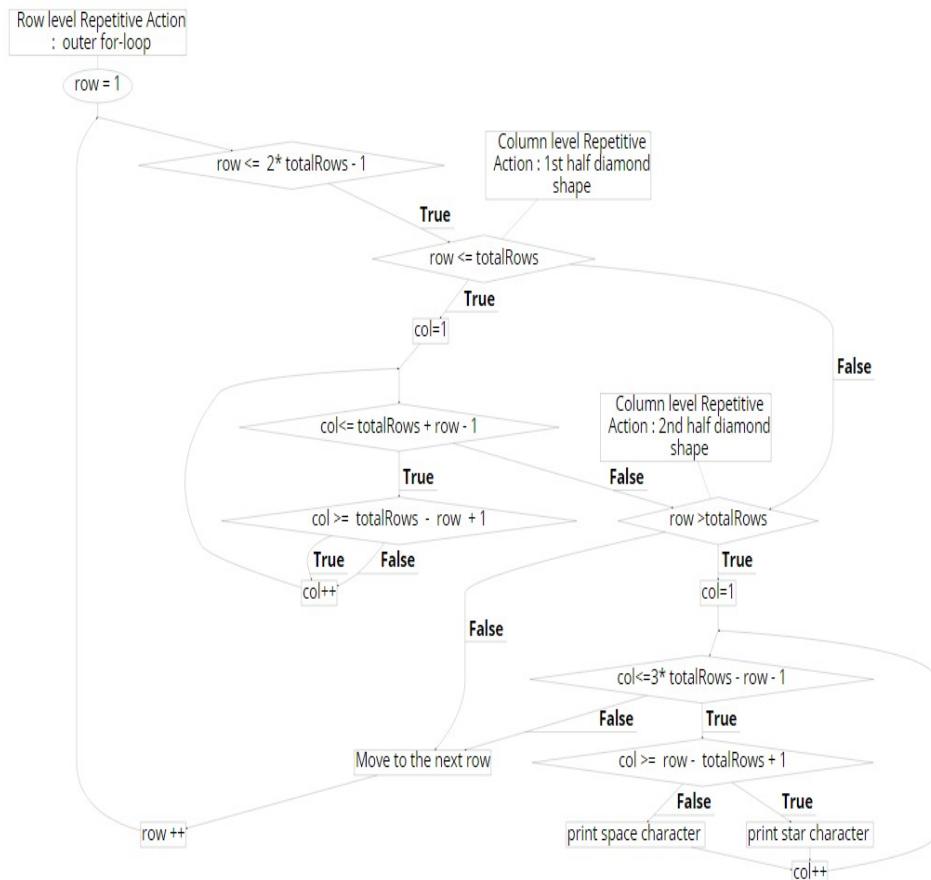
Row level Repetitive Action						
Variable: row	Column level Repetitive Action					
	Variable: col (Printing of star (*) characters)					
	min_val			max_val		
1	5	totalRows +1 - row		5	totalRows + row - 1	
2	4	totalRows +1 - row		6	totalRows + row - 1	
3	3	totalRows +1 - row		7	totalRows + row - 1	
4	2	totalRows +1 - row		8	totalRows + row - 1	
5	1	totalRows +1 - row		9	totalRows + row - 1	
6	2	(row - 4)		8	2*totalRows - row + 4)	
7	3	(row - 4)		7	2*totalRows - row + 4)	
8	4	(row - 4)		6	2*totalRows - row + 4)	
9	5	(row - 4)		5	2*totalRows - row + 4)	

We can replace repeated value 4 as (totalRows - 1). It will be more generalized.

Row level Repetitive Action						
Variable: row	Column level Repetitive Action					
	Variable: col (Printing of star (*) characters)					
	min_val			max_val		
1	5	totalRows +1 - row		5	totalRows + row - 1	
2	4	totalRows +1 - row		6	totalRows + row - 1	
3	3	totalRows +1 - row		7	totalRows + row - 1	
4	2	totalRows +1 - row		8	totalRows + row - 1	
5	1	totalRows +1 - row		9	totalRows + row - 1	
6	2	row -(totalRows - 1)		8	2*totalRows - row +(totalRows - 1)	
7	3	row -(totalRows - 1)		7	2*totalRows - row +(totalRows - 1)	
8	4	row -(totalRows - 1)		6	2*totalRows - row +(totalRows - 1)	
9	5	row -(totalRows - 1)		5	2*totalRows - row +(totalRows - 1)	

Row level Repetitive Action			
Variable:	Column level Repetitive Action		
row	Variable: col (Printing of star (*) characters)		
	min_val		max_val
1	totalRows +1 - row	totalRows +1 - row	totalRows + row - 1
2	totalRows +1 - row		totalRows + row - 1
3	totalRows +1 - row		totalRows + row - 1
4	totalRows +1 - row		totalRows + row - 1
5	totalRows +1 - row		totalRows + row - 1
6	row - totalRows + 1		3*totalRows - row - 1
7	row - totalRows + 1	row - totalRows + 1	3*totalRows - row - 1
8	row - totalRows + 1		3*totalRows - row - 1
9	row - totalRows + 1		3*totalRows - row - 1

The following given flow-chart easily show our logical expressions consideration till now.



Since there are two sets of generalized expressions, so we need two sets of column level **for-loop**.

If - condition	min_val	max_val
Logical expression (set1)	totalRows +1 - row	totalRows + row - 1
Logical expression (set 2)	row – totalRows + 1	3*totalRows - row – 1

Let's execute the following program.

PatternH2_1.py

```
totalRows = 4 # number of rows to display

# Row level Repetitive Action :
# Action1. Executes Column level Repetitive Action.
# Action2. The control of the program should move to the next row.
# so that the printing of characters is on a new line next time.
for row in range(1, 2 * totalRows):

    # 1st half : Column level Repetitive Action :
    if (row <= totalRows):
        for col in range(1, totalRows + row):
            if (col >=(totalRows - row + 1)):
                print("*", end="")
            else:
                print("", end="")

    # Ends inner for-loop_1

    # 2nd half : Column level Repetitive Action :
    if (row > totalRows):
        for col in range(1, 3 * totalRows - row):
            if (col >=(row - totalRows + 1)):
                print("*", end="")
            else:
                print("", end="")

    # Ends inner for-loop_2

    # move control of a program to the next row
    # in order to switch the printing of characters to the next line.
    print()
    # Ends outer for-loop
```

Output

```
*
```

```
***
```

```
*****
```

```
*****
```

```
***
```

```
*
```

Let's try to simplify the program further by simplifying the **if-conditions** and **loop-condition expressions**.

The **if-condition** gives the minimum limit, a starting point for which the program will start printing star("*") character(s) till the maximum value of variable col is given by the **loop condition**.

Let's consider;

colMin → variable type of integer to store the integer value of the column, a starting point from which program will start printing star("*") character.

colMax → variable type of integer to store the integer value of the column, a maximum value till program will stop printing star("*") character(s).

Let's analyze the following expression when **col** is a variable that **starts from value 0**. The variable:col will **contain the values as col=0,1,2,3**.

It implies $\text{col} \leq (4 - 1)$ or $\text{col} < 4$, both means the same.

Similarly,

When variable $\text{col}=5,6,\dots$, will have the values, which means the logical condition to cover these values can be expressed as;

$\text{col} \geq (4 + 1)$ or $\text{col} > 4$, both means the same.

In the same way, the **if-conditions** and **loop-condition expressions** can be simplified.

Variable: row	if-conditions	
	Variable: colMin	
	default	1
$\text{row} \leq \text{totalRows}$	$\text{col} \geq (\text{totalRows} - \text{row} + 1)$	$\text{col} > (\text{totalRows} - \text{row})$
$\text{row} > \text{totalRows}$	$\text{col} \geq (\text{row} - \text{totalRows} + 1)$	$\text{col} > (\text{row} - \text{totalRows})$

Variable: row	loop-conditions	
	Variable: colMax	
	default	1
$\text{row} \leq \text{totalRows}$	$\text{col} \leq (\text{totalRows} + \text{row} - 1)$	$\text{col} < (\text{totalRows} + \text{row})$
$\text{row} > \text{totalRows}$	$\text{col} \leq (3 * \text{totalRows} - \text{row} - 1)$	$\text{col} < (3 * \text{totalRows} - \text{row})$

Using variables **colMin & colMax**, we can merge the two inner-loops by dynamically assigning these expressions to these variables.

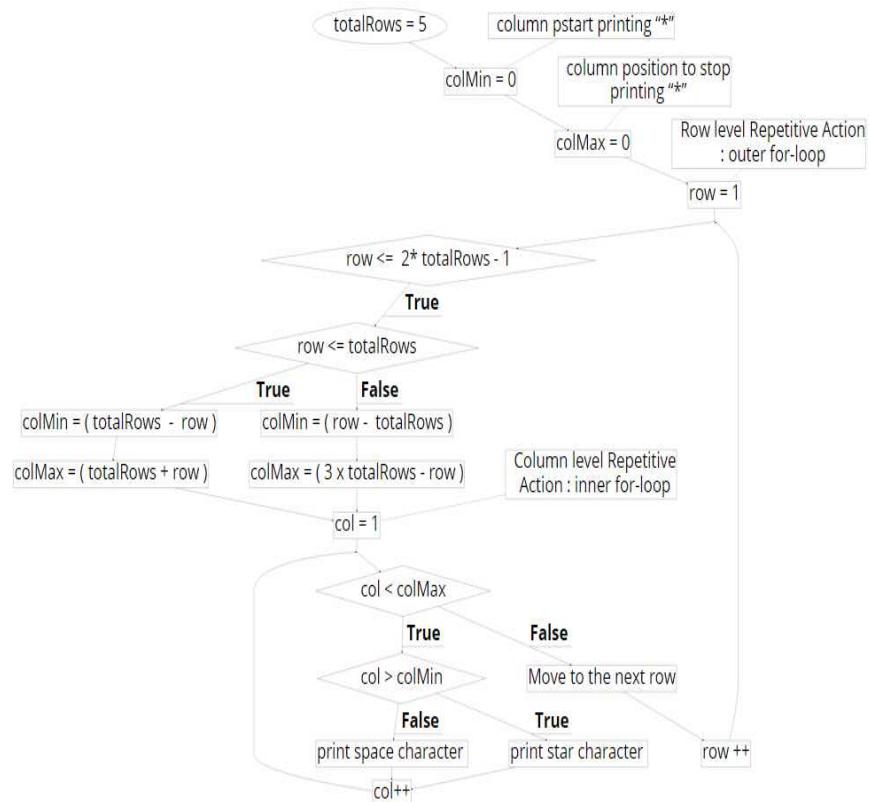
We can use the **if-else** statement to update the variables **colMin** & **colMax** dynamically, which are as follows.

```
if ( row <= totalRows ):
    colMin =( totalRows - row )
    colMax =( totalRows + row )
else:
    colMin =( row - totalRows )
    colMax =( 3 * totalRows - row )
```

Further, we can use variables **colMin** & **colMax** in the inner *for-loop*.

In this way, we can try merging two inner for-loops into a **single inner *for-loop***.

The following flowchart gives the overall idea of how the program following this should get updated as per our previous discussion.



So, let's update our previous program with these changes and execute the program. Also, test for a different number of lines by changing the value to 7 for the variable: `totalRows`.

PatternH2_2.py

```
totalRows = 5 # number of rows to display
colMin = 0 # column position to start printing character *
colMax = 0 # column position to stop printing character *

# Row level Repetitive Action :
# Action1. Executes Column level Repetitive Action:
# Action2. The control of the program should move to the next row
# so that the printing of characters is on a new line next time.
for row in range(1, 2 * totalRows):

    if (row <= totalRows):
        colMin =(totalRows - row)
        colMax =(totalRows + row)
    else:
        colMin =(row - totalRows)
        colMax =(3 * totalRows - row)
    # Column level Repetitive Action :
    for col in range(1, colMax):
        if (col > colMin):
            print("*", end="")
        else:
            print("", end="")

    # Ends inner for-loop
    # move control of a program to the next row
    # in order to switch the printing of characters to the next line.
    print()
# Ends outer for-loop
```

Output: `totalRows = 5`

```
*
```



```
***
```



```
*****
```



```
*****
```



```
*****
```



```
*****
```



```
*****
```



```
***
```



```
*
```

Output: `totalRows = 7`

```
*
```



```
***
```



```
*****
```



```
*****
```



```
*****
```



```
*****
```



```
*****
```



```
*****
```



```
***
```



```
*
```

Alternative solution3

Let's start with the logic trace table that we got from the previous solution1. The variables are row & totalRows=5. The variable spaces **represent the printing of space characters considered before printing of star(*) characters**. The variable col represents the printing of star(*) characters.

Row level Repetitive Action					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: spaces	Variable: col	Character count	Total Characters	
1	1,2,3,4	5	4 Spaces + 1 star	5	*
2	1,2,3	4, 5,6	3 Spaces + 3 stars	6	***
3	1,2	3, 4, 5,6,7	2 Spaces + 5 stars	7	*****
4	1	2, 3, 4, 5,6,7,8,	1 Space + 7 stars	8	*****
5	0	1, 2, 3, 4, 5,6,7,8,9	0 Space + 9 stars	9	*****
6	1	2, 3, 4, 5,6,7,8,	1 Space + 7 stars	8	*****
7	1,2	3, 4, 5,6,7	2 Spaces + 5 stars	7	*****
8	1,2,3	4, 5,6	3 Spaces + 3 stars	6	***
9	1,2,3,4	5	4 Spaces + 1 star	5	*

The printing of space(s) comes first before the Print star("*) character(s). If we closely observe the total character count divided into two halves. One part goes to the printing of space character(s), and the other goes to the printing of star character(s) as shown in the column **Character count**. Let's take out the minimum value (**min_val**) and maximum value(**max_val**) for printing space and star character(s) from the column **character count**. Hence, the following trace table can be drawn.

Row level Repetitive Action: totalRows=5					Expected Output
Variable: row	Column level Repetitive Action				
	Variable: col	space character(s) count	star character(s) count		
		min_val	max_val	min_val	max_val
		1			
1	1	4	totalRows - 1	1	1
2	1	3	4 - 1	1	3
3	1	2	3 - 1	1	5
4	1	1	2 - 1	1	7
5	0	0	1 - 1	1	9
6	1	1	0 + 1	1	7
7	1	2	1 + 1	1	5
8	1	3	2 + 1	1	3
9	1	4	3 + 1	1	1
		1			
		1 + 2			
		3 + 2			
		5 + 2			
		7 + 2			
		9 - 2			
		7 - 2			
		5 - 2			
		3 - 2			

We need to derive the generalized expression from the numeric pattern of minimum value (**min_val**) and maximum value(**max_val**). Those generalized expressions will help us to define the **for-loop** at the row and the column level. Two inner for-loops are required at the column level: one for printing space character(s) and the other for printing star character(s).

The column **max_val** for space character(s) count started at value 4 (**totalRows – 1**) and then decreased by value 1 consistently till it reached 0 at the row=5(total number of rows). Then it starts increasing by value 1 consistently till it reached maximum row value as;

$$9 = (2*5 - 1) \text{ or } (2*\text{totalRows} - 1), \text{ since } \text{totalRows}=5.$$

The column **max_val** for star character(s) count started at value 1 and increased by value 2 consistently until 9 at the row=5(total number of rows). Then it starts decreases by value 2 consistently till **maximum row value 9 or (2*5 – 1) or (2*totalRows – 1)**, since totalRows=5.

Let's consider;

spaceMax → variable type of integer to store integer value one at a time and starts at the value (**totalRows – 1**).

starMax → variable type of integer to store integer value one at a time and starts at the value 1.

Using variables **spaceMax & starCount**, we can use two inner-loops by dynamically assigning maximum limit to print space and star character(s), respectively.

Further, we can use variables **spaceMax & starMax** in inner **for-loops**.

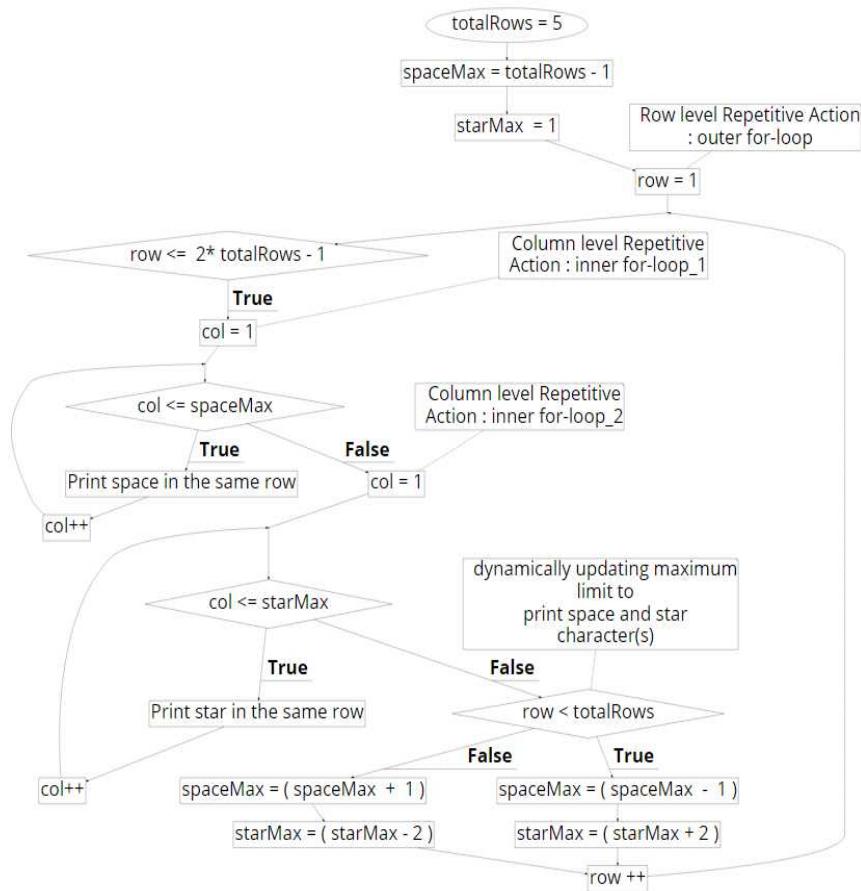
In this way, we can use two inner for-loops for printing a single row of space and star character(s), which are as follows.

row < totalRows	row >= totalRows
spaceMax = (spaceMax – 1)	spaceMax = (spaceMax + 1)
starMax = (starMax + 2)	starMax = (starMax – 2)

We can use an **if-else** statement to update these variables, spaceMax & starMax, dynamically.

Row level Repetitive Action						
Variable: row	Column level Repetitive Action					
totalRows=5	Variable: col		Variable: col			
	min_val	max_val			min_val	max_val
		Variable: spaceMax		Value Difference	Variable: starMax	Value Difference
1	1	totalRows	4	- 1	1	1
2	1	4	3	- 1	1	1
3	1	3	2	- 1	1	3
4	1	2	1	- 1	1	5
5	0	1	0	- 1	1	7
6	1	0	1	+ 1	1	9
7	1	1	2	+ 1	1	7
8	1	2	3	+ 1	1	5
(2*totalRows – 1)	1	3	4	+ 1	1	3

With the help of the following flowchart, we can get an overall idea.



Let's execute the following program.

PatternH3.py

```
totalRows = 5 # number of rows to display
spaceMax = totalRows - 1 # maximum count to print space
starMax = 1 # maximum count to print star *

# Row level Repetitive Action :
# Action1. Executes Column level Repetitive Action:
# Action2. Move to next row.
for row in range(1, 2 * totalRows):
    # Column level Repetitive Action : Print space in the same row
    for col in range(1, spaceMax + 1):
        print("", end="")
    # Ends inner for-loop_1

    # Column level Repetitive Action : Print star in the same row
    for col in range(1, starMax + 1):
        print("*", end="")
    # Ends inner for-loop_2

    print()# Move to next row.

    if (row < totalRows):
        spaceMax =(spaceMax - 1)
        starMax =(starMax + 2)
    else:
        spaceMax =(spaceMax + 1)
        starMax =(starMax - 2)

    # Ends outer for-loop
```

Output

```
*
```

```
***
```

```
*****
```

```
*****
```

```
*****
```

```
****
```

```
***
```

```
*
```

Problem H1:

The program (**PatternH3.py**) used the conditional operator <(less than) as shown:

```
if ( row < totalRows ):  
    spaceMax =(spaceMax - 1 )  
    starMax =(starMax + 2 )  
  
else:  
    spaceMax =(spaceMax + 1 )  
    starMax =(starMax - 2 )
```

Write a program **PatternH4.py** which will use conditional operator <=(less than equals to) instead of conditional operator <(less than) to generate the same output as done by **PatternH3.py** program. Modify the program (**PatternH3.py**) to generate the pattern **Pattern1H** by using conditional operator <=(less than or equals to).

```
if ( row <= totalRows ):  
    spaceMax =( spaceMax - 1 )  
    starMax =( starMax + 2 )  
  
else:  
    spaceMax =( spaceMax + 1 )  
    starMax =( starMax - 2 )
```

Test the output for total number of rows = 10 or variable: **totalRows = 10**.

Problems. Write programs to output the following by making use of a logic trace table.

Output1

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

Output2

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

Output3

PatternAA

Output4

卷之三

PatternBB

PatternCC

**
*
*
**

PatternDD

PatternEE

PatternFF

PatternGG

PatternHH

PatternII

Problem GBSwitchCase: Please apply a switch-case statement to generate the following output based on the value of border-style values.

Border Style = 1																									
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23		
1																		0							
2																	0	0							
3															0			0							
4														0					0						
5														0				0							
6														0			0								
7														0											
8					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
9				0										0			0					0			
10			0											0				0					0		
11		0												0				0						0	
12	0	0	0	0	0	0	0	0										0	0	0	0	0	0	0	

Border Style = 2																								
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
1																		0						
2																0	0	0						
3															0	0		0	0					
4														0	0			0	0					
5														0	0		0	0						
6														0	0	0								
7														0										
8					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9					0	0	0	0	0	0	0	0				0	0	0	0	0	0	0	0	0
10				0	0					0	0						0	0				0	0	
11				0	0	0	0	0	0	0							0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0										0	0	0	0	0	0	0

Border Style = 0																							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1												0											
2												0	0	0									
3											0	0	0	0	0								
4											0	0	0	0	0	0	0	0					
5											0	0	0	0	0								
6											0	0	0										
7											0												
8								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9							0	0	0	0	0	0	0		0	0	0	0	0	0	0	0	0
10						0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0	0
11			0	0	0	0	0	0	0	0					0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0									0	0	0	0	0	0	0	0

NumberPattern 1A

Our programming job is to print this particular logic-based pattern on one part of the computer screen. **Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes**—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique (**row, column**) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

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

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	1				
Row=2	2	2			
Row=3	3	3	3		
Row=4	4	4	4	4	
Row=5	5	5	5	5	5

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (**row, column**) numeric paired values.

For example;

(4, 3) means printing numeric characters at the assumed screen location 4th row and 3rd column.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	1,1				
Row=2	2,1	2,2			
Row=3	3,1	3,2	3,3		
Row=4	4,1	4,2	4,3	4,4	
Row=5	5,1	5,2	5,3	5,4	5,5

Analysis of output

It can also be interpreted in a tabular form.

row	1				
Col	1				
Output	1				

For row =1, $1 \leq \text{col} \leq 1$

row	2				
Col	1	2			
Output	2	2			

For row =2, $1 \leq \text{col} \leq 2$

row	3				
Col	1	2	3		
Output	3	3	3		

For row =3, $1 \leq \text{col} \leq 3$

row	4				
Col	1	2	3	4	
Output	4	4	4	4	

For row =4, $1 \leq \text{col} \leq 4$

row	5				
Col	1	2	3	4	5
Output	5	5	5	5	5

For row =5, $1 \leq \text{col} \leq 5$

COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action				Expected Output
variable: row	Column level Repetitive Action			
lower_limit <= variable:col <= upper_limit				
totalRows = 5	Range	lower_limit	upper_limit	
row = 1	$1 \leq \text{col} \leq 1$	1	1	1
row = 2	$1 \leq \text{col} \leq 2$	1	2	2 2
row = 3	$1 \leq \text{col} \leq 3$	1	3	3 3 3
row = 4	$1 \leq \text{col} \leq 4$	1	4	4 4 4 4
row = 5	$1 \leq \text{col} \leq 5$	1	5	5 5 5 5 5

The `upper_limit` value is based on how many times the value is repeated in a particular row.

The `lower_limit` of col variable has the same numeric value expression for every value of the row. It's consistent for all the rows.

Likewise, we need to develop such a consistent expression for `upper_limit`, which should suit any number of rows.

The other remaining variables are row & totalRows=5. Let's try to generalize using these variables.

For all the rows, the row value prints exactly the number of times the `upper_limit` col value.

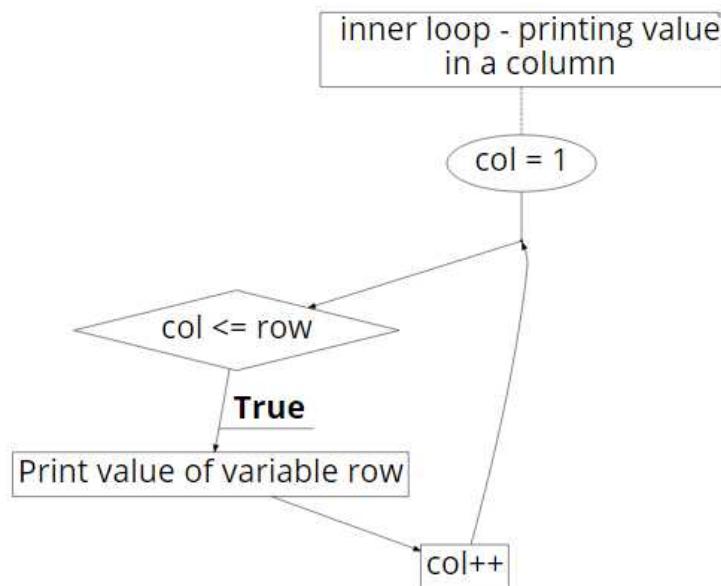
Column level Repetitive Action			
row	<code>lower_limit</code>	<code>upper_limit</code>	
		default	1
row = 1	1	1	row
row = 2	1	2	row
row = 3	1	3	row
row = 4	1	4	row
row = 5	1	5	row

So the `upper_limit` of col variable is generalized to row variable.

The `lower_limit` & `upper_limit` are consistent for all the rows. It's now generalized for any number of rows.

We can now define column level **for-loop**, which is for every row value.

Elements of Column level for-loop	
<code>min_val</code>	1
<code>max_val</code>	row
<code>variable_name</code>	col
<code>statement(s)</code>	<code>print(row, "", end="")</code>



ROW LEVEL REPETITIVE ACTION

The same following statements had been repeated 5 times(total number of rows).

Column level for-loop # inner for-loop

`print()# move to the next line`

The minimum and maximum value can be given for the **for-loop** as

`min_val = 1 & max_val = 5 or totalRows`

Let's consider;

row → variable type of integer to store integer value one at a time and represents row number between 1 and 5(totalRows). The **totalRows** has initially assigned the value 5.

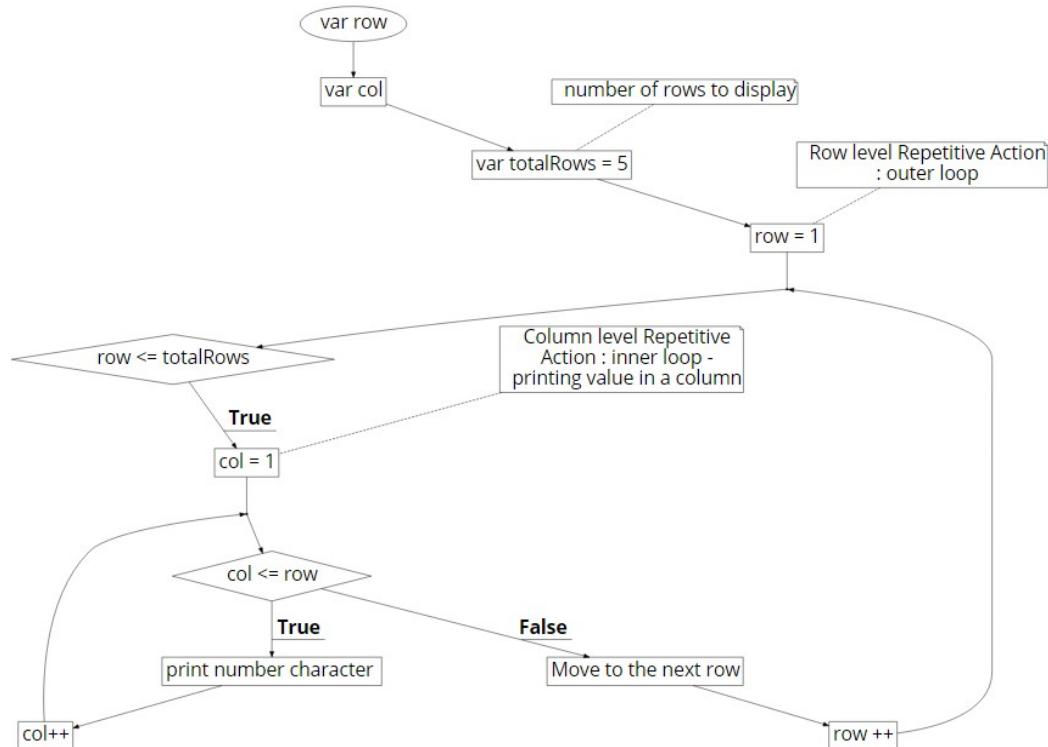
Outer **for-loop** starts at row=1 & will repeat till row \leq **totalRows**.

Elements Of Row Level for-loop

min_val	1
max_val	totalRows
variable_name	row
statement(s)	Column level for-loop <code>print()# move to the next line</code>

We got everything related to **for-loop** in order to implement row level repetition.

We can summarize all our findings in the following flowchart.



Let's execute the following program.

NumberPattern1A.py

```
totalRows = 5 # number of rows to display

for row in range(1, totalRows + 1):

    for col in range(1, row + 1):
        # added extra space for clarity in output.

    print(row, "", end="")

print()# move to the next line
```

Output: totalRows = 5

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

Output: totalRows = 7

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

NumberPattern 1B

Our programming job is to print this particular logic-based pattern on one part of the computer screen. **Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes**—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique (**row, column**) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

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

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1					1
Row=2					2 2
Row=3					3 3 3
Row=4					4 4 4 4
Row=5	5	5	5	5	5

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (**row, column**) numeric paired values.

For example;

(**4, 5**) means printing numeric characters at the assumed screen position **4th row and 5th column**.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1					1,5
Row=2				2,4	2,5
Row=3			3,3	3,4	3,5
Row=4		4,2	4,3	4,4	4,5
Row=5	5,1	5,2	5,3	5,4	5,5

Analysis of output

In this case, we need to take care of printing two different types of characters at a particular column in a particular row. The character types are the following.

1) Space character

2) Numeric character

Tabular form representation

row	1				
col	1	2	3	4	5
Output	space	space	space	space	1

For row =1, $1 \leq \text{spaces} \leq 4, 1 \geq \text{num} \geq 1$

row	2				
col	1	2	3	4	5
Output	space	space	space	2	2

For row =2, $1 \leq \text{spaces} \leq 3, 2 \geq \text{num} \geq 1$

row	3				
col	1	2	3	4	5
Output	space	space	3	3	3

For row =3, $1 \leq \text{spaces} \leq 2, 3 \geq \text{num} \geq 1$

row	4				
col	1	2	3	4	5
Output	space	4	4	4	4

For row =4, $1 \leq \text{spaces} \leq 1, 4 \geq \text{num} \geq 1$

row	5				
col	1	2	3	4	5
Output	5	5	5	5	5

For row =5, $5 \geq \text{num} \geq 1$

Observations

1. There is repetition at row level vertically, i.e., Row = 1 to 5.
2. There is repetition at col level horizontally., i.e., Col = 1 to 5.
This gets divided into two parts. One for repetitively printing spaces and the remaining column for printing numeric characters.
3. We need to explore the unknown logical expressions which are required to define the following given inner-loops. Those are marked as **Unknown** in the table given below.

For Loop	Lower limit	Upper limit
Outer loop	row=1	row<=5
Inner Loop - spaces	spaces=1	spaces <=?(Unknown)
Inner Loop - Number	num =?(Unknown)	num = 1

COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action						
row	Column level Repetitive Action					
	lower_limit <=spaces<= upper_limit			upper_limit >=num>=lower_limit		
	Range	lower_limit	upper_limit	Range	upper_limit	lower_limit
1	1 <=spaces<=4	1	4	1>=num>=1	1	1
2	1 <=spaces<=3	1	3	2>=num>=1	2	1
3	1 <=spaces<=2	1	2	3>=num>=1	3	1
4	1 <=spaces<=1	1	1	4>=num>=1	4	1
5		1	0	5>=num>=1	5	1

The lower_limit for spaces & num variable has the same numeric value expression for all the rows.

It's consistent for all the rows. Likewise, we need to develop such a consistent expression for upper_limit, for which it should suit any number of rows.

The other remaining variables are row & totalRows=5. Let's try to get a generalized logical condition using these variables.

Column level Repetitive Action							
row	lower_limit <=spaces<= upper_limit				upper_limit >=num >= lower_limit		
	lower_limit	upper_limit			upper_limit		lower_limit
		default	1	2	default	1	
1	1	4	totalRows - 1	totalRows - row	1	row	1
2	1	3	totalRows - 2	totalRows - row	2	row	1
3	1	2	totalRows - 3	totalRows - row	3	row	1
4	1	1	totalRows - 4	totalRows - row	4	row	1
5	1	0	totalRows - 5	totalRows - row	5	row	1

So the upper_limit of spaces & num variables are generalized. The lower_limit & upper_limit are consistent for all the rows. It's now generalized for n number of rows.

So our **If-condition**(lower_limit <=spaces<= upper_limit) becomes

$$1 \leq spaces \leq (totalRows - row)$$

The **If-condition**(upper_limit >=num >= lower_limit) becomes

$$row \geq num \geq 1$$

So it has two repetition processes, one for printing spaces and the second for printing numeric values. It means it requires **two inner for-loops** (nested for — loops).

For Loop	Lower limit	Upper limit
Outer loop	row=1	row<= totalRows
Inner Loop - spaces	col=1	col <=(totalRows – row)
Inner Loop - Number	num = row	num = 1

We can now define column level **for-loop**, which is for every row value.

Elements of Column level **for-loop**

min_val	1
max_val	totalRows -row
variable_name	col
statement(s)	<code>print("", end="")</code>

Elements of Column level **for-loop**

min_val	1
max_val	row
variable_name	num
statement(s)	<code>print(num, "", end="")</code>

ROW LEVEL REPETITIVE ACTION

The same following statements had been repeated 5 times(total number of rows).

Column level for-loop for printing space characters # **inner for-loop**

Column level for-loop for printing star characters # **inner for-loop**

`print()# move control to the next line`

We can observe from the pattern that the total number of rows is 5.

We have assigned a unique value to each row starting from 1 to 5.

So, the minimum and maximum value can be given for the **for-loop** at row level as

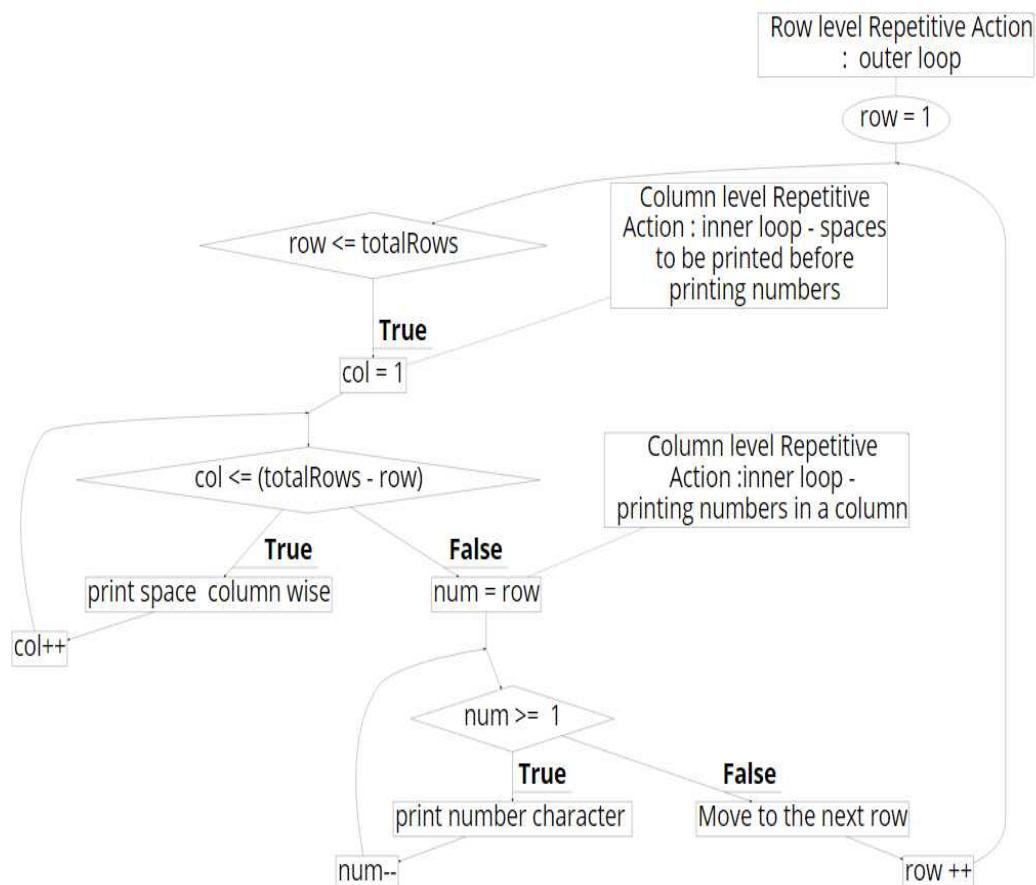
`min_val = 1 & max_val = 5 or totalRows`

Outer **for-loop** starts at row=1 & will repeat till row \leq **totalRows**.

Elements Of Row Level for-loop

min_val	1
max_val	totalRows
variable_name	row
statement(s)	Column level for-loop for printing <u>space characters</u> . Column level for-loop for printing <u>star characters</u> print()# move control to the next line

We got everything related to the **for-loop** required for row level repetition.



Let's execute the following program.

NumberPattern1B.py

```
totalRows = 5 # number of rows to display
for row in range(1, totalRows + 1):
    # spaces to be printed before printing numbers
    for col in range(1, totalRows - row + 1):
        print("", end="")

    for num in range(row, 0,-1):
        # added extra space for clarity in output.
        print(num, "", end="")

    print()# move control to the next line
```

Output: totalRows = 5

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

Output: totalRows = 7

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

Please, test for a different number of lines; let it be 7. It should correctly work so that we can be confirmed that our solution becomes generalized and fit for any number of lines.

Alternative solution

We observed previously that for row=5, the values of col and numeric output are just reversed.

Let's devise a solution by decrementing the for-loop for col variable (5 to 1), so the numeric output will match the row variable's value.

row	5				
col	1	2	3	4	5
output	5	4	3	2	1

For row =5, $5 \geq \text{num} \geq 1$, $1 \leq \text{col} \leq 5$

row	1				
col	5	4	3	2	1
output	space	space	space	space	1

row =1 , $1 \geq \text{col} \geq 1$ (print row value) or row $\geq \text{col} \geq 1$ (print row value)

row	2				
col	5	4	3	2	1
output	space	space	space	2	2

row = 2 , $2 \geq \text{col} \geq 1$ (print row value) or row $\geq \text{col} \geq 1$ (print row value)

row	3				
col	5	4	3	2	1
output	space	space	3	3	3

row = 3 , $3 \geq \text{col} \geq 1$ (print row value) or row $\geq \text{col} \geq 1$ (print row value)

row	4				
col	5	4	3	2	1
output	space	4	4	4	4

row = 4 , $4 \geq \text{col} \geq 1$ (print row value) or row $\geq \text{col} \geq 1$ (print row value)

row	5				
col	5	4	3	2	1
output	5	5	5	5	5

row = 5 , $5 \geq \text{col} \geq 1$ (print row value) or row $\geq \text{col} \geq 1$ (print row value)

So, if **-condition** =($\text{col} \geq 1$ and $\text{col} \leq \text{row}$)

$\text{col} \geq 1$ is redundant condition ,since it's a lower limit in the **inner for-loop** for $\text{col} = 5$ to 1.

So, if **-condition** reduced to ($\text{col} \leq \text{row}$)

The overall observation is at each row, either printing space characters or row values in a continuous fashion. If we take care of only the logical condition of printing spaces, then it's quite obvious that the condition can be expressed in if-else.

Note :-The printing of space or numeric characters is continuous.

Let's execute the following program.

NumberPattern1B_2.py

```
totalRows = 5 # number of rows to display

for row in range(1, totalRows + 1):
    # prints variable col value for the range (row >= col >=1)
    for col in range(totalRows, 0,-1):
        if(col <= row):
            print(row, end="")# 1 space
        else:
            print("", end="")# 2 spaces

    print()# moves control to the next line
```

Output: totalRows = 5

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

Output: totalRows = 7

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

Please, test for a different number of lines. Let it be 7. It should work correctly.

NumberPattern 1C

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique (row, column) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

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

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	5	5	5	5	5
Row=2	4	4	4	4	
Row=3	3	3	3		
Row=4	2	2			
Row=5	1				

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (row, column) numeric paired values.

For example;

(2, 3) means printing numeric characters at the assumed screen location 2nd row and 3rd column.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	1,1	1,2	1,3	1,4	1,5
Row=2	2,1	2,2	2,3	2,4	
Row=3	3,1	3,2	3,3		
Row=4	4,1	4,2			
Row=5	5,1				

Analysis of output

Let's devise a solution by incrementing the for-loop for col variable (1 to 5) and determining the if-condition(lower_limit <=col<= upper_limit) of col variable to give the desired numeric output.

row	1				
col	1	2	3	4	5
output	5	5	5	5	5

row =1 , 1 <=Col <= 5 (print upper_limit=5)

row	2				
col	1	2	3	4	5
output	4	4	4	4	

row = 2 ,(1 <=Col <= 4)(print upper_limit=4)

row	3				
col	1	2	3	4	5
output	3	3	3		

row = 3 ,(1 <=Col <= 3)(print upper_limit=3)

row	4				
col	1	2	3	4	5
output	2	2			

row = 4 ,(1 <=Col <= 2)(print upper_limit=2)

row	5				
col	1	2	3	4	5
output	1				

row = 5 ,(1 <=Col <= 1)(print upper_limit=1)

COLUMN LEVEL REPETITIVE ACTION

The numeric output will be the value of the row variable printed that many times equals the upper_limit of col variable. The upper_limit of the col variable varies with the value of the row variable.

Row level Repetitive Action			Expected Output
row	Column level Repetitive Action		
	lower_limit <=col<= upper_limit		
	lower_limit	upper_limit	
1	1	5	5 5 5 5 5
2	1	4	4 4 4 4
3	1	3	3 3 3
4	1	2	2 2
5	1	1	1

The lower_limit has the same numeric value expression for the rows. It's consistent for all the rows.

Likewise, we need to develop such a consistent expression for upper_limit, which should suit any number of rows. The other remaining variables are **row**, **totalRows=5**.

Let's try to generalize logical expressions using these variables.

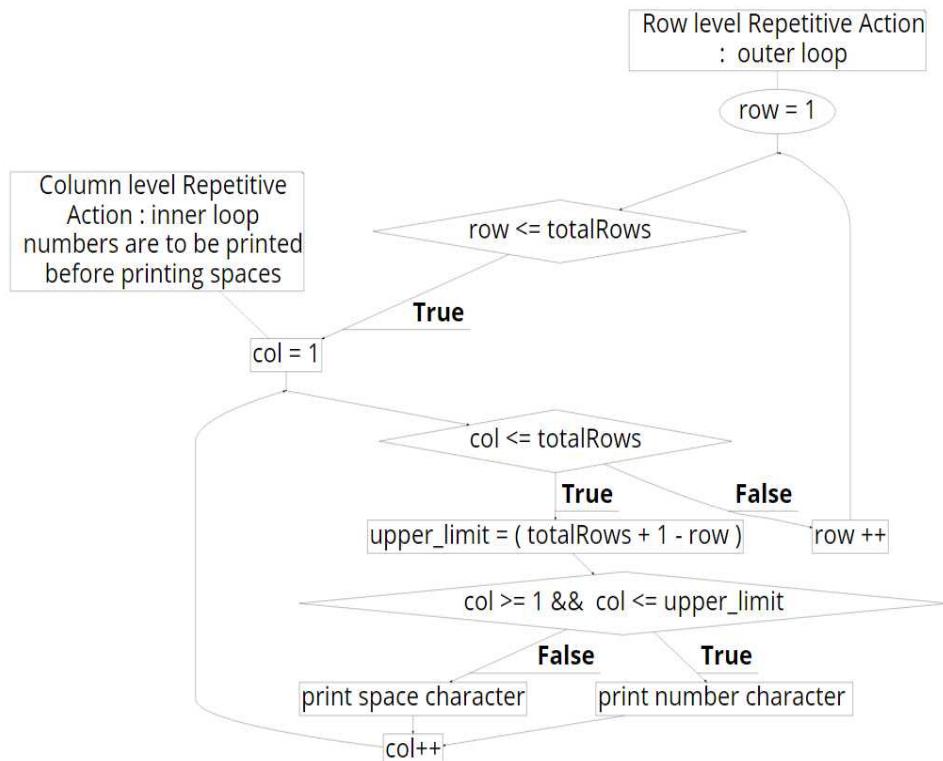
Column Level Repetitive Action							
row	lower_limit	lower_limit <= col <= upper_limit					
		upper_limit					
		default	1	2	3		
1	1	5	totalRows	(totalRows +1)- 1	(totalRows +1)- row		
2	1	4	totalRows -1	(totalRows +1)- 2	(totalRows +1)- row		
3	1	3	totalRows - 2	(totalRows +1)- 3	(totalRows +1)- row		
4	1	2	totalRows - 3	(totalRows +1)- 4	(totalRows +1)- row		
5	1	1	totalRows - 4	(totalRows +1)- 5	(totalRows +1)- row		

So the upper_limit of col variable is generalized to (totalRows + 1 - row).

The lower_limit & upper_limit are consistent for all the rows. It's now generalized for n number of rows. So our If-condition becomes;

If-condition	lower_limit <= col <= upper_limit
Mathematical expression	$1 \leq col \leq (\text{totalRows} + 1 - \text{row})$
syntax	$\text{col} \geq 1 \text{ and } \text{col} \leq (\text{totalRows} + 1 - \text{row})$

The flowchart can be shown, which is as follows.



NumberPattern1C_1.py

```
totalRows = 5 # number of rows to display

for row in range(1, totalRows + 1):

    # numbers to be printed before printing spaces
    for col in range(1, totalRows + 1):

        upper_limit =(totalRows + 1 - row)

        if(col >= 1 and col <= upper_limit):
            print(upper_limit, end="")
        else:
            print("", end="")

    # Ends inner loop

    print()# moves control to the next line

# Ends outer loop
```

Since inner for-loop's *lower_limit* matches with the if-condition's *lower_limit*.

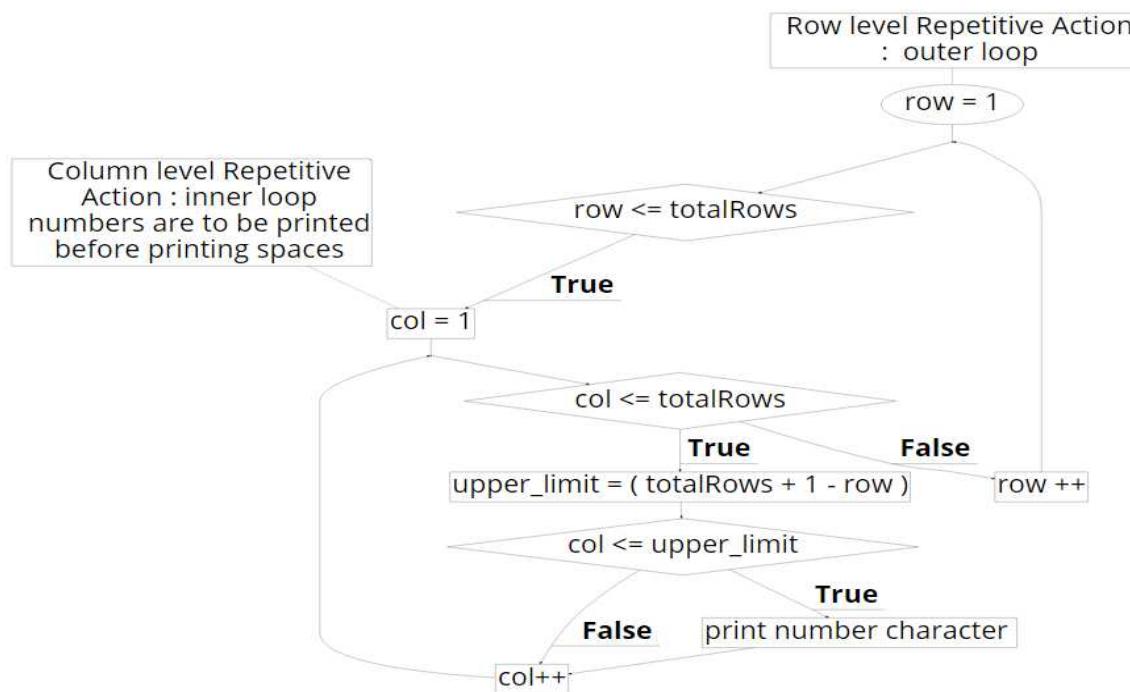
```
for(col = 1 col <= totalRows col++):
    if( col >= 1 and col <=( totalRows + 1 – row )):
        ---- statements -----
    else:
        # Print spaces
```

So, we can safely neglect *lower_limit* in the if-condition as it has already been taken care of by its own for-loop's starting value. Hence, our final if-condition will be,

If-condition	lower_limit <= col <= upper_limit
Mathematical expression	col <= ((totalRows + 1) – row)
syntax	col <= (totalRows + 1 - row)

After some space, if we need to print some text or characters, then, in that case, we are bound to put logic to print spaces first in a program. The other way round, it will not be a sensible exercise.

So in this particular case, we can avoid the participation of else-condition, which further simplifies our condition logic. We can further refine our flow-chart given as follows.



NumberPattern1C_2.py

```

totalRows = 5 # number of rows to display

for row in range(1, totalRows + 1):

    # spaces are to be printed before printing numbers.
    # This for-loop prints column-wise
    for col in range(1, totalRows + 1):
        upper_limit =(totalRows + 1 - row)
        if(col <= upper_limit):
            print(upper_limit, end="")# ends with space.

            # inner for-loop

    print()# moves control to the next line

    # outer for-loop

```

We know that the total number of column level repetitions is the total number of rows.

Furthermore, the inner loop's if-condition is responsible for decreasing the count of printing of numeric characters in columns. So the total number of repetitions should also go on decreasing as the number of rows increases. Rows, where not all column places are filled with numeric character(s), unnecessarily loop more. We can stop this extra looping by dynamically matching the *upper_limit* of the inner for-loop with the if-condition.

Hence, we can completely remove the if-condition and replace the logical expression of *upper_limit* of inner for-loop with the logical expression used in the **If-condition**.

NumberPattern1C_3.py

```
totalRows = 5 # number of rows to display

for row in range(1, totalRows + 1):

    upper_limit =(totalRows + 1 - row)

    for col in range(1, upper_limit + 1):
        print(upper_limit, end="")

    print()# moves control to the next line
```

Output: totalRows = 5

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

Output: totalRows = 7

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

Please, test the output for a different number of lines. Let it be 7. It should work correctly.

Alternative solution

This time let's use decrementing outer for-loop using row variable (5 to 1) and try to determine the logical condition in order to position the numeric character(s) in our row by column picture format.

row	5				
col	1	2	3	4	5
output	5	5	5	5	5

row = 5 , 1 <= Col <= 5 (print upper_limit=5)

row	4				
col	1	2	3	4	5
output	4	4	4	4	

row = 4 ,(1 <= Col <= 4)(print upper_limit=4)

row	3				
col	1	2	3	4	5
output	3	3	3		

row = 3 ,(1 <= Col <= 3)(print upper_limit=3)

row	2				
col	1	2	3	4	5
output	2	2			

row = 2 ,(1 <=Col <= 2)(print upper_limit=2)

row	1				
col	1	2	3	4	5
output	1				

row = 1 ,(1 <=Col <= 1)(print upper_limit=1)

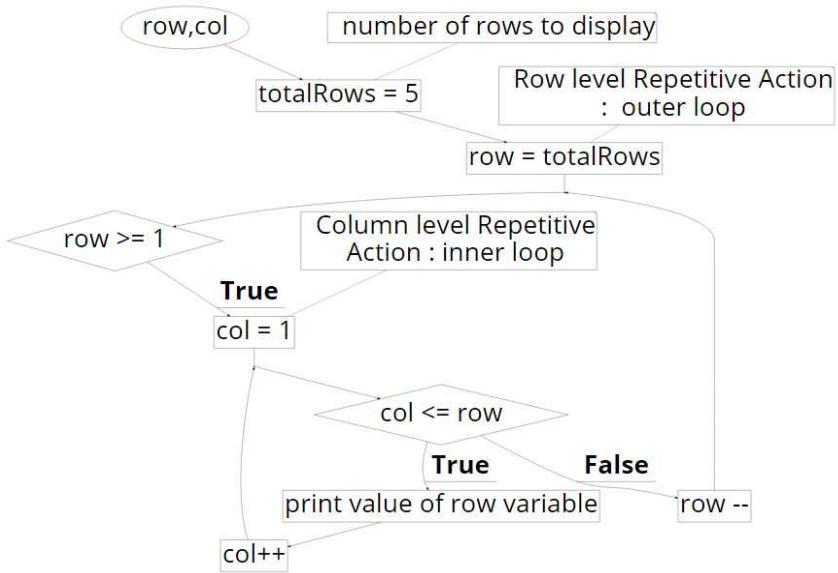
Column level Repetitive Action			Expected Output	
row	lower_limit <=col<= upper_limit			
	lower_limit	upper_limit		
5	1	5	5	5 5 5 5 5
4	1	4	4	4 4 4 4
3	1	3	3	3 3 3
2	1	2	2	2 2
1	1	1	1	1

The numeric output will be the value of the row variable printed that many times equals the upper_limit of the row variable. The upper_limit of the row variable varies and matches with the value of the row variable.

Column level Repetitive Action			Expected Output	
row	If-condition(lower_limit <=col<= upper_limit)			
	lower_limit	upper_limit		
5	1	row	5	5 5 5 5 5
4	1	row	4	4 4 4 4
3	1	row	3	3 3 3
2	1	row	2	2 2
1	1	row	1	1

So our loop-condition(lower_limit <=col<= upper_limit) becomes

loop-condition	lower_limit <=col<= upper_limit
Mathematical expression	$1 \leq col \leq row$
syntax	<code>for col in range(1, row + 1):</code>



Hence, our final program will be,

NumberPattern2C.py

```

totalRows = 5 # number of rows to display

for row in range(totalRows, 0,-1):

    # row value to be printed
    for col in range(1, row + 1):
        print(row, end ="")

    # Ends inner for-loop

    print()# moves control to the next line

# Ends outer for-loop

```

Output: totalRows = 5

```

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

```

Output: totalRows = 7

```

7 7 7 7 7 7 7
6 6 6 6 6 6
5 5 5 5 5
4 4 4 4
3 3 3
2 2
1

```

Please, test for a different number of lines. Let it be 7. It should work correctly.

NumberPattern 1D

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique (row, column) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

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

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	5	5	5	5	5
Row=2		4	4	4	4
Row=3			3	3	3
Row=4				2	2
Row=5					1

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (row, column) numeric paired values. For example; (3, 4) means printing numeric characters at the assumed screen location 3rd row and 4th column.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	1,1	1,2	1,3	1,4	1,5
Row=2		2,2	2,3	2,4	2,5
Row=3			3,3	3,4	3,5
Row=4				4,4	4,5
Row=5					5,5

Analysis of output

In this case, we need to take care of printing two different types of characters at a particular column in a particular row. The character types are the following.

1) Space character

2) Numeric character

row	5				
Col	1	2	3	4	5
Output	5	5	5	5	5

For row =1, $0 \leq \text{spaces} \leq 0$, $1 \leq \text{num} \leq 5$

row	4				
Col	1	2	3	4	5
Output	space	4	4	4	4

For row =2, $1 \leq \text{spaces} \leq 1$, $1 \leq \text{num} \leq 4$

row	3				
Col	1	2	3	4	5
Output	space	space	3	3	3

row =3, $1 \leq \text{spaces} \leq 2$, $1 \leq \text{num} \leq 3$

row	2				
Col	1	2	3	4	5
Output	space	space	space	2	2

For row =4, $1 \leq \text{spaces} \leq 3$, $1 \leq \text{num} \leq 2$

row	1				
Col	1	2	3	4	5
Output	space	space	space	space	1

For row =5, $1 \leq \text{spaces} \leq 4$, $1 \leq \text{num} \leq 1$

Observations

1. There is repetition at row level vertically, i.e., row = 1 to 5.
2. There is repetition at col level horizontally., i.e., col = 1 to 5.

This gets divided into two parts. One for repetitively printing spaces and the remaining column for printing numeric characters. We need to explore the unknown logical expressions which are required to define the following given inner-loops. Those are marked as **Unknown** in the table given below.

For Loop	Lower limit	Upper limit
Outer loop(decrementing)	row=1	row>=5
Inner Loop - spaces	spaces=1	spaces <=?(Unknown)
Inner Loop - Number	num = 1	num =? (Unknown)

F
o
r

COLUMN LEVEL REPETITIVE ACTION

It has two repetition processes, one for printing spaces and the second for printing numeric values.

It means it requires 2 inner for-loops and 1 outer loop for iterating rows—the relation between row value and upper limit for inner for-loop for spaces and numbers.

Row level Repetitive Action					Expected Output
row	Column level Repetitive Action				
	lower_limit <= col_sp <= upper_limit	lower_limit <= col_num <= upper_limit	lower_limit	upper_limit	
5	1	0	1	5	5 5 5 5 5
4	1	1	1	4	4 4 4 4
3	1	2	1	3	3 3 3
2	1	3	1	2	2 2
1	1	4	1	1	1

The lower_limit for spaces & num variable has the same numeric value expression for all the rows.

It's consistent for all the rows. Likewise, we need to develop such a consistent expression for upper_limit, for which it should suit any number of rows.

The other remaining variables are row & **totalRows=5**. Let's try to generalize using these variables.

Column level Repetitive Action						
row	lower_limit <= col_sp <= upper_limit			lower_limit <= col_num <= upper_limit		
	lower_limit	upper_limit	upper_limit	lower_limit	lower_limit	lower_limit
5	1	0	totalRows - row	1	5	5 5 5 5 5
4	1	1	totalRows - row	1	4	4 4 4 4
3	1	2	totalRows - row	1	3	3 3 3
2	1	3	totalRows - row	1	2	2 2
1	1	4	totalRows - row	1	1	1

Now the upper_limit of col_sp variable related inner for-loop is generalized, the same expression on all the rows. The lower_limit of col_num variable related inner for-loop is not generalized as not having the same expression on all the rows.

Since the lower_limit of col_num variable values matches with row variable values.

So, we can use the row variable to make it consistent given as follows.

Column level Repetitive Action							
row	lower_limit <= col_sp <= upper_limit			lower_limit <= col_num <= upper_limit			
	lower_limit		upper_limit	upper_limit		default	lower_limit
		default	1				1
5	1	0	totalRows - row	1	5	row	
4	1	1	totalRows - row	1	4	row	
3	1	2	totalRows - row	1	3	row	
2	1	3	totalRows - row	1	2	row	
1	1	4	totalRows - row	1	1	row	

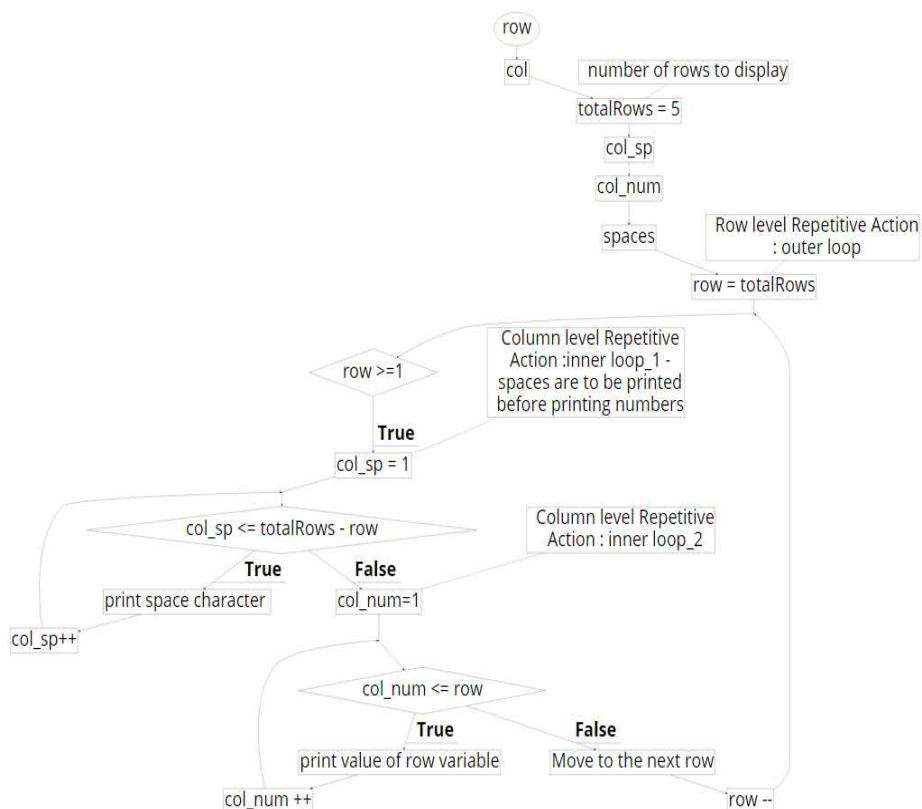
The lower_limit of the col_sp variable is the same for all the rows. The upper_limit of the col_num variable is the same for all the rows. It means the upper_limit of col_sp & col_num variables are generalized for any number of rows. So our inner for-loop condition for printing space characters becomes;

loop-condition	lower_limit <= col_sp <= upper_limit
Mathematical expression	$1 \leq col_sp \leq (totalRows - row)$
syntax	for col in range(1, totalRows - row + 1):

So our inner for-loop condition for printing numeric characters becomes

loop-condition	lower_limit <= col_num <= upper_limit
Mathematical expression	$1 \leq col_num \leq row$
syntax	for num in range(1, row + 1):

The flowchart for our observations can be given as follows.



Let's write the program as per the flowchart.

Please do test for a different number of lines. Let's **change** it to input value 7 and execute it.

NumberPattern1D.py

```
totalRows = 5 # number of rows to display
for row in range(totalRows, 0,-1):
# spaces are to be printed
    for col in range(1, totalRows - row + 1):
        print("", end="")# ends with a space
    # End inner for-loop_1
```

```
# row value is to be printed
    for num in range(1, row + 1):
        print(row, end="")# ends with a space
            # End inner for-loop_2
    print()# moves control to the next line
# End outer for-loop
```

Output: totalRows = 5

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

Output: totalRows = 7

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

Alternative solution

Since the column values need to be split clearly between printing spaces and numbers, we can try to use the if-else condition construct.

NumberPattern2D.py

```
totalRows = 5 # number of rows to display
for row in range(totalRows, 0,-1):
    for col in range(1, totalRows + 1):
        if (col <=(totalRows - row)):
            # spaces to be printed
        print("", end="")
    else:
        # row value to be printed
    print(row, end="")
# Ends inner for-loop
print()# moves control to the next line
# Ends outer for-loop
```

Output

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

Alternative solution

Let's devise a solution by decrementing the **for-loop** for col variable (5 to 1) so the numeric output will match with the value of row variable.

row	5
col	5
output	5

row = 5 , 5>=col>=1 (print row value) or row >=col>=1 (print row value)

row	4
col	5
output	space

row = 4 , 4>=col>=1 (print row value) or row >=col>=1 (print row value)

row	3
col	5
output	space

row = 3 , 3>=col>=1 (print row value) or row >=col>=1 (print row value)

row	2
col	5
output	space

row = 2 , 2>=col>=1 (print row value) or row >=col>=1 (print row value)

row	1
col	5
output	space

row = 1 , 1>=col>=1 (print row value) or row >=col>=1 (print row value)

Row level Repetitive Action					Expected Output
row	Column level Repetitive Action				
	upper_limit >=col>= upper_limit				
	upper_limit		lower_limit		
	default	1			
5	5	row	1		5 5 5 5 5
4	4	row	1		4 4 4 4
3	3	row	1		3 3 3
2	2	row	1		2 2
1	1	row	1		1

So, if-condition can be given as

if-condition	<code>upper_limit >= col >= lower_limit</code>
Mathematical expression	<code>row >= col >= 1</code>
syntax	<code>if(col >= 1 and col <= row)</code>

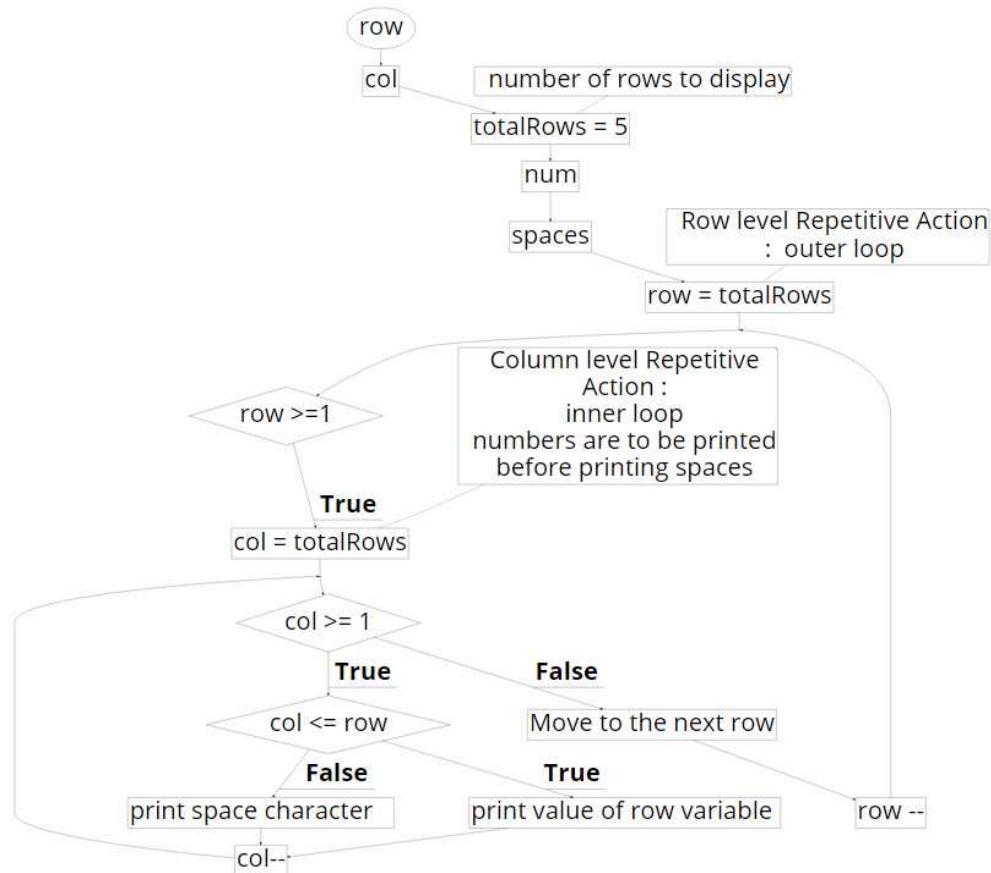
The part of if-condition `col >= 1` is already taken care of in the `lower_limit` of inner for-loop.

So, if –condition now becomes

if-condition	<code>upper_limit >= col >= lower_limit</code>
Mathematical expression	<code>col <= row</code>
syntax	<code>if(col <= row):</code>

The overall observation at each row is either its printing space characters or row values in a continuous fashion in every column but in a different way.

If we take care of the condition of printing numbers only, then the else part will for printing spaces and can be easily expressed in an **if-else statement** for printing desired output.



Let's try to execute the following program.

NumberPattern3D.py

```
totalRows = 5 # number of rows to display

for row in range(totalRows, 0,-1):

    for col in range(totalRows, 0,-1):

        if(col <= row):
            # row value to be printed
        print(row , end="")
        else:
            # spaces to be printed
        print(" " , end="")
    # Ends inner for-loop

    print()# moves control to the next line

# Ends outer for-loop
```

Output:totalRows = 5

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

Output:totalRows = 7

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

Please, test for a different number of lines. Let it be 7. It should work correctly.

NumberPattern 1E

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes —the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique (**row, column**) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

									1
								2	2
								3	3
								4	4
								5	5

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1					1				
Row=2					2	2	2		
Row=3			3	3	3	3	3		
Row=4		4							
Row=5	5								

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (**row, column**) numeric paired values. **For example;** (**4, 3**) means printing numeric characters at the assumed screen location **4th row and 3rd column**.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1					1 , 5				
Row=2				2 , 4	2 , 5	2 , 6			
Row=3			3 , 3	3 , 4	3 , 5	3 , 6	3 , 7		
Row=4		4 , 2	4 , 3	4 , 4	4 , 5	4 , 6	4 , 7	4 , 8	
Row=5	5 , 1	5 , 2	5 , 3	5 , 4	5 , 5	5 , 6	5 , 7	5 , 8	5 , 9

Analysis of output

row	1								
col	1	2	3	4	5	6	7	8	9
output					1				

For row = 1, $5 \leq \text{Col} \leq 5$, (printing value of row)

row	2								
col	1	2	3	4	5	6	7	8	9
output				2	2	2			

For row = 2, $4 \leq \text{Col} \leq 6$, (printing value of row)

row	3								
col	1	2	3	4	5	6	7	8	9
output		3	3	3	3	3	3		

For row = 3, $3 \leq \text{Col} \leq 7$, (printing value of row)

row	4								
col	1	2	3	4	5	6	7	8	9
output		4	4	4	4	4	4	4	

For row = 4, $2 \leq \text{Col} \leq 8$, (printing value of row)

row	5								
col	1	2	3	4	5	6	7	8	9
output	5	5	5	5	5	5	5	5	5

For row = 5, $1 \leq \text{Col} \leq 9$, (printing value of row)

COLUMN LEVEL REPETITIVE ACTION

A total maximum of 9 columns is needed to print the bottom of the pattern.

Row level Repetitive Action				Expected Output				
row	Column level Repetitive Action							
	(lower_limit <= col <= upper_limit)							
	range		lower_limit	upper_limit				
1	$5 \leq \text{Col} \leq 5$		5	5		1		
2	$4 \leq \text{Col} \leq 6$		4	6		2	2	2
3	$3 \leq \text{Col} \leq 7$		3	7		3	3	3
4	$2 \leq \text{Col} \leq 8$		2	8		4	4	4
5	$1 \leq \text{Col} \leq 9$		1	9		5	5	5

Both **lower_limit** & **upper_limit** column values are not consistent for all the rows. We need to develop consistent expressions for both **upper_limit** and **lower_limit**, which should suit n number of rows. The other remaining variables are row, totalRows = 5. Let's try to generalize the numeric sequences of the **lower_limit** & **upper_limit** columns into a valid logical expression using variables.

Column level Repetitive Action						
row	lower_limit <= col <= upper_limit					
	lower_limit		upper_limit			
	1		2		1	
1	5	totalRows	totalRows +1 - 1	5	totalRows	totalRows +1 - 1
2	4	totalRows -1	totalRows +1 - 2	6	totalRows + 1	totalRows + 2 - 1
3	3	totalRows -2	totalRows +1 - 3	7	totalRows + 2	totalRows + 3 - 1
4	2	totalRows -3	totalRows +1 - 4	8	totalRows + 3	totalRows + 4 - 1
5	1	totalRows -4	totalRows +1 - 5	9	totalRows + 4	totalRows + 5 - 1

Column level Repetitive Action				
row	If-condition(lower_limit <= col <= upper_limit)			
	lower_limit		upper_limit	
	default	3	default	3
1	5	totalRows +1 - row	5	totalRows + row - 1
2	4	totalRows +1 - row	6	totalRows + row - 1
3	3	totalRows +1 - row	7	totalRows + row - 1
4	2	totalRows +1 - row	8	totalRows + row - 1
5	1	totalRows +1 - row	9	totalRows + row - 1

The lower_limit & upper_limit are consistent for all the rows. It's now generalized for any number of rows to print numeric value. So our If-condition becomes;

if-condition	lower_limit <= col <= upper_limit
Mathematical expression	(totalRows +1 - row) <= col <= (totalRows + row - 1)
syntax	if(col >= ((totalRows - row)+ 1) and col <= ((totalRows + row)- 1)):

When the if-condition is satisfied, it will print the row value at that position at a particular row or else will Print space character. The diagram shows that, in total, 9 columns are necessary to print the entire bottom of the pattern. The for-loop condition is given as;

loop-condition	lower_limit <= col <= upper_limit
Mathematical expression	1 <= col <= 9
syntax	for col in range(1, 9 + 1):

The upper_limit of the col variable of inner for-loop is the value 9, which is a hard-coded numerical value required to be generalized.

We can represent value 9 = (totalRows + 4), since totalRows = 5.

The hard-coded values in logical conditions make the program fail for different input values. Somehow, it has to be generalized using variables that were considered in writing our program.

loop-condition	<code>lower_limit <= col <= upper_limit</code>
Mathematical expression	$1 \leq col \leq totalRows + 4$ $1 \leq col \leq totalRows + (totalRows - 1)$ $1 \leq col \leq (2 * totalRows - 1)$
syntax	<code>for col in range(1, 2 * totalRows):</code>

We are now able to generalize using variable **totalRows**.

Let's try to verify the upper limit for total columns which we will display for each row

for totalRows = 5 , the col <= 9 ($2 * 5 - 1$)

for totalRows = 4 , the col <= 7 ($2 * 4 - 1$)

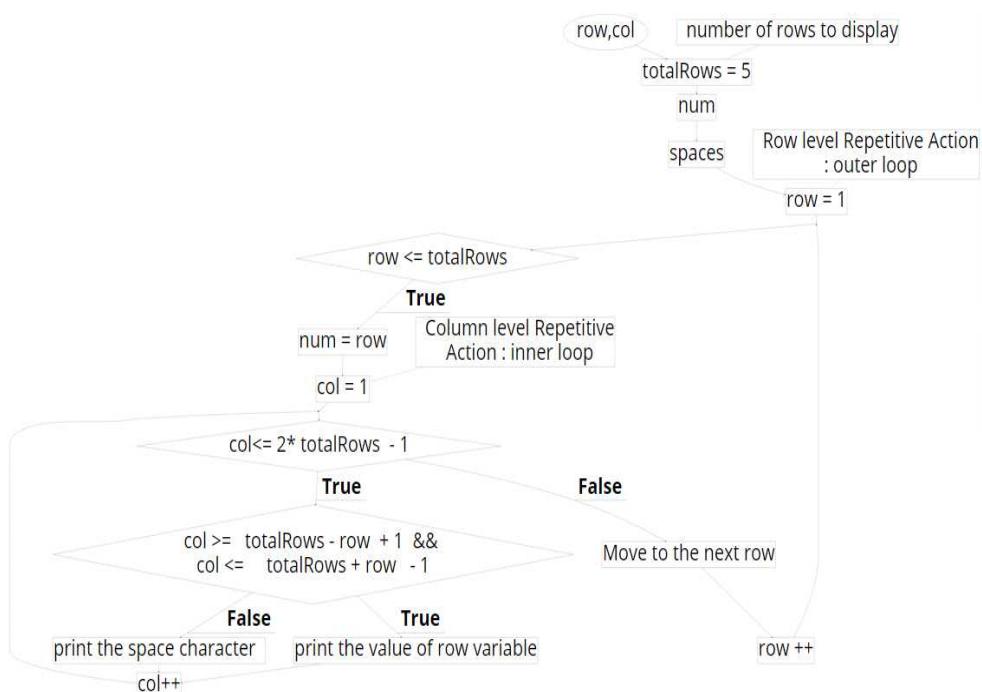
for totalRows = 3 , the col <= 5 ($2 * 3 - 1$)

for totalRows = 2 , the col <= 3 ($2 * 2 - 1$)

for totalRows = 1 , the col <= 1 ($2 * 1 - 1$)

So this generalization for total columns as per given rows holds valid.

All these observations can be easily captured in a flowchart given as follows.



Let's write the program based on the observations captured in a flowchart.

NumberPattern1E.py

```
totalRows = 5 # number of rows to display

for row in range(1, totalRows + 1):

    # repetition happens ( 2* totalRows - 1) times for columns

    for col in range(1, 2 * totalRows):

        if (col >=((totalRows - row)+ 1) and col <=((totalRows + row)- 1)):

            # row value to be printed
            print(row, end="")# ends with a space

        else:

            # spaces to be printed
            print("", end="")#ends with a space

    # end inner for - loop

    print()# moves control to the next line

# end outer for - loop
```

Output:totalRows=5

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

Output:totalRows=8

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

Please, test for a different number of lines. Let it be 8. It should work correctly.

NumberPattern 1F

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique (row, column) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

1
0 0
1 1 1
0 0 0 0
1 1 1 1 1

1				
0	0			
1	1	1		
0	0	0	0	
1	1	1	1	1

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	1				
Row=2	0	0			
Row=3	1	1	1		
Row=4	0	0	0	0	
Row=5	1	1	1	1	1

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (row, column) numeric paired values. **For example;** (4, 3) means printing numeric characters at the assumed screen location 4th row and 3rd column.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	1,1				
Row=2	2,1	2,2			
Row=3	3,1	3,2	3,3		
Row=4	4,1	4,2	4,3	4,3	
Row=5	5,1	5,2	5,3	5,4	5,5

Analysis of output

It can also be interpreted in a tabular form.

row	1				
Col	1				
Output	1				

For row =1, $1 \leq \text{col} \leq 1$

row	2				
Col	1	2			
Output	0	0			

For row =2, $1 \leq \text{col} \leq 2$

row	3				
Col	1	2	3		
Output	1	1	1		

For row =3, $1 \leq \text{col} \leq 3$

row	4				
Col	1	2	3	4	
Output	0	0	0	0	

For row =4, $1 \leq \text{col} \leq 4$

row	5				
Col	1	2	3	4	5
Output	1	1	1	1	1

For row =5, $1 \leq \text{col} \leq 5$

COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action			Expected Output	
row	Column level Repetitive Action			
	Values repeated	lower_limit <= col <= upper_limit	lower_limit	upper_limit
1	1	1	1	1
2	0	1	2	0 0
3	1	1	3	1 1 1
4	0	1	4	0 0 0 0
5	1	1	5	1 1 1 1 1

The *upper_limit* value is based on how many times the value is repeated in a particular row.

The *lower_limit* of col variable has the same numeric value expression for every value of the row. It's consistent for all the values rows. Likewise, we need to develop such a consistent expression for *upper_limit*, which should suit any number of rows using available variables. The variables are *row* & *totalRows=5*.

Moreover, the repeated values in the column appear alternatively in a row. It seems we can establish a relationship between the appearing of 0 or 1 value with the row value.

If we observe, it is following the even-odd rule. For even value rows, when the row value is a multiple of 2, we need to print zeros or ones otherwise. Thus, we can try to output 0 or 1 depending upon the value of the **row** variable. Let's try to generalize columns **upper_limit** and **values_repeated** by expressing in terms of available variables.

Column level Repetitive Action						Expected Output
row	values_repeated		lower_limit <= col <= upper_limit			
	default	1	lower_limit	upper_limit	1	
1	1	row%2==1	1	1	row	1
2	0	row%2==0	1	2	row	0 0
3	1	row%2==1	1	3	row	1 1 1
4	0	row%2==0	1	4	row	0 0 0 0
5	1	row%2==1	1	5	row	1 1 1 1 1

For all the rows, the value prints exactly the number of times the **upper_limit** of col value.

So the **upper_limit** of col variable is generalized to **row** variable.

The **lower_limit & upper_limit** are consistent for all the rows.

It's now generalized for any number of rows.

We can now define column level **for-loop**, which is for every row value.

Elements of Column level for-loop	
min_val	1
max_val	row
variable_name	col
statement(s)	print(row, "", end="")

Also, printing 0 or 1 can be given by checking the if-condition (`row%2==1`) or (`row%2==0`).

General Rule

Such a situation may come where we need to check whether the logical condition is following the pattern of (**row+col**) values or (**row-col**) values or (**row*col**) values or (**row%col**) values.

The general rule is, somehow, we need to express hard-coded values in terms of available variable(s) if following the same numeric pattern/variations or simple math formula made up of available variable(s).

ROW LEVEL REPETITIVE ACTION

The same following statements had been repeated 5 times(total number of rows).

Column level for-loop # inner for-loop

print()# Move to the next row

The minimum and maximum value can be given for the **for-loop** as

min_val = 1 & max_val = 5 or totalRows

Let's consider;

row → variable type of integer to store integer value one at a time and represents row number between 1 and 5(totalRows). The **totalRows** has initially assigned the value 5.

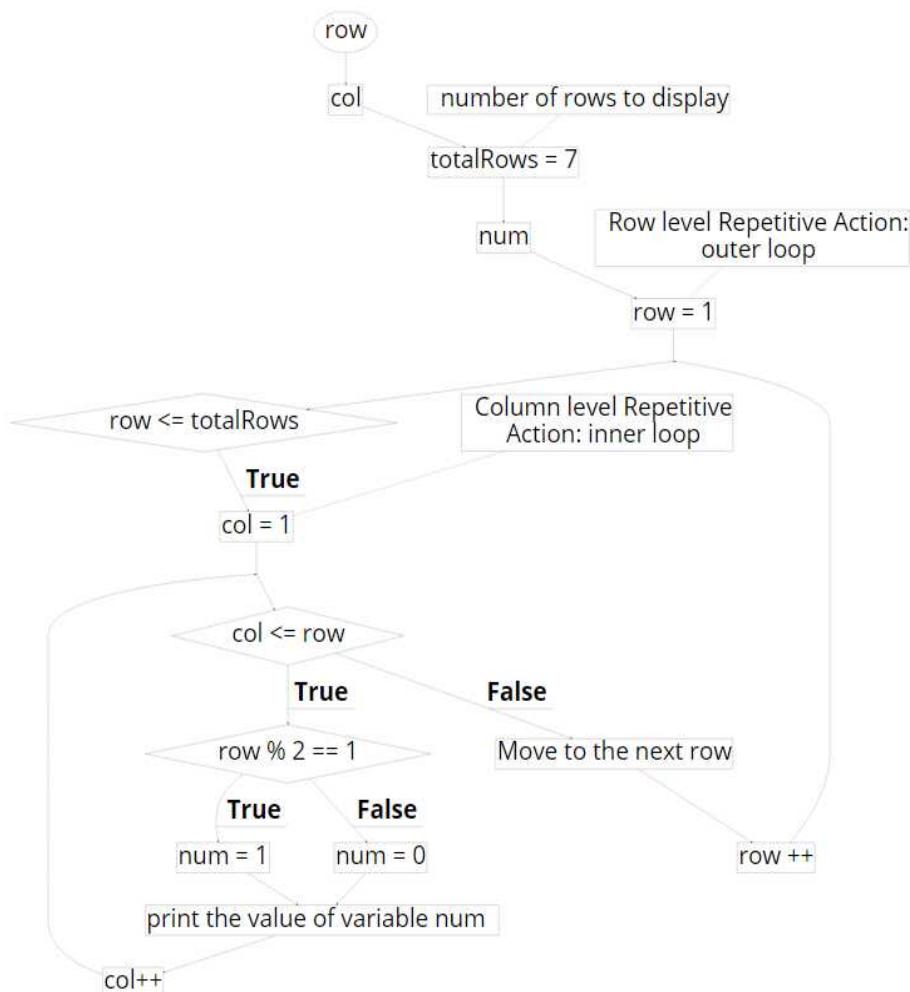
Outer **for-loop** starts at row=1 & will repeat till row \leq **totalRows**.

We got everything related to **for-loop** for row level repetition.

Elements Of Row Level for-loop

min_val	1
max_val	totalRows
variable_name	row
statement(s)	Column level for-loop # Move to the next row print()

We can summarize our observations in a flowchart, as shown in the following.



Let's execute the following program.

NumberPattern1F.py

```
totalRows = 7 # number of rows to display

# Row level Repetitive Action:
for row in range(1, totalRows + 1):

    # Column level Repetitive Action:
    # print the value of variable:num column-wise
    for col in range(1, row + 1):

        if (row % 2 == 1):
            num = 1
        else:
            num = 0

        print(num, end="")
        # Ends inner loop

        # move control of a program to the next row in order to
        # switch the printing of characters to the next line.
    print()

    # Ends outer for-loop
```

Output

```
1
0 0
1 1 1
0 0 0 0
1 1 1 1 1
0 0 0 0 0 0
1 1 1 1 1 1 1
```

NumberPattern AA

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique (row, column) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

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

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	1				
Row=2	1	2			
Row=3	1	2	3		
Row=4	1	2	3	4	
Row=5	1	2	3	4	5

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (row, column) numeric paired values. **For example;** (4, 3) means printing numeric characters at the assumed screen location 4th row and 3rd column.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	1,1				
Row=2	2,1	2,2			
Row=3	3,1	3,2	3,3		
Row=4	4,1	4,2	4,3	4,3	
Row=5	5,1	5,2	5,3	5,4	5,5

Analysis of output

It can also be interpreted in a tabular form.

row	1				
Col	1				
Output	1				

For row =1, $1 \leq \text{col} \leq 1$

row	2				
Col	1	2			
Output	1	2			

For row =2, $1 \leq \text{col} \leq 2$

row	3				
Col	1	2	3		
Output	1	2	3		

For row =3, $1 \leq \text{col} \leq 3$

row	4				
Col	1	2	3	4	
Output	1	2	3	4	

For row =4, $1 \leq \text{col} \leq 4$

row	5				
Col	1	2	3	4	
Output	1	2	3	4	5

For row =5, $1 \leq \text{col} \leq 5$

Observations

1. There is repetition required at row level vertically, i.e., row = 1 to 5.
 2. There is repetition required at col level horizontally., i.e., col = 1 to 5.
- So it's a 2-way repetition process which means it requires 2 for-loops (nested for-loop).

For Loop	Lower limit	Upper limit
Outer loop	row=1	row\leq5
Inner Loop	col=1	(Unknown) Need to find a relation to how exactly it's varying.

COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action			Expected Output
row	Column Level Repetitive Action		
	loop-condition(lower_limit <=col<= upper_limit)	lower_limit	upper_limit
row = 1		1	1
row = 2		1	2
row = 3		1	3
row = 4		1	4
row = 5		1	5

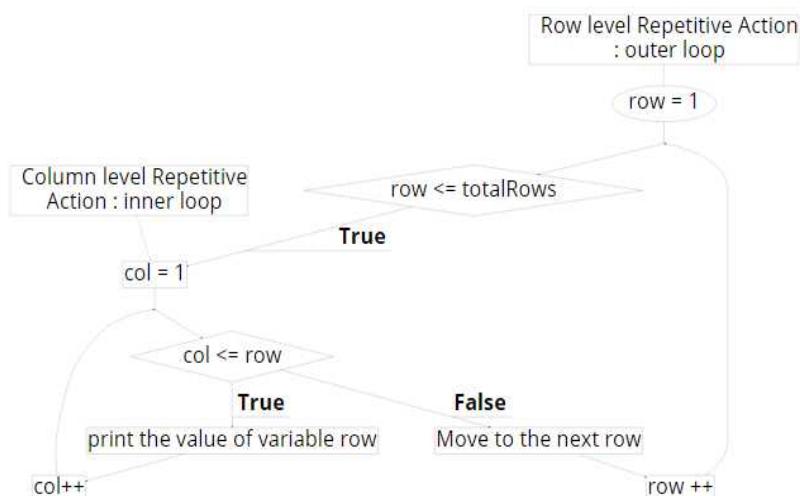
The lower_limit has the same numeric value expression for the rows. It's consistent for all the rows. Likewise, we need to develop such a consistent expression for upper_limit, which should suit any number of rows. The other remaining variables are row & totalRows=5. Let's try to generalize numeric patterns using these variables.

Row level Repetitive Action			Expected Output
row	Column Level Repetitive Action		
	loop-condition(lower_limit <=col<= upper_limit)	lower_limit	upper_limit
row = 1		1	row
row = 2		2	row
row = 3	1	3	row
row = 4		4	row
row = 5		5	row

So the upper_limit of col variable is generalized to (row). For all the rows, the col values match exactly the output that we need. The lower_limit & upper_limit are consistent for all the rows. It's now generalized for any number of rows. So our **loop-condition** becomes;

loop-condition	lower_limit <=col<= upper_limit
Mathematical expression	$1 \leq col \leq row$
syntax	<code>for col in range(1, row + 1):</code>

The related flowchart can be given as follows.



Let's implement the program.

NumberPatternAA.py

```
totalRows = 5 # number of rows to display

for row in range(1, totalRows + 1):

    for col in range(1, row + 1):

        # ends with extra space for clarity in output.
        print(col,end ="")

    # end inner for-loop

    print()# moves control to the next line

# end outer for-loop
```

Output: totalRows = 5

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

Output: totalRows = 10

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

Please, test for a different number of lines. Let it be 10. It should work correctly.

NumberPattern BB

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique (row, column) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

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

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1					1
Row=2				2	1
Row=3			3	2	1
Row=4		4	3	2	1
Row=5	5	4	3	2	1

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (row, column) numeric paired values. **For example;** (4, 3) means printing numeric characters at the assumed screen location 4th row and 3rd column.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1					1,5
Row=2				2,4	2,5
Row=3			3,3	3,4	3,5
Row=4		4,2	4,3	4,4	4,5
Row=5	5,1	5,2	5,3	5,4	5,5

Analysis of output

In this case, we need to take care of printing two different types of characters at a particular column in a particular row:

- 1) Space character
- 2) Numeric character

row	1				
col	1	2	3	4	5
output	space	space	space	space	1

For row =1, $1 \leq \text{spaces} \leq 4, 1 \geq \text{num} \geq 1$

row	2				
Col	1	2	3	4	5
Output	space	space	space	2	1

For row =2, $1 \leq \text{spaces} \leq 3, 2 \geq \text{num} \geq 1$

row	3				
Col	1	2	3	4	5
Output	space	space	3	2	1

For row =3, $1 \leq \text{spaces} \leq 2, 3 \geq \text{num} \geq 1$

row	4				
Col	1	2	3	4	5
Output	space	4	3	2	1

For row =4, $1 \leq \text{spaces} \leq 1, 4 \geq \text{num} \geq 1$

row	5				
Col	1	2	3	4	5
Output	5	4	3	2	1

For row =5, $5 \geq \text{num} \geq 1$

Observations

1. There is repetition at row level vertically, i.e., Row = 1 to 5.
2. There is repetition at col level horizontally., i.e., Col = 1 to 5.

This gets divided into two parts. One for repetitively printing spaces and the remaining column for printing numeric characters. We need to explore the unknown logical expressions which are required to define the following given inner-loops. Those are marked as **Unknown** in the table given below.

For Loop	Lower limit	Upper limit
Outer loop	row=1	row<=5
Inner Loop - spaces	spaces=1	Unknown
Inner Loop - Number	Unknown	num = 1

COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action					Expected Output
row	Column level Repetitive Action				
	lower_limit <=spaces<= upper_limit	upper_limit>=num >= lower_limit			
	lower_limit	upper_limit	upper_limit	lower_limit	
row = 1	1	4	1	1	1
row = 2	1	3	2	1	2 1
row = 3	1	2	3	1	1 3 2 1
row = 4	1	1	4	1	4 3 2 1
row = 5	1	0	5	1	5 4 3 2 1

The lower_limit for spaces & num variable has the same numeric value expression for all the rows. It's consistent for all the rows. Likewise, we need to develop such a consistent expression for upper_limit, for which it should suit any number of rows. The other remaining variables are **row** and **totalRows = 5**. Let's try to generalize data of the column **upper_limit** using these variables.

Row level Repetitive Action					
row	Column level Repetitive Action				
	lower_limit <=spaces<= upper_limit			upper_limit>=num >= lower_limit	
	lower_limit	upper_limit		upper_limit	lower_limit
row = 1		4	totalRows - 1	totalRows - row	1 row
row = 2		3	totalRows - 2	totalRows - row	2 row
row = 3	1	2	totalRows - 3	totalRows - row	3 row 1
row = 4		1	totalRows - 4	totalRows - row	4 row
row = 5		0	totalRows - 5	totalRows - row	5 row

Row level Repetitive Action					
row	Column level Repetitive Action				
	lower_limit <=spaces<= upper_limit			upper_limit>=num >= lower_limit	
	lower_limit	upper_limit		upper_limit	lower_limit
row = 1		totalRows - row		row	
row = 2		totalRows - row		row	
row = 3	1	totalRows - row	totalRows - row	row row	1
row = 4		totalRows - row		row	
row = 5		totalRows - row		row	

So the upper_limit of spaces & num variables are generalized. The lower_limit & upper_limit are consistent for all the rows. It's now generalized for any number of rows. Since it has two repetition processes;

- one repetition process for printing spaces.
- second repetition process for printing numeric values.

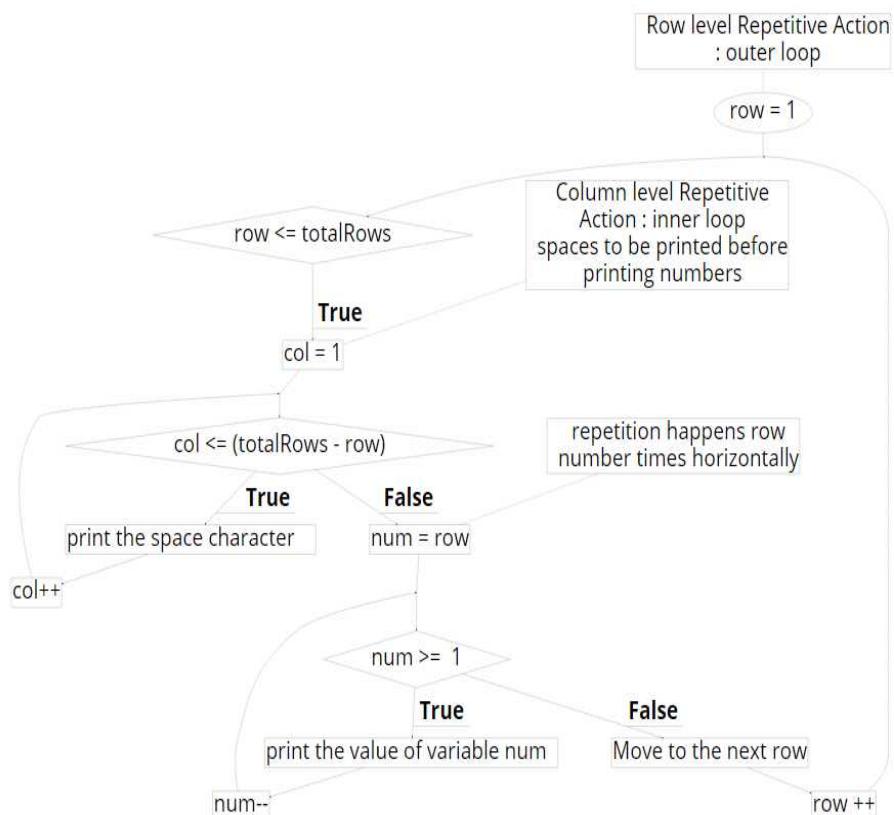
So our inner for-loop condition for printing space characters becomes;

loop-condition	<code>lower_limit <= col <= upper_limit</code>
Mathematical expression	<code>1 <= col <= (totalRows - row)</code>
syntax	<code>for col in range(1, totalRows - row + 1):</code>

So our inner for-loop condition for printing numeric characters becomes;

loop-condition	<code>lower_limit <= num <= upper_limit</code>
Mathematical expression	<code>1 <= num <= row</code>
syntax	<code>for num in range(row, 0, -1):</code>

The flowchart can be used to summarize all the gathered information. It is given as follows.



Let's execute the following program. Please, test for a different number of lines. Let it be 7.

NumberPatternBB.py

```
totalRows = 5 # number of rows to display

for row in range(1, totalRows + 1):

    # spaces to be printed before printing numbers
    for col in range(1, totalRows - row + 1):
        print("", end="")
        # Ends inner for-loop_1
    for num in range(row, 0,-1):
        # added extra space for clarity in output.
        print(num,"", end="")
        # Ends inner for-loop_2

    print()# moves control to the next line

# Ends outer for-loop
```

Output:totalRows = 5

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

Output:totalRows = 7

1					
2	1				
3	2	1			
4	3	2	1		
5	4	3	2	1	
6	5	4	3	2	1
7	6	5	4	3	2

Alternative solution

We observed previously that for row=5, the values of col and numeric output are just reversed. Let devise a solution by decrementing the for-loop for col variable (5 to 1), so the numeric output will match partially or wholly with the values of the col variable.

row	1				
col	5	4	3	2	1
output	space	space	space	space	1

row = 1 , 1>=col>=1 (print col value) or row >=col>=1 (print col value)

row	2				
col	5	4	3	2	1
output	space	space	space	2	1

row = 2 , 2>=col>=1 (print col value) or row >=col>=1 (print col value)

row	3				
col	5	4	3	2	1
output	space	space	3	2	1

row = 3 , 3>=col>=1 (print col value) or row >=col>=1 (print col value)

row	4				
col	5	4	3	2	1
output	space	4	3	2	1

row = 4 , 4>=col>=1 (print col value) or row >=col>=1 (print col value)

row	5				
col	5	4	3	2	1
output	5	4	3	2	1

row = 5 , $5 \geq col \geq 1$ (print col value) or $row \geq col \geq 1$ (print col value)

So, if-condition now becomes (**col ≥ 1 and col $\leq row$**).

$col \geq 1$ is redundant condition, since it's a lower limit in the inner for-loop for $col = 5$ to 1.

So, if-condition is reduced to (**col $\leq row$**)

if-condition	<code>lower_limit <= col <= upper_limit</code>
Mathematical expression	$1 \leq col \leq row$
syntax	<code>if(col <= row):</code>

The overall observation is at each row, either printing space characters or col values in a continuous fashion. It's quite obvious that we can use the if-else statement as the printing of characters is continuous. Let's update the program.

NumberPatternBB2.py

```
totalRows = 5 # number of rows to display
```

```
# Row level Repetitive Action :
for row in range(1, totalRows + 1):
# Column level Repetitive Action :
# To print numeric character column-wise, in the same row.
for col in range(totalRows, 0,-1):
    if (col <= row):
        print(col,"", end="")
    else:
        print(" ", end="")
# Ends inner for-loop

print()# moves control to the next line
# Ends outer for-loop
```

Output

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

NumberPattern CC

Our programming job is to print this particular logic-based pattern on one part of the computer screen. **Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes**—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique **(row, column)** numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

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

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1					1
Row=2				2	1
Row=3			3	2	1
Row=4		4	3	2	1
Row=5	5	4	3	2	1

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable **(row, column)** numeric paired values. **For example;** **(2, 3)** means printing numeric characters at the assumed screen location **2nd row and 3rd column**.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	1,1	1,2	1,3	1,4	1,5
Row=2	2,1	2,2	2,3	2,4	
Row=3	3,1	3,2	3,3		
Row=4	4,1	4,2			
Row=5	5,1				

Analysis of output

Let devise a solution by applying incrementing for-loop using a range of values of **variable: col(1 to 5)**. The numeric output may match partially or wholly with the values of the col variable. So **if-condition** can be expressed using col variable.

row	1				
col	1	2	3	4	5
output	1	2	3	4	5

row =1 , 1 <=Col <= 5 (print col value)

row	2				
col	1	2	3	4	5
output	1	2	3	4	

row = 2 ,(1 <=Col <= 4)(print col value)

row	3				
col	1	2	3	4	5
output	1	2	3		

row = 3 ,(1 <=Col <= 3)(print col value)

row	4				
col	1	2	3	4	5
output	1	2			

row = 4 ,(1 <=Col <= 2)(print col value)

row	5				
col	1	2	3	4	5
output	1				

row = 5 ,(1 <=Col <= 1)(print col value)

COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action			Expected Output
row	Column level Repetitive Action		If-condition(lower_limit <=col<= upper_limit)
	lower_limit	upper_limit	
1	1	5	1 2 3 4 5
2	1	4	1 2 3 4
3	1	3	1 2 3
4	1	2	1 2
5	1	1	1

The **lower_limit** has the same numeric value expression for the rows. It's consistent for all the rows. Likewise, we need to develop such a consistent expression for **upper_limit**, which should suit any number of rows. The considered variables are row, totalRows=5. Let's try to generalize the numeric sequence of the **upper_limit** column into a logical expression using variables.

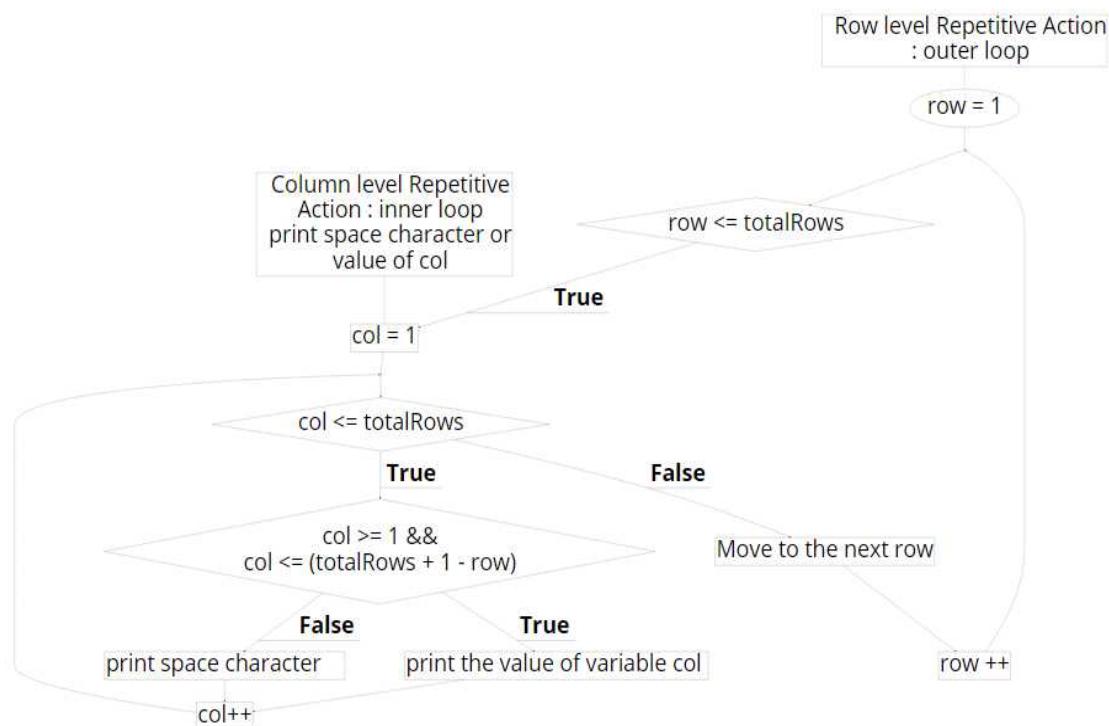
The lower_limit of col variable is generalized to value 1, since being consistent for all the row values.

Row level Repetitive Action						
row	Column level Repetitive Action					
	If-condition(lower_limit <=col<= upper_limit)					
1	1	lower_limit	upper_limit			
2		5	totalRows	(totalRows +1)- 1	(totalRows +1)- row	(totalRows+1)- row
3		4	totalRows -1	(totalRows +1)- 2	(totalRows +1)- row	
4		3	totalRows - 2	(totalRows +1)- 3	(totalRows +1)- row	
5		2	totalRows - 3	(totalRows +1)- 4	(totalRows +1)- row	
		1	totalRows - 4	(totalRows +1)- 5	(totalRows +1)- row	

So the upper_limit of col variable is generalized to (**totalRows + 1 – row**). Thus, lower_limit & upper_limit are consistent for all the rows. It's now generalized for any number of rows. So, our If-condition becomes;

if-condition	lower_limit <=col<= upper_limit
Mathematical expression	1 <= col <=((totalRows +1)- row)
syntax	if (col >= 1 and col <=(totalRows + 1 - row)):

We can summarize our findings in a flowchart given as follows.



Let's implement the program.

NumberPatternCC1.py

```
totalRows = 5 # number of rows to display

# Row level Repetitive Action :
for row in range(1, totalRows + 1):

    # Column level Repetitive Action :
    # print characters in the same row
    # spaces to be printed before printing numbers
    for col in range(1, totalRows + 1):

        if (col >= 1 and col <=(totalRows + 1 - row)):
            print(col, end="")
        else:
            print("", end="")

    # Ends inner for-loop

    print()# move control to the next line

# Ends outer for-loop
```

Output

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

The inner for-loop's *lower_limit* matches with the if-condition's lower_limit.

```
for col in range(1, totalRows + 1):

    if (col >= 1 and col <=(totalRows + 1 - row)):
```

We can neglect *lower_limit* in if-condition as it becomes a redundant condition because it is already been taken care in *lower_limit* of inner for-loop. Hence, our final if-condition can be given as;

if-condition	lower_limit <= col <= upper_limit
Mathematical expression	col <= ((totalRows + 1) - row)
syntax	if (col <= (totalRows + 1 - row)):

NumberPatternCC2.py

```
totalRows = 5 # number of rows to display

# Row level Repetitive Action :
for row in range(1, totalRows + 1):

    # Column level Repetitive Action :
    # print characters in the same row

    # spaces to be printed before printing numbers
    for col in range(1, totalRows + 1):

        if (col <= (totalRows + 1 - row)):
            print(col, end = "")
        else:
            print("", end = "")

    # Ends inner for-loop

    print()# moves control to the next line
# Ends outer for-loop
```

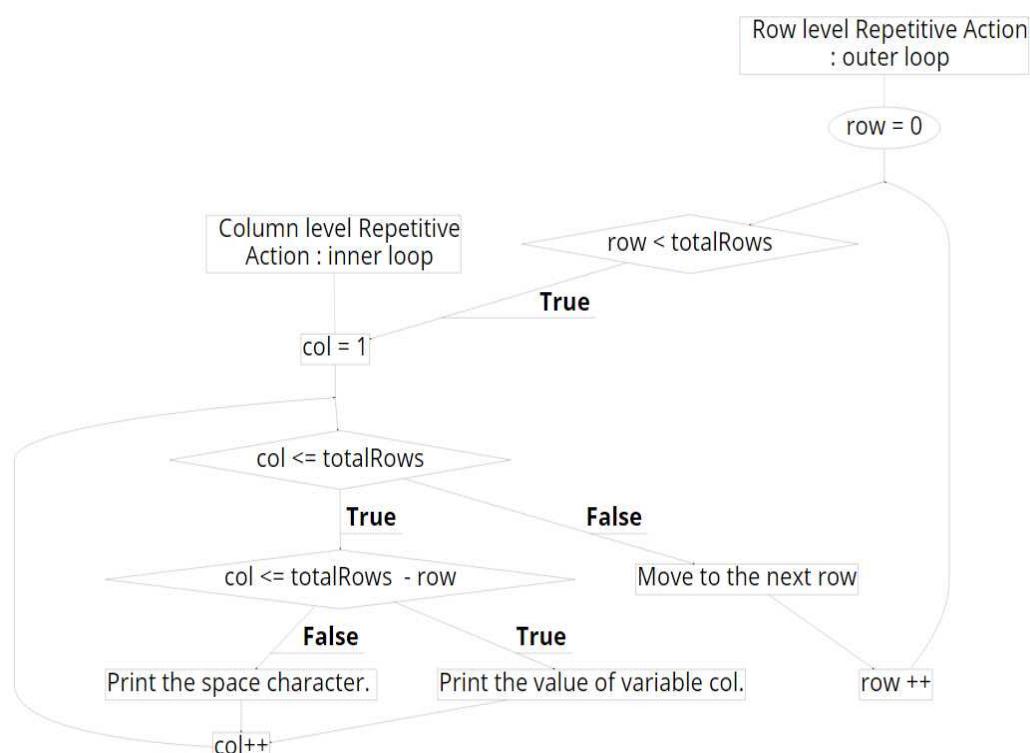
Output

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

Further, simplify by subtracting 1 from row values and upper_limit of col values by utilizing the previously obtained trace table.

Column level Repetitive Action					
row		col			
	1	lower_limit	upper_limit		
			1	2	
1 - 1	0	1	(totalRows +1)– row - 1	(totalRows – row)	
2 - 1	1	1	(totalRows +1)– row - 1	(totalRows – row)	
3 - 1	2	1	(totalRows +1)– row - 1	(totalRows – row)	
4 - 1	3	1	(totalRows +1)– row - 1	(totalRows – row)	
5 - 1	4	1	(totalRows +1)– row - 1	(totalRows – row)	

if-condition	lower_limit <= col <= upper_limit
Mathematical expression	col <= (totalRows – row)
syntax	if(col <= (totalRows - row)):



Let's update the program with our latest logical condition modification.

NumberPatternCC3.py

```
totalRows = 5 # number of rows to display

# Row level Repetitive Action :
for row in range(0, totalRows):

    # Column level Repetitive Action :
    # print characters in the same row

        # spaces to be printed before printing numbers
    for col in range(1, totalRows + 1):

        if (col <=(totalRows - row)):
            print(col, end ="")
        else:
            print("", end ="")

    #Ends inner for-loop

    print()# move control to the next line

#Ends outer for-loop
```

Output

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

NumberPattern DD

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique (row, column) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

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

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	5	4	3	2	1
Row=2		4	3	2	1
Row=3			3	2	1
Row=4				2	1
Row=5					1

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (row, column) numeric paired values. For example; (4, 5) means printing numeric characters at the assumed screen location 4th row and 5th column.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	1,1	1,2	1,3	1,4	1,5
Row=2		2,2	2,3	2,4	2,5
Row=3			3,3	3,4	3,5
Row=4				4,4	4,5
Row=5					5,5

Analysis of output

Let's apply decrementing for-loop for col variable (5 to 1) or ($5 \geq \text{col} \geq 1$) so that the numeric output will match partially or wholly with the values of the **variable:col**.

row	1				
col	5	4	3	2	1
output	5	4	3	2	1

row =1 , $5 \geq \text{col} \geq 1$ (print col value)

row	2				
col	5	4	3	2	1
output		4	3	2	1

row =2 , $4 \geq \text{col} \geq 1$ (print col value)

row	3				
col	5	4	3	2	1
output			3	2	1

row =3 , $3 \geq \text{col} \geq 1$ (print col value)

row	4				
col	5	4	3	2	1
output				2	1

row =4 , $2 \geq \text{col} \geq 1$ (print col value)

row	5				
col	5	4	3	2	1
output					1

row =5 , $1 \geq \text{Col} \geq 1$ (print col value)

COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action			Expected Output	
row	Column level Repetitive Action			
	upper_limit $\geq \text{col} \geq$ lower_limit			
	lower_limit		upper_limit	
1	1		5	
2	1		4	
3	1		3	
4	1		2	
5	1		1	

The lower_limit has the same consistent expression as the numeric value for all the rows.

Likewise, we need to develop such a consistent expression for upper_limit, which should suit any number of rows. The available variables are row & totalRows = 5. Let's try to generalize the numeric sequences of the upper_limit column into a valid logical expression using variables.

Column level Repetitive Action

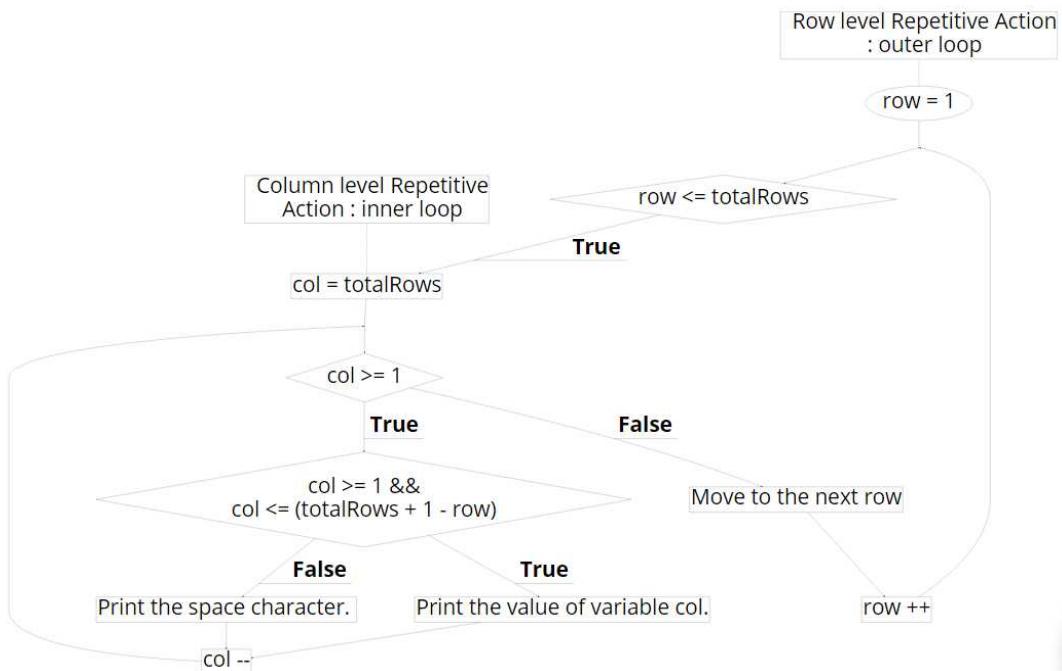
row	If-condition(upper_limit >= col >= lower_limit)				
	lower_limit	upper_limit			
		1	2	3	
1	1	5	totalRows	totalRows + 1 - 1	totalRows + 1 - row
2	1	4	totalRows - 1	totalRows + 1 - 2	totalRows + 1 - row
3	1	3	totalRows - 2	totalRows + 1 - 3	totalRows + 1 - row
4	1	2	totalRows - 3	totalRows + 1 - 4	totalRows + 1 - row
5	1	1	totalRows - 4	totalRows + 1 - 5	totalRows + 1 - row

The lower_limit & upper_limit are consistent for all the rows. The upper_limit is generalized to (**totalRows + 1 - row**). This logical expression is such a formula, commonly applied on all rows and columns, which helps to give matching output.

So our If-condition becomes

if-condition	upper_limit >= col >= lower_limit
Mathematical expression	$1 \leq col \leq (\text{totalRows} + 1) - \text{row}$
syntax	<code>if(col >= 1 and col <= (totalRows + 1 - row)):</code>

The flowchart can be given as follows.



NumberPatternDD1.py

```
totalRows = 5 # number of rows to display

# Row level Repetitive Action:
for row in range(1, totalRows + 1):

    # Column level Repetitive Action:
    # print numeric or space character in the same row
    # spaces are to be printed before printing numbers
    for col in range(totalRows, 1 - 1,-1):

        if (col >= 1 and col <=(totalRows + 1 - row)):
            print(col, end="")
        else:
            print("", end="")

    # Ends inner for-loop

    print()# Moves control to the next line

# Ends outer for-loop
```

Output

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

The decrementing inner for-loop's lower_limit matches with the if-condition's lower_limit.

```
for col in range(totalRows, 1 - 1,-1):

    if (col >= 1 and col <=(totalRows + 1 - row)):
```

We can safely neglect logical expression for **lower_limit** in if-condition as it is already taken care of by **lower_limit** of the related for-loop. Hence, our final **if-condition** will be;

if-condition	upper_limit >= col >= lower_limit
Mathematical expression	col <= ((totalRows + 1) - row)
syntax	if(col <= (totalRows + 1 - row)):

NumberPatternDD2.py

```
totalRows = 5 # number of rows to display

# Row level Repetitive Action :
for row in range(1, totalRows + 1):

    # Column level Repetitive Action:
    # print numeric or space character in the same row
    # spaces are to be printed before printing numbers
    for col in range(totalRows, 0,-1):

        if (col <=(totalRows + 1 - row)):
            print(col, end ="")
        else:
            print("", end ="")

    # Ends inner for-loop

    print()# moves control to the next line
# Ends outer for-loop
```

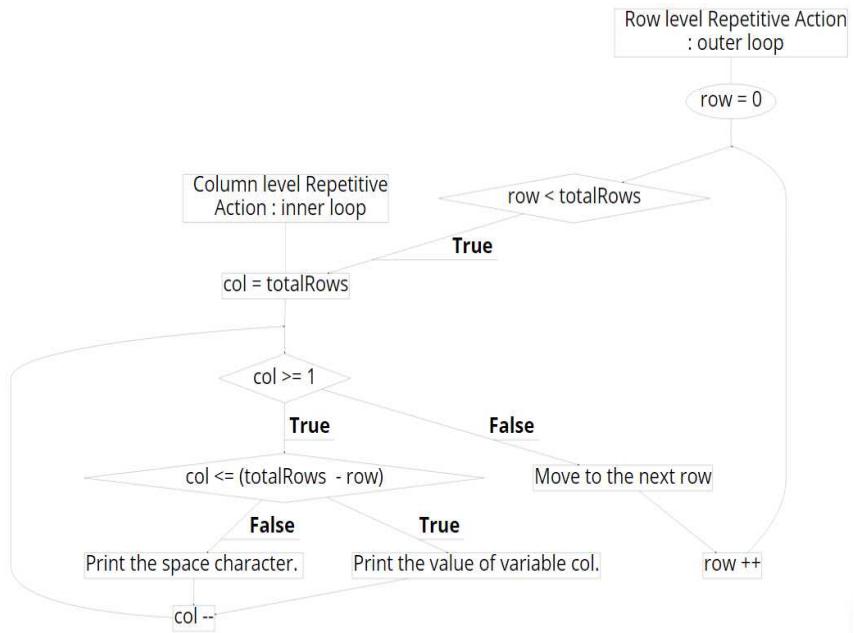
Further, simplifying upper_limit by subtracting 1 as shown in the following

Column level Repetitive Action				
row		upper_limit >=col>= lower_limit		
		lower_limit	upper_limit	
1 - 1	0	1	(totalRows +1)- row - 1	(totalRows – row)
2 - 1	1	1	(totalRows +1)- row - 1	(totalRows – row)
3 - 1	2	1	(totalRows +1)- row - 1	(totalRows – row)
4 - 1	3	1	(totalRows +1)- row - 1	(totalRows – row)
5 - 1	4	1	(totalRows +1)- row - 1	(totalRows – row)

Hence, our improved if-condition will be;

if-condition	upper_limit >=col>= lower_limit
Mathematical expression	col <=((totalRows - row)
syntax	if(col <=(totalRows - row)):

This modification can be easily expressed in the flowchart.



The updated final solution can be expressed in the following program.

```

NumberPatternDD3.py
totalRows = 5 # number of rows to display

# Row level Repetitive Action :
for row in range(0, totalRows):

    # Column level Repetitive Action:
    # print numeric or space character in the same row
    # spaces are to be printed before printing numbers
    for col in range(totalRows, 0,-1):

        if (col <=(totalRows - row)):
            print(col,end ="")
        else:
            print("", end ="")

        # move control to the next line
    print()

```

Output

```

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

```

NumberPattern EE

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will be to assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes —the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique **(row, column)** numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1					1				
Row=2				2	1	2			
Row=3			3	2	1	2	3		
Row=4		4	3	2	1	2	3	4	
Row=5	5	4	3	2	1	2	3	4	5

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable **(row, column)** numeric paired values. **For example;** **(4, 3)** means printing numeric characters at the assumed screen location **4th row and 3rd column**.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1					1 , 5				
Row=2				2 , 4	2 , 5	2 , 6			
Row=3			3 , 3	3 , 4	3 , 5	3 , 6	3 , 7		
Row=4		4 , 2	4 , 3	4 , 4	4 , 5	4 , 6	4 , 7	4 , 8	
Row=5	5 , 1	5 , 2	5 , 3	5 , 4	5 , 5	5 , 6	5 , 7	5 , 8	5 , 9

Analysis of output

row	1								
col	1	2	3	4	5	6	7	8	9
output					1				

For row = 1, $5 \leq \text{Col} \leq 5$

row	2								
col	1	2	3	4	5	6	7	8	9
output				2	1	2			

For row = 2, $4 \leq \text{Col} \leq 6$

row	3								
col	1	2	3	4	5	6	7	8	9
output			3	2	1	2	3		

For row = 3, $3 \leq \text{Col} \leq 7$

row	1								
col	1	2	3	4	5	6	7	8	9
output			4	3	2	1	2	3	4

For row = 4, $2 \leq \text{Col} \leq 8$

row	1								
col	1	3	4	5	6	7	8	9	
output	5	3	2	1	2	3	4	5	

For row = 5, $1 \leq \text{Col} \leq 9$

COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action				Expected Output	
row	Column level Repetitive Action				
	lower_limit <= col <= upper_limit				
	range		lower_limit	upper_limit	
row = 1	$5 \leq \text{Col} \leq 5$		5	5	1
row = 2	$4 \leq \text{Col} \leq 6$		4	6	2 1 2
row = 3	$3 \leq \text{Col} \leq 7$		3	7	3 2 1 2 3
row = 4	$2 \leq \text{Col} \leq 8$		2	8	4 3 2 1 2 3 4
row = 5	$1 \leq \text{Col} \leq 9$		1	9	5 4 3 2 1 2 3 4 5

Both the lower_limit & upper_limit are not consistent for all the rows. We need to develop consistent expressions for both upper_limit and lower_limit, which should suit any number of rows. The other remaining variables are **row**, **totalRows = 5**. Let's try to generalize the numeric sequences of the lower_limit & upper_limit columns into a valid logical expression using variables.

Column level Repetitive Action							
row	If-condition(lower_limit <= col <= upper_limit)						
	lower_limit			upper_limit			
	1	2		1	2		
1	5	totalRows	totalRows +1 - 1	5	totalRows	totalRows +1 - 1	
2	4	totalRows -1	totalRows +1 - 2	6	totalRows + 1	totalRows + 2 - 1	
3	3	totalRows -2	totalRows +1 - 3	7	totalRows + 2	totalRows + 3 - 1	
4	2	totalRows -3	totalRows +1 - 4	8	totalRows + 3	totalRows + 4 - 1	
5	1	totalRows -4	totalRows +1 - 5	9	totalRows + 4	totalRows + 5 - 1	

Column level Repetitive Action							
row	lower_limit <= col <= upper_limit						
	lower_limit			upper_limit			
	default	3		default	3		
1	5	totalRows +1 - row		5	totalRows + row - 1		
2	4	totalRows +1 - row		6	totalRows + row - 1		
3	3	totalRows +1 - row		7	totalRows + row - 1		
4	2	totalRows +1 - row		8	totalRows + row - 1		
5	1	totalRows +1 - row		9	totalRows + row - 1		

The lower_limit & upper_limit have become consistent for all the rows.

The upper_limit is generalized to (**totalRows + row - 1**). This logical expression is such a formula, commonly applied on all rows and columns, which helps to give matching output.

So our If-condition becomes;

if-condition	lower_limit <= col <= upper_limit
Mathematical expression	(totalRows +1 - row) <= col <= (totalRows + row - 1)
syntax	<code>if(col >=((totalRows - row)+ 1) and col <=((totalRows + row)- 1)):</code>

When the if-condition is satisfied, it will print the numeric character at that position at a particular row or else will print the space character. We can observe two things from the given output, which can be given as;

a) The values at each row start with the row value and goes on decreasing till the value 1 and further increase till it reaches the value of row value. The value is basically oscillating like a pendulum of a wall clock. Let's choose a variable **num** that represents a row value every time a new row starts. The value of variable **num** then oscillates to give output in a single row. It can be shown by the following table.

Table for num variable starting at row value.

row	num	col < totalRows	col = totalRows	col > totalRows
1	1		1	
2	2	2	1	2
3	3	3 2	1	2 3
4	4	4 3 2	1	2 3 4
5	5	5 4 3 2	1	2 3 4 5

If(col < totalRows):
 num value gets printed.
 num value get decreased by 1.

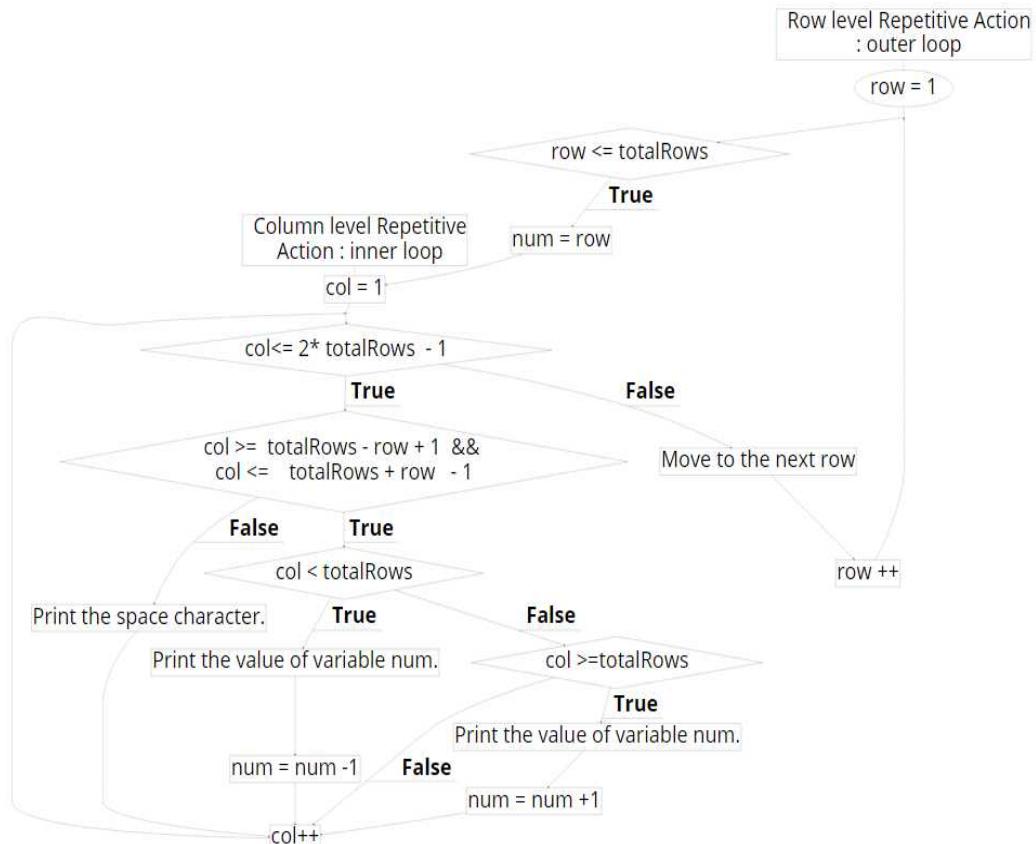
If(col >= totalRows):
 num value gets printed.
 num value get increased by 1.

- b) The column counts are more than the **totalRows** count other than the first row, and the input value of the variable **totalRows** is 5. The column count is different on each row. Considering all these factors, the upper_limit of the inner for-loop need to be generalized.

loop-condition	<code>lower_limit <= col <= upper_limit</code>
Mathematical expression	<code>1 <= col <= 9</code>
	<code>1 <= col <= totalRows + 4</code>
	<code>1 <= col <= totalRows +(totalRows - 1)</code>
	<code>1 <= col <= 2 * totalRows - 1</code>
syntax	<code>for col in range(1, 2 * totalRows):</code>

So this generalization to calculate total columns for a given row value is satisfactory.

The flow-chart can be given as;



NumberPatternEE.py

```
totalRows = 8 # number of rows to display

    # Row level Repetitive Action :
for row in range(1, totalRows + 1):

    # Total repetition column-wise is equals to (2* totalRows - 1).
num = row
    # Column level Repetitive Action:
    # print numeric or space character in the same row
    # spaces are to be printed before printing numbers
for col in range(1, 2 * totalRows):

    if (col >=((totalRows - row)+ 1) and col <=((totalRows + row)- 1)):

        if (col < totalRows):
            # 1st half triangle
print(num, end="")
num = num - 1
    elif (col >= totalRows):
        # 2nd half traingle
print(num, end="")
num = num + 1

    else:
print("", end="")
# end inner for - loop

print()# move control to the next line

    # end outer for-loop
```

Output

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

ProblemGBEE: Write a logic trace table and a program for the following output.

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

NumberPattern FF

Our programming job is to print this particular logic-based pattern on one part of the computer screen. **Our approach will be to assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes** —the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique **(row, column)** numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1	1	2	3	4	5	4	3	2	1
Row=2	1	2	3	4		4	3	2	1
Row=3	1	2	3				3	2	1
Row=4	1	2						2	1
Row=5	1								1

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable **(row, column)** numeric paired values. **For example;** **(2, 3)** means printing character at the assumed screen location 2nd row and 3rd column.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1	(1,1) 1	(1,2) 2	(1,3) 3	(1,4) 4	(1,5) 5	(1,6) 4	(1,7) 3	(1,8) 2	(1,9) 1
Row=2	(2,1) 1	(2,2) 2	(2,3) 3	(2,4) 4	2,5 space	(2,6) 4	(2,7) 3	(2,8) 2	(2,9) 1
Row=3	(3,1) 1	(3,2) 2	(3,3) 3	3,4 space	3,5 space	3,6 space	(3,7) 3	(3,8) 2	(3,9) 1
Row=4	(4,1) 1	(4,2) 2	4,3 space	4,4 space	4,5 space	4,6 space	4,7 space	(4,8) 2	(4,9) 1
Row=5	(5,1) 1	5,2 space	5,3 space	5,4 space	5,5 space	5,6 space	5,7 space	5,8 space	(5,9) 1

Analysis of output

In general, three repetitive actions take place.

- Printing of numbers starting from 1 in incrementing order.
- Printing of spaces.
- Printing numbers in decrementing order and its upper limit will start from the last updated incremented value.

COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action			
row	Column level Repetitive Action		
	Variable :numInc	Variable :space (space counts)	Variable :numDec
1	1, 2 ,3 ,4, 5	0(no space)	4, 3, 2, 1
2	1, 2, 3, 4	1	4, 3, 2, 1
3	1, 2, 3	1,2,3	3, 2, 1
4	1, 2	1,2,3,4,5	2, 1
5	1	1,2,3,4,5,6,7	1

Row level Repetitive Action						
row	Column level Repetitive Action					
	Variable :numInc		Variable :space		Variable :numDec	
	lower_limit	upper_limit	lower_limit	upper_limit	upper_limit	lower_limit
1	1	5	0	0	4	1
2	1	4	1	1	4	1
3	1	3	1	3	3	1
4	1	2	1	5	2	1
5	1	1	1	7	1	1

We know that the variable **totalRows=5** gives the total number of rows. We need to generalize to get a unique expression. The **upper_limit** of variable **numInc** and **numDec** can be expressed in terms of variable **totalRows**. The **lower_limit** of variable **numInc** and **numDec** is already generalized.

Row level Repetitive Action						
row	Column level Repetitive Action					
	Variable :numInc		Variable :space		Variable :numDec	
	lower_limit	upper_limit	lower_limit	upper_limit	upper_limit	lower_limit
1	1	1	totalRows - 0	0	0	totalRows - 1
2		1	totalRows - 1	1	1	totalRows - 1
3		1	totalRows - 2	1	3	totalRows - 2
4		1	totalRows - 3	1	5	totalRows - 3
5		1	totalRows - 4	1	7	totalRows - 4

If we take the row variable values starting from 0 instead of 1, it will help us generalize the **upper_limit** of variable **numInc** and **numDec**. Let's try to generalize both of these using **variable: row**.

Row level Repetitive Action						
row	Column level Repetitive Action					
	Variable :numInc		Variable :space		Variable :numDec	
	lower_limit	upper_limit	lower_limit	upper_limit	upper_limit	lower_limit
		1			1	
row = 0	1	totalRows - 0	0	0	totalRows - 1	
row = 1		totalRows - 1	1	1	totalRows - 1	
row = 2		totalRows - 2	1	3	totalRows - 2	1
row = 3		totalRows - 3	1	5	totalRows - 3	
row = 4		totalRows - 4	1	7	totalRows - 4	

The part of `upper_limit` of variable `numInc` and `numDec` matches with the numeric pattern of row variable. Hence, we can use the row variable to generalize the expressions.

Row level Repetitive Action						
row	Column level Repetitive Action					
	Variable :numInc		Variable :space		Variable :numDec	
	lower_limit	upper_limit	lower_limit	upper_limit	upper_limit	lower_limit
		1			1	
row = 0	1	totalRows - row	0	0	totalRows - 1	
row = 1		totalRows - row	1	1	totalRows - row	
row = 2		totalRows - row	1	3	totalRows - row	1
row = 3		totalRows - row	1	5	totalRows - row	
row = 4		totalRows - row	1	7	totalRows - row	

For the `upper_limit` of variable `numDec`, we got two expressions. Somehow by putting a condition, we need to switch the expression depending upon the value of the row variable. Let's take one temporary variable `ul_numDec` to represent `upper_limit` of the `variable:numDec`; then it can be given as;

```
Variable: ul_numDec
if(row == 0 )
ul_numDec is ( totalRows - 1)
else
ul_numDec is (totalRows - row).
```

Row level Repetitive Action						
row	Column level Repetitive Action					
	Variable :numInc		Variable :space		Variable :numDec	
	lower_limit	upper_limit	lower_limit	upper_limit	upper_limit	lower_limit
		1			1	
row = 0	1	totalRows - row	0	0	ul_numDec	
row = 1			1	1	ul_numDec	
row = 2			1	3	ul_numDec	
row = 3			1	5	ul_numDec	
row = 4			1	7	ul_numDec	

Until now, we can generalize variables `numInc` and `numDec`, since their `lower_limit` and `upper_limit` are showing the same expression for all the row values.

What remained now is to generalize the `upper_limit` for the variable `spaces`.

Column level Repetitive Action								
row	Variable: space							
	lower_limit		upper_limit					
0	0	0	1	2	3	4	5	
1	1	1	1	1	1 + 1 - 1	2 * 1 - 1	2 * row - 1	
2	1	3	2 + 1	2 + 2 - 1	2 + 2 - 1	2 * 2 - 1	2 * row - 1	
3	1	5	3 + 2	3 + 3 - 1	3 + 3 - 1	2 * 3 - 1	2 * row - 1	
4	1	7	4 + 3	4 + 4 - 1	4 + 4 - 1	2 * 4 - 1	2 * row - 1	

For the upper_limit of variable **space** at row = 0, the value will be $(2 * 0 - 1)$ or -1. The lower_limit value of variable **space** at row = 0 is also 0, so the loop-condition $(0 < -1)$ will fail and no space will get printed. This holds True even for **loop-condition** $(1 < -1)$. Hence, we are in a position to get a generalized expression for lower_limit and upper_limit.

loop-condition	lower_limit <= space <= upper_limit
Mathematical expression	$1 \leq space \leq (2 * row - 1)$
syntax	for(space = 1 space <= (2 * row - 1) space++)

This time somehow, we are able to generalize the logical expressions; otherwise, we need to follow the **if-else condition** as we did for the variable **numDec** using temporary variable **ul_numDec**.

Alternate solution

Row level Repetitive Action				
row	Column level Repetitive Action			
	Variable: col	Variable: col	Variable: col	Prnumbers
1	1, 2, 3, 4	Pr5(no space)	6, 7, 8, 9	4, 3, 2, 1
2	1, 2, 3, 4	5	6, 7, 8, 9	4, 3, 2, 1
3	1, 2, 3	4, 5, 6	7, 8, 9	3, 2, 1
4	1, 2	3, 4, 5, 6, 7	8, 9	2, 1
5	1	2, 3, 4, 5, 6, 7, 8	9	1

Row level Repetitive Action							
row	Column level Repetitive Action						
	Variable: col		Variable: col		Variable: col		
	Prnumbers		Print spaces		Prnumbers		
	lower_limit	upper_limit	lower_limit	upper_limit	lower_limit	upper_limit	Output
1	1	4	Pr5(no space)		6	9	4, 3, 2, 1
2	1	4	5	5	6	9	4, 3, 2, 1
3	1	3	4	6	7	9	3, 2, 1
4	1	2	3	7	8	9	2, 1
5	1	1	2	8	9	9	1

For printing characters at the first row, means at row=1, this pattern doesn't require printing any space. So for the sake of initialization, we can give its upper_limit value as -1 and lower_limit value as 0. If we observe for printing spaces, the numeric pattern is that the lower_limit value is less than the upper_limit value for every row except the first row. This will help to allow condition to fail, and we will not be able to print space in the first row. This what we need.

Moreover, it's common practice to initialize a variable with invalid data to verify the programming logic's validity whenever that programming logic gets executed. We do exercises on programming patterns so that we can visualize the flow of data based on the values of variables. These variables actually act as check-points on whether the logic executed as expected, and this was our purpose of running exercises on programming patterns.

Row level Repetitive Action								
row	Column level Repetitive Action							
	Variable: col		Variable: col		Variable: col			
Prnumbers		Print spaces		Prnumbers				
	lower_limit	upper_limit	lower_limit	upper_limit	lower_limit	upper_limit	Output	
1	1	totalRows - 1	0	-1	6	9	4, 3, 2, 1	
2	1	totalRows - 1	totalRows - 0	totalRows + 0	6	9	4, 3, 2, 1	
3	1	totalRows - 2	totalRows - 1	totalRows + 1	7	9	3, 2, 1	
4	1	totalRows - 3	totalRows - 2	totalRows + 2	8	9	2, 1	
5	1	totalRows - 4	totalRows - 3	totalRows + 3	9	9	1	

If we take values of row variables starting from 0 instead of 1, it will help us generalize the **upper_limit**. Let's try to generalize both of these using variables. If we observe, we can try to generalize lower_limit & upper_limit of printing spaces column using **row** variable. Using simple math, we can alter lower_limit & upper_limit so that part of the expression that follows a numeric pattern may match the row variable's numeric pattern.

Row level Repetitive Action								
row	Column level Repetitive Action							
	Variable: col		Variable: col		Variable: col			
Prnumbers		Print spaces		Prnumbers				
	Lower_limit	Upper_limit	Lower_limit	Upper_limit	lower_limit	Upper_limit	Output	
0	1	totalRows - 1	0	-1	6	9	4, 3, 2, 1	
1	1	totalRows - 1	totalRows - 0	totalRows + 0	6	9	4, 3, 2, 1	
2	1	totalRows - 2	totalRows - 1	totalRows + 1	7	9	3, 2, 1	
3	1	totalRows - 3	totalRows - 2	totalRows + 2	8	9	2, 1	
4	1	totalRows - 4	totalRows - 3	totalRows + 3	9	9	1	

Row level Repetitive Action								
row	Column level Repetitive Action							
	Variable: col		Variable: col		Variable: col			
Prnumbers		Print spaces		Prnumbers				
	lower_limit	upper_limit	lower_limit	upper_limit	lower_limit	upper_limit	Output	
0	1	totalRows - 1	0	-1	6	9	4, 3, 2, 1	
1	1	totalRows - row	totalRows - 1 + 1	totalRows + 1 - 1	6	9	4, 3, 2, 1	
2	1	totalRows - row	totalRows - 2 + 1	totalRows + 2 - 1	7	9	3, 2, 1	
3	1	totalRows - row	totalRows - 3 + 1	totalRows + 3 - 1	8	9	2, 1	
4	1	totalRows - row	totalRows - 4 + 1	totalRows + 4 - 1	9	9	1	

Row level Repetitive Action								
row	Column level Repetitive Action							
	Variable: col		Variable: col			Variable: col		
	Prnumbers		Print spaces			Prnumbers		
	lower_limit	upper_limit	lower_limit	upper_limit	lower_limit	upper_limit	Output	
0	1	totalRows - 1	0	-1	6	9	4, 3, 2, 1	
1	1	totalRows - row	totalRows - row + 1	totalRows + row - 1	6	9	4, 3, 2, 1	
2	1	totalRows - row	totalRows - row + 1	totalRows + row - 1	7	9	3, 2, 1	
3	1	totalRows - row	totalRows - row + 1	totalRows + row - 1	8	9	2, 1	
4	1	totalRows - row	totalRows - row + 1	totalRows + row - 1	9	9	1	

If we build our programming logic giving conditions around a set1 of printing numbers and printing spaces, the last else-part will take for set2 of printing numbers. We need to switch the expression of lower_limit or upper_limit by putting a condition depending upon the value of the row variable.

Prnumbers(set 1)

```
lower_limit variable : lw_lt_num
if(row == 0)
    lw_lt_num =( totalRows - 1 )
else
    lw_lt_num =( totalRows - row )
```

Print spaces

```
lower_limit variable : lw_lt_spaces
if(row == 0)
    lw_lt_spaces = 0
else
    lw_lt_spaces =( totalRows - row + 1 )
```

Print spaces

```
upper_limit variable : up_lt_spaces
if(row == 0)
    up_lt_spaces =-1
else
    up_lt_spaces =( totalRows + row - 1 )
```

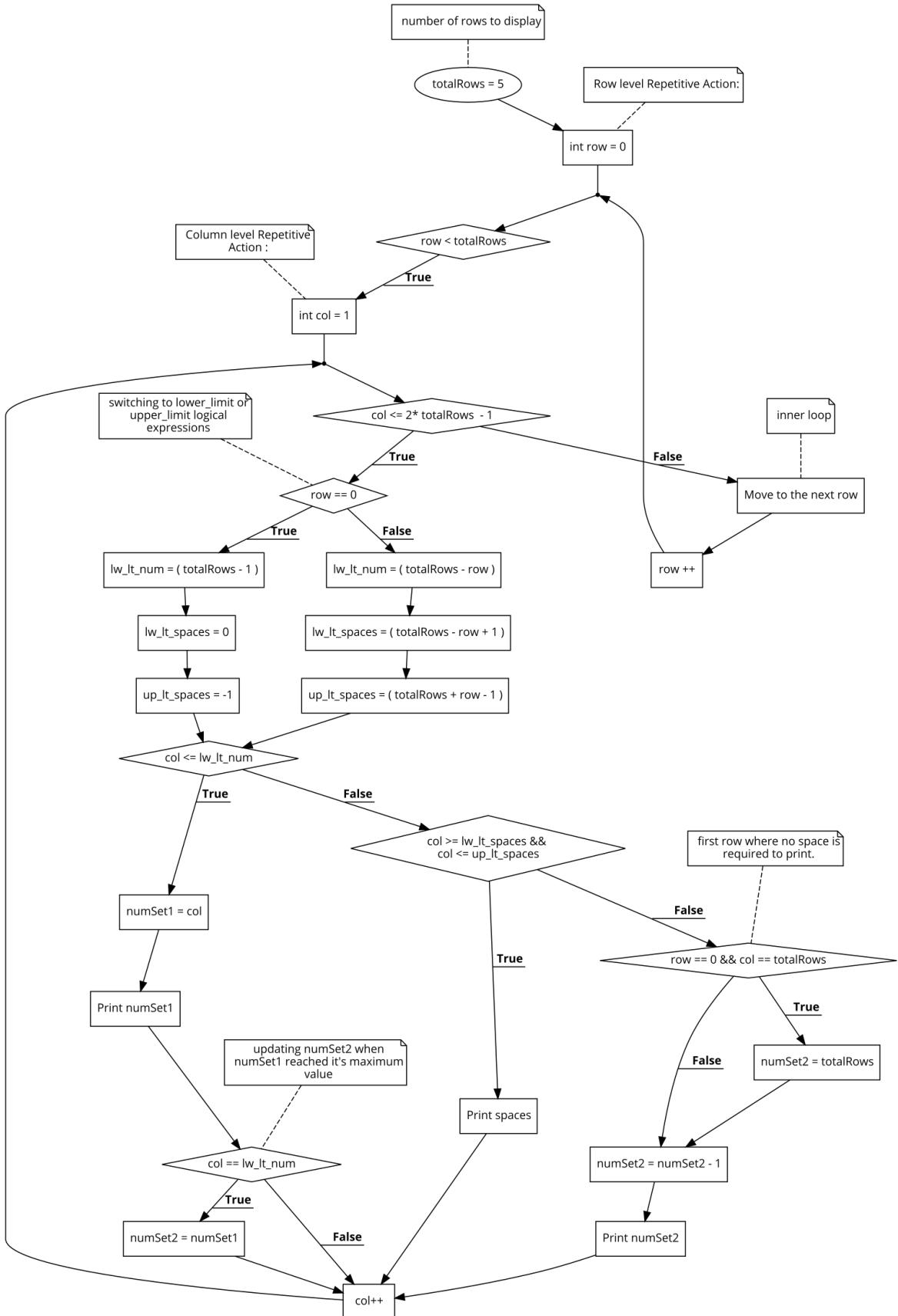
We know that the total number of rows is given by the **variable: totalRows = 5**.

The **variable: col** takes the range of values from 1 to 9.

Col = 1, 2, 3... 9 **or**
 Col = 1, 2, 3...(10 -1) **or**
 Col = 1, 2, 3...,(2*5 -1) **or**
 Col = 1, 2, 3...,(2* **totalRows** -1)

Hence, we expressed the **upper_limit** of **variable: col** as $(2 * \text{totalRows} - 1)$ in a generalized form.

We can summarize all our observations in a flowchart given as follows.



NumberPatternFF.py

```
totalRows = 5 # number of rows to display
numSet2 = 0

# Row level Repetitive Action :
for row in range(0, totalRows):
    # Column level Repetitive Action :
    for col in range(1, 2 * totalRows):
        # switching to lower_limit or upper_limit
        # logical expressions

    if (row == 0):
        lw_lt_num =(totalRows - 1)
        lw_lt_spaces = 0
        up_lt_spaces =-1
    else:
        lw_lt_num =(totalRows - row)
        lw_lt_spaces =(totalRows - row + 1)
        up_lt_spaces =(totalRows + row - 1)

    if (col <= lw_lt_num):
        numSet1 = col
        print(numSet1, end ="")
            # updating numSet2 when numSet1 reached
            # it's maximum value
    if (col == lw_lt_num):
        numSet2 = numSet1

    elif (col >= lw_lt_spaces and col <= up_lt_spaces):
        print("", end ="")# 2 spaces
    else:
        # first row where no space is required to print.

    if (row == 0 and col == totalRows):
        numSet2 = totalRows

    print(numSet2, end ="")
    numSet2 = numSet2 - 1

# Ends inner for-loop

print()# Moves control to the next line
```

Ends outer for-loop

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

If-else Shortcut

Short-cut in any matter is a special case to be applied in a specific situation. In programming, people unknowingly confuse themselves simply because they choose a special case and try to apply it commonly. Even in the choice of words, two words may give almost similar meanings, and when we actually apply those in different situations, then only we will be in a position to explore its true potential meaning.

The shortcut to the if-else option is in almost every programming language known as the ternary operator and given by the symbol (:).

When to apply if-else shortcut

The ternary operator is a sort of syntactic sugar for if-else statements usually applied in a situation where both **if-block** and **else-block** assign a different value to the same result variable. For example, let's suppose that the if-else statement is something like the following:

If-else statement	If-else shortcut or ternary operator (:)
<pre>if(condition): result = one_value else: result = other_value</pre>	<pre>result = one_value if (condition) else other_value</pre>
Simple if-else can be easily replaced by the use of if-else shortcut represented by the one liner ternary operator . In such situations only it makes True sense to use ternary operator to increase code readability.	<ul style="list-style-type: none">Two outcome result values are possible which are separated by a symbol colon (:).If condition is <u>True</u>, then <i>one_value</i> will get assigned to the variable <i>result</i>.If condition is <u>False</u>, then <i>one_value</i> will get assigned to the variable <i>result</i>.

In the program **NumberPatternFF.py**, we can use the ternary operator to improve the part of source code readability.

Separate if-else	Many If-else combined in single if-else
<pre>if(row == 0): lw_lt_num =(totalRows - 1) else: lw_lt_num =(totalRows - row) if(row == 0): lw_lt_spaces = 0 else: lw_lt_spaces =(totalRows - row + 1) if(row == 0): up_lt_spaces =-1 else: up_lt_spaces =(totalRows + row - 1)</pre>	<pre>if(row == 0): lw_lt_num =(totalRows - 1) lw_lt_spaces = 0 up_lt_spaces =-1 else: lw_lt_num =(totalRows - row) lw_lt_spaces =(totalRows - row + 1) up_lt_spaces =(totalRows + row - 1)</pre>
Equivalent If-else shortcut or ternary operator (:) expressions	

```

lw_lt_num =(totalRows - 1) if (row == 0) else (totalRows - row)
lw_lt_spaces = 0 if (row == 0) else (totalRows - row + 1)
up_lt_spaces ==-1 if (row == 0) else (totalRows + row - 1)

```

I am not really a great fan of exceptions, but just to share information, even **nested ternary operator is possible** like nested if-else. In most cases, it is not an easy read regarding the evaluation of logical expressions. In some cases, proper formatting of nested ternary operators may help but still implemented code will not be quick to grasp for most junior-level programmers. As per my opinion, please avoid nested ternary operators. The updated program is demonstrating the use of the ternary operator (:) to gain readability regarding that coding part. Please notice code changes.

NumberPatternFF2.py

```

totalRows = 5 # number of rows to display
numSet2 = 0
# Row level Repetitive Action :
for row in range(0, totalRows):
    # Column level Repetitive Action :
    for col in range(1, 2 * totalRows):
        # use of ternary operator, a simpler if-else shortcut
        lw_lt_num =(totalRows - 1) if (row == 0) else (totalRows - row)
        lw_lt_spaces = 0 if (row == 0) else (totalRows - row + 1)
        up_lt_spaces ==-1 if (row == 0) else (totalRows + row - 1)
        if (col <= lw_lt_num):
            numSet1 = col
            print(numSet1, end="")
                # updating numSet2 when numSet1 reached
                # it's maximum value
        if (col == lw_lt_num):
            numSet2 = numSet1
        elif (col >= lw_lt_spaces and col <= up_lt_spaces):
            print("", end="# 2 spaces"
        else:
            # first row : no space required to print.
        if (row == 0 and col == totalRows):
            numSet2 = totalRows
            print(numSet2, end="")
            numSet2 = numSet2 - 1
# Ends inner for-loop
print()# Moves control to the next line
# Ends outer for-loop

```

NumberPattern GG

Our approach will be to assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique **(row, column)** numeric combination. With this assumption, the computer screen will simply be a unique combination of **(row, column)** numeric value paired boxes that are uniformly spread all across the screen.

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

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1	A	B	C	D	E	D	C	B	A
Row=2	A	B	C	D		D	C	B	A
Row=3	A	B	C				C	B	A
Row=4	A	B					B	A	
Row=5	A								A

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable **(row, column)** numeric paired values. **For example;** **(4, 3)** means printing character at the assumed screen location **4th row and 3rd column**.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1	(1,1) A	(1,2) B	(1,3) C	(1,4) D	(1,5) E	(1,6) D	(1,7) C	(1,8) B	(1,9) A
Row=2	(2,1) A	(2,2) B	(2,3) C	(2,4) D	2,5 space	(2,6) D	(2,7) C	(2,8) B	(2,9) A
Row=3	(3,1) A	(3,2) B	(3,3) C	3,4 space	3,5 space	3,6 space	(3,7) C	(3,8) B	(3,9) A
Row=4	(4,1) A	(4,2) B	4,3 space	4,4 space	4,5 space	4,6 space	4,7 space	(4,8) B	(4,9) A
Row=5	(5,1) A	5,2 space	5,3 space	5,4 space	5,5 space	5,6 space	5,7 space	5,8 space	(5,9) A

This pattern is a special case of **NumberPatternFF**, which prints alphabets instead of numbers.

Analysis of output

In general, three repetitive actions take place.

a) Printing of alphabets starting from A in incrementing order till the mid column.

b) Printing of spaces

c) Printing of alphabets in decrementing order.

Every character that we see on the keyboard, whether alphabets, numbers or other special characters, has a unique code value, which is basically a number. So every character, even the numbers also internally represented by a numeric value called **Unicode**. We can use the in-built function **chr()** to get alphabets from their respective Unicode values. It converts equivalent Unicode integer value to the character. *For example;*

```
// converting integer data-type to char data-type
    unicode = 65
    alphabet = chr(unicode)
```

So in an increment value order of Unicode, we can get alphabets. We can verify this from the following program that prints all 26 alphabets, capital or small, and Unicode.

Capital Alphabets	Unicode	Small Alphabets	Unicode
A	65	a	97
B	66	b	98
.....
Z	90	z	122

UniCodeToAlphabets.py

```
print("Alphabets and its equivalent unicode.")
print("-----")
for cnt in range(0,26):
    unicodeCap =( 65 + cnt )
    unicodeSmall =( 97 + cnt )
    print( chr(unicodeCap),"=",unicodeCap , "| ",end="")
    print( chr(unicodeSmall),"=",unicodeSmall )
```

Output

Alphabets and its equivalent unicode.

```
-----
A=65 | a=97
B=66 | b=98
C=67 | c=99
.....
Y=89 | y=121
Z=90 | z=122
```

```

NumberPatternGG.py
totalRows = 5 # number of rows to display
numSet2 = 0

# Row level Repetitive Action :
for row in range(0, totalRows):
    # Column level Repetitive Action :
    for col in range(1, 2 * totalRows):
        # use of ternary operator, an simpler if-else shortcut
        lw_lt_num =(totalRows - 1) if (row == 0) else (totalRows - row)
        lw_lt_spaces = 0 if (row == 0) else (totalRows - row + 1)
        up_lt_spaces =-1 if (row == 0) else (totalRows + row - 1)
        if (col <= lw_lt_num):
            numSet1 = col
        # Unicode of capital letters starts with value 65.
        # column number starts with value 1 instead of value 0 and so -1.
        print(chr (numSet1 - 1 + 65), end="")
            # updating numSet2 when numSet1 reached
            # it's maximum value
        if (col == lw_lt_num):
            numSet2 = numSet1
        elif (col >= lw_lt_spaces and col <= up_lt_spaces):
            print("", end="# 2 spaces")
        else:
            # first row where no space is required to print.
            if (row == 0 and col == totalRows):
                numSet2 = totalRows
                # Unicode of capital letters starts with value 65
            # column number starts with value 1 instead of value 0 and so -1.
            print(chr(numSet2 - 1 + 65), end="")
            numSet2 = numSet2 - 1
        # Ends inner for-loop
        print()# Moves control to the next line
    # Ends outer for-loop

```

Output

```

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

```

With a little bit of modification to the program **NumberPatternGG.py**, we can generate a similar pattern in small alphabets if we try. Instead of using the following for capital letters;

```
chr(numSet1 - 1 + 65)# Unicode for capital alphabet 'A' is 65
```

We can use the following calculation to generate a pattern with small alphabets.

```
chr(numSet1 - 1 + 97)# Unicode for small alphabet 'a' is 97
```

Although we can get output in small alphabets with minor changes, the real point is that most of the code will get repeated. We can avoid such repetition of programming logic by the concept of **functions** or methods. We can create our own function and use it on-demand similarly as we use different versions of in-built methods/functions like **print()**.

Writing our own functions

We have often used the concept of generalized expression based on variable(s) to represent hard-coded values. We can still go ahead to generalize programming logic too. One of the major advantages of function with input parameter(s) is to achieve customized output by reusing the same programming logic. Writing functions is a sensible idea that suggests how we can encapsulate different programming logics under different names. Hence, repetition of the same piece of programming logic should not happen in other functions. It should be sliced out properly as only one type of action. We can use them on demand, similar to how Lego toys are in use, building something meaningful. We will learn the concept of code reuse by implementing the most commonly used syntax to create a function. The function name should begin with keyword **def** and must end with a colon (:). A function may omit or require parameters. It all depends on whether the function requires to be input values. Basically, function parameter works as a variable and so allow various input value. Our functions should serve as a unique formula and optionally allow for input values as required. We already got this idea by using different versions of in-built methods or functions like **print()**—the example to create our own function with parameters given as follows. The programming statements written in a function should be equally indented.

```
def functionName( parameter1, parameter2,...):
```

```
# programming logic
    statement-1
    statement-2
    .....
    statement-n
```

Where *parameter1, parameter2...* are the inputs to a function. We can store the output result of a function in a variable by means of the function call given as follows.

```
result = functionName( value1, value2,...) # variable:result stores the output of this function call.
```

functionName(value1, value2,...) is a function call, and we can call it many times within a program to implement targeted functionality. The rules for naming a function or a variable is the same and given as;

- 1) Function name should begin with either character: A-Z, a-z, and underscore(_).
- 2) Remaining part of the function name can contain the characters from A-Z, a-z, digits(0-9), and underscore(_).
- 3) Any reserved word which a programming language is using can't be used. For example, we cannot create our own function and name it again as **print()** or create a variable with a name **for**.

Using the **printPattern** method, we can print the pattern for the following three different output types, encapsulating a common programming logic. The **printPattern** simply prints the pattern that matches the input parameters.

printPattern(param11, param12)	printPattern(param21, param22)	printPattern(param31, param32)
A	A	a
A B	B A	a b
A B C	C B A	a b c
A B C D	D C B A	a b c d
A B C D E	D C B A	a b c d e
		d c b a
		a
		1
		1 2
		1 2 3
		1 2 3 4
		1 2 3 4 5
		4 3 2 1
		5 4 3 2 1

PatternByFunctionCall.py

```
# Main program showing the code re-usability through functional calls.
# It also suggests applying DRY(Don't Repeat Yourself) principle.
#



totalRows = 5 # number of rows to display
unicode_A = 65 # unicode for letter A
unicode_a = 97 # unicode for letter a
unicode_1 = 49 # unicode for digit 1.



# method or function definition
def printPattern(totalRows, unicode):



    numSet1 = 0
    numSet2 = 0



    # Row level Repetitive Action :
    for row in range(0, totalRows):
    # Column level Repetitive Action :
    for col in range(1, 2 * totalRows):



        # use of ternary operator, an simpler if-else shortcut
        lw_lt_num =(totalRows - 1) if (row == 0) else (totalRows - row)
        lw_lt_spaces = 0 if (row == 0) else (totalRows - row + 1)
        up_lt_spaces =-1 if (row == 0) else (totalRows + row - 1)

        if (col <= lw_lt_num):
            numSet1 = col

        # Unicode value is from function parameter - unicode.
        # column number starts with value 1 instead of value 0 and so -1.
        print(chr (numSet1 - 1 + unicode), end="")
        # updating numSet2 when numSet1 reached it's maximum value
        if (col == lw_lt_num):
            numSet2 = numSet1

        elif (col >= lw_lt_spaces and col <= up_lt_spaces):
            print("", end="# 2 spaces")
        else:
            # first row where no space is required to print.
            if (row == 0 and col == totalRows):
                numSet2 = totalRows

            # Unicode value is from function parameter - unicode.
            # column number starts with value 1 instead of value 0 and so -1.
            print(chr(numSet2 - 1 + unicode), end="")
            numSet2 = numSet2 - 1

        # Ends inner for-loop
        print()# Moves control to the next line
        # Ends outer for-loop

    # method ends here
```

```

print("Capital Letter Pattern GG")
print("-----")
printPattern(totalRows, unicode_A)# function call

print()
print("Small Letter Pattern GG")
print("-----")
printPattern(totalRows, unicode_a)# function call

print()
print("Number Pattern FF")
print("-----")
printPattern(totalRows, unicode_1)# function call

```

Output

Capital Letter Pattern GG

```

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

```

Small Letter Pattern GG

```

-----
a b c d e d c b a
a b c d d c b a
a b c c b a
a b b a
a a

```

Number Pattern FF

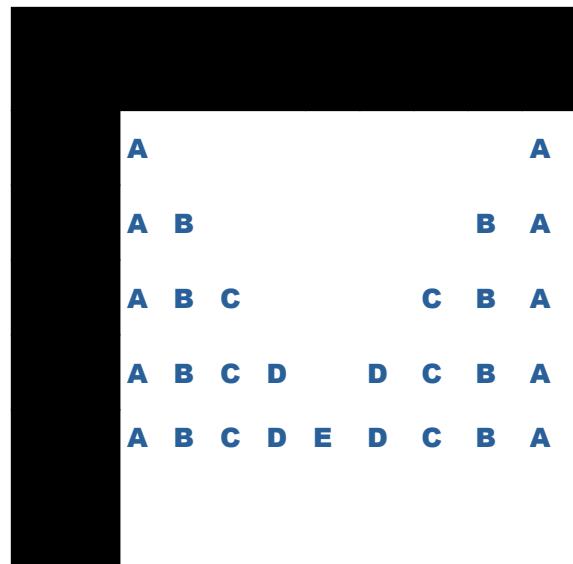
```

-----
1 2 3 4 5 4 3 2 1
1 2 3 4 4 3 2 1
1 2 3 3 2 1
1 2 2 1
1 1

```

NumberPattern HH

Our approach will be to assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique **(row, column)** numeric combination. With this assumption, the computer screen will simply be a unique combination of **(row, column)** numeric value paired boxes that are uniformly spread all across the screen.



	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1	A								A
Row=2	A	B						B	A
Row=3	A	B	C				C	B	A
Row=4	A	B	C	D		D	C	B	A
Row=5	A	B	C	D	E	D	C	B	A

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable **(row, column)** numeric paired values. **For example;** **(4, 3)** means printing character at the assumed screen location **4th row and 3rd column**.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1	(1,1) A	1,2 space	1,3 space	1,4 space	1,5 space	1,6 space	1,7 space	1,8 space	(1,9) A
Row=2	(2,1) A	(2,2) B	2,3 space	2,4 space	2,5 space	2,6 space	2,7 space	(2,8) B	(2,9) A
Row=3	(3,1) A	(3,2) B	(3,3) C	3,4 space	3,5 space	3,6 space	(3,7) C	(3,8) B	(3,9) A
Row=4	(4,1) A	(4,2) B	(4,3) C	(4,4) D	4,5 space	(4,6) D	4,7 C	(4,8) B	(4,9) A
Row=5	(5,1) A	(5,2) B	(5,3) C	(5,4) D	(5,5) E	(5,6) D	5,7 C	5,8 B	(5,9) A

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1	a								a
Row=2	a	b						b	a
Row=3	a	b	c				c	b	a
Row=4	a	b	c	d		d	c	b	a
Row=5	a	b	c	d	e	d	c	b	a

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1	1								1
Row=2	1	2						2	1
Row=3	1	2	3				3	2	1
Row=4	1	2	3	4		4	3	2	1
Row=5	1	2	3	4	5	4	3	2	1

Analysis of output

This pattern is just the reverse of pattern **NumberPatternGG**. What we have printed in the first row in the pattern **NumberPatternGG**, same we need to print in the last row. So, by following the same trick, we can generate this pattern. We can follow the given strategy for this particular pattern;

- a) The outer for-loop, which indicates the program's control row-wise movement, needs to reverse its direction. It means we need to implement decrementing for-loop of the following.

```
# Row level Repetitive Action :
```

```
for row in range(0, totalRows):
```

The decrementing for-loop for the above will be as follows;

```
# Row level Repetitive Action :
```

```
for row in range(totalRows-1,-1,-1):
```

- b) Next step, column-wise, allow it to print in the same way as it was done in **NumberPatternGG**.

Hence, basically, we only need to make a change in a row-level for-loop.

ReversePatternByFunctionCall.py

```
# Main program showing the code re-usability through functional calls.
# It also suggests applying DRY(Don't Repeat Yourself) principle.

totalRows = 5 # number of rows to display
unicode_A = 65 # unicode for letter A
unicode_a = 97 # unicode for letter a
unicode_1 = 49 # unicode for digit 1.

# method or function definition
def printPattern(totalRows, unicode):

    numSet1 = 0
    numSet2 = 0

    # Row level Repetitive Action :
    for row in range(totalRows-1,-1,-1):
        # Column level Repetitive Action :
        for col in range(1, 2 * totalRows):

            # use of ternary operator, an simpler if-else shortcut
            lw_lt_num =(totalRows - 1) if (row == 0) else (totalRows - row)
            lw_lt_spaces = 0 if (row == 0) else (totalRows - row + 1)
            up_lt_spaces =-1 if (row == 0) else (totalRows + row - 1)

            if (col <= lw_lt_num):
                numSet1 = col

            # Unicode value is from function parameter - unicode.
            # column number starts with value 1 instead of value 0 and so -1.
            print(chr (numSet1 - 1 + unicode), end="")

            # updating numSet2 when numSet1 reached its maximum value
            if (col == lw_lt_num):
                numSet2 = numSet1

            elif (col >= lw_lt_spaces and col <= up_lt_spaces):
                print("", end="# 2 spaces")
            else:
                # first row where no space is required to print.
                if (row == 0 and col == totalRows):
                    numSet2 = totalRows

            # Unicode value is from function parameter - unicode.
            # column number starts with value 1 instead of value 0 and so -1.
            print(chr(numSet2 - 1 + unicode), end="")
            numSet2 = numSet2 - 1

        # Ends inner for-loop
        print()# Moves control to the next line
    # Ends outer for-loop

    # method ends here
```

```

print("Capital Letter Pattern GG")
print("-----")
printPattern(totalRows, unicode_A)# function call

print()
print("Small Letter Pattern GG")
print("-----")
printPattern(totalRows, unicode_a)# function call

print()
print("Number Pattern FF")
print("-----")
printPattern(totalRows, unicode_1)# function call

```

Output

Capital Letter Pattern GG

```

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

```

Small Letter Pattern GG

```

-----
a a
a b b a
a b c c b a
a b c d d c b a
a b c d e d c b a

```

Number Pattern FF

```

-----
1 1
1 2 2 1
1 2 3 3 2 1
1 2 3 4 4 3 2 1
1 2 3 4 5 4 3 2 1

```

Problem: Outputting following patterns using the concept of function.

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1	A	B	C	D	E	D	C	B	A
Row=2	A	B	C	D		D	C	B	A
Row=3	A	B	C				C	B	A
Row=4	A	B						B	A
Row=5	A								A
Row=6	A	B						B	A
Row=7	A	B	C				C	B	A
Row=8	A	B	C	D		D	C	B	A
Row=9	A	B	C	D	E	D	C	B	A

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1	a	b	c	d	e	d	c	b	a
Row=2	a	b	c	d		d	c	b	a
Row=3	a	b	c				c	b	a
Row=4	a	b						b	a
Row=5	a								a
Row=6	a	b						b	a
Row=7	a	b	c				c	b	a
Row=8	a	b	c	d		d	c	b	a
Row=9	a	b	c	d	e	d	c	b	a

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1	1	2	3	4	5	4	3	2	1
Row=2	1	2	3	4		4	3	2	1
Row=3	1	2	3				3	2	1
Row=4	1	2						2	1
Row=5	1								1
Row=6	1	2						2	1
Row=7	1	2	3				3	2	1
Row=8	1	2	3	4		4	3	2	1
Row=9	1	2	3	4	5	4	3	2	1

NumberPattern AAA

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will be to assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique (row, column) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

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

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	1				
Row=2	2	3			
Row=3	3	4	5		
Row=4	4	5	6	7	
Row=5	5	6	7	8	9

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (row, column) numeric paired values. For example;

(4, 3) means printing numeric characters at the assumed screen location **4th row and 3rd column**.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	1,1				
Row=2	2,1	2,2			
Row=3	3,1	3,2	3,3		
Row=4	4,1	4,2	4,3	4,4	
Row=5	5,1	5,2	5,3	5,4	5,5

Analysis of output.

It can also be interpreted in a tabular form.

row	1				
Col	1				
Output	1				

For row =1, $1 \leq \text{col} \leq 1$

row	2				
Col	1	2			
Output	2	3			

For row =2, $1 \leq \text{col} \leq 2$

row	3				
Col	1	2	3		
Output	3	4	5		

For row =3, $1 \leq \text{col} \leq 3$

row	4				
Col	1	2	3	4	
Output	4	5	6	7	

For row =4, $1 \leq \text{col} \leq 4$

row	5				
Col	1	2	3	4	5
Output	5	6	7	8	9

For row =5, $1 \leq \text{col} \leq 5$

We need to generalize the lower_limit & upper_limit for all the rows, so it should be suitable for any number of rows.

Observations

1. There is repetition required at row level vertically, i.e., Row = 1 to 5.
2. There is repetition required at col level horizontally., i.e., Col = 1 to 5. So it's a 2-way repetition process means it requires 2 for-loops(nested for – loops).
3. We need to explore the unknown logical expressions which we will use to define the following given inner-loop. Those unknown logical expressions marked as **Unknown** in the table given below.

For Loop	Lower limit	Upper limit
Outer loop	row=1	row<=5
Inner Loop	col=(Unknown) Need to find a relation how exactly it's varying	(Unknown) Need to find a relation to how exactly it's varying.

COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action							Expected Output
row	Column level Repetitive Action						
	loop-condition(lower_limit <=col<= upper_limit)		lower_limit		upper_limit		
	default	1	default	1	2	3	
1	1	row	1	1	1+1 - 1	row + row - 1	1
2	2	row	3	2 + 1	2+2 - 1	row + row - 1	2 3
3	3	row	5	3 + 2	3+3 - 1	row + row - 1	3 4 5
4	4	row	7	4 + 3	4+4 - 1	row + row - 1	4 5 6 7
5	5	row	9	5 + 4	5+5 - 1	row + row - 1	5 6 7 8 9

The lower_limit matches the row value so that it can be generalized to the variable row.

Row level Repetitive Action							Expected Output
row	Column level Repetitive Action						
	loop-condition(lower_limit <=col<= upper_limit)		lower_limit		upper_limit		
	default	1	default	1	2	3	
1	1	row	1	1	1+1 - 1	2*row - 1	1
2	2	row	3	2 + 1	2+2 - 1	2*row - 1	2 3
3	3	row	5	3 + 2	3+3 - 1	2*row - 1	3 4 5
4	4	row	7	4 + 3	4+4 - 1	2*row - 1	4 5 6 7
5	5	row	9	5 + 4	5+5 - 1	2*row - 1	5 6 7 8 9

Row level Repetitive Action							Expected Output
row	Column level Repetitive Action						
	loop-condition(lower_limit <=col<= upper_limit)		lower_limit		upper_limit		
	1	2	default	2	3	4	
1	row	row	1	1+1 - 1	2*row - 1	2*row - 1	1
2	row		3	2+2 - 1	2*row - 1		2 3
3	row		5	3+3 - 1	2*row - 1		3 4 5
4	row		7	4+4 - 1	2*row - 1		4 5 6 7
5	row		9	5+5 - 1	2*row - 1		5 6 7 8 9

The lower_limit & upper_limit have become consistent for all the rows. The lower_limit is generalized to row, and the upper_limit is generalized ($2*row - 1$). These logical expressions are such that, commonly applying for all rows and columns helps to give matching output.

So our final logical expression becomes;

loop-condition	lower_limit <=col<= upper_limit
Mathematical expression	$row \leq col \leq (2*row - 1)$
syntax	<code>for col in range(row, 2 * row):</code>

```
NumberPatternAAA.py
totalRows = 5 # number of rows to display

# Row level Repetitive Action :
for row in range(1, totalRows + 1):

    # Column level Repetitive Action:
    # print character in the same row.
    for col in range(row, 2 * row):

        # added extra space for clarity in output.
        print(col, end = "")
    # Ends inner for-loop

    # move control to the next line
    print()
# Ends inner for-loop
```

Output

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

ProblemGBAAA: Modify program NumberPatternAAA.py so that it should show the following output using switch-case for values 1 and 2, respectively.

Output

```
1
3
3 5
5 7
5 7 9
```

Output

```
2
4
4 6
6 8
```

NumberPattern BBB

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will be to assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique (**row, column**) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

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

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1					1
Row=2					3 2
Row=3					5 4 3
Row=4			7	6	5 4
Row=5	9	8	7	6	5

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (**row, column**) numeric paired values. For example; (4, 3) means printing numeric characters at the assumed screen location 4th row and 3rd column.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1					1,5
Row=2				2,4	2,5
Row=3			3,3	3,4	3,5
Row=4		4,2	4,3	4,4	4,5
Row=5	5,1	5,2	5,3	5,4	5,5

Analysis of output

In this case, we need to take care of printing two different types of characters at a particular column in a particular row:

1) Space character

2) Numeric character

row	1				
Col	1	2	3	4	5
Output	space	space	space	space	1

For row =1, $1 \leq \text{spaces} \leq 4, 1 \geq \text{num} \geq 1$

row	2				
Col	1	2	3	4	5
Output	space	space	space	3	2

For row =2, $1 \leq \text{spaces} \leq 3, 3 \geq \text{num} \geq 2$

row	3				
Col	1	2	3	4	5
Output	space	space	5	4	3

For row =3, $1 \leq \text{spaces} \leq 2, 5 \geq \text{num} \geq 3$

row	4				
Col	1	2	3	4	5
Output	space	7	6	5	4

For row =4, $1 \leq \text{spaces} \leq 1, 7 \geq \text{num} \geq 4$

row	5				
Col	1	2	3	4	5
Output	9	8	7	6	5

For row =5, $9 \geq \text{num} \geq 5$

Observations

- a. There is repetition at row level vertically, i.e., Row = 1 to 5.
There is repetition at col level horizontally., i.e., Col = 1 to 5.
- b. These get divided into two parts—one for repetitively printing spaces and the remaining column for printing numeric characters.
- c. We need to explore the remaining logical expressions to define the following given inner-loops. We can mark as **Unknown** the logical expressions which are not discovered in the table given below.

For Loop	Lower limit	Upper limit
Outer loop	row=1	row<=5
Inner Loop - spaces	col=1	Unknown
Inner Loop - Number	num = Unknown	num = 1

COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action					Expected Output
row	Column Level Repetitive Action				
	lower_limit <= col <= upper_limit		upper_limit >= num >= lower_limit		
	lower_limit	upper_limit	upper_limit	lower_limit	
1	1	4	1	1	1
2	1	3	3	2	3 2
3	1	2	5	3	5 4 3
4	1	1	7	4	7 6 5 4
5	1	0	9	5	9 8 7 6 5

We need to develop a consistent expression for upper_limit, for which it should suit any number of rows. We can refer previous pattern problem **PatternAAA**'s result for the num variable.

The other variables are row & totalRows=5. Let's try to generalize using these variables.

Row level Repetitive Action							
row	Column Level Repetitive Action						
	lower_limit <= col <= upper_limit				upper_limit >= num >= lower_limit		
	lower_limit		upper_limit		upper_limit		lower_limit
1	1	4	totalRows - 1	totalRows - row	1	row	row
2	1	3	totalRows - 2	totalRows - row	3	row + 1	row
3	1	2	totalRows - 3	totalRows - row	5	row + 2	row
4	1	1	totalRows - 4	totalRows - row	7	row + 3	row
5	1	0	totalRows - 5	totalRows - row	9	row + 4	row

Again trying to express the hardcoded numeric values of upper_limit related to variable num in terms of variable row.

Column Level Repetitive Action							
row	lower_limit <= col <= upper_limit			upper_limit >= num >= lower_limit			
	lower_limit	upper_limit	upper_limit				lower_limit
1	1	totalRows - row	1	row	row + 1 - 1	row + row - 1	row
2	1	totalRows - row	3	row + 1	row + 2 - 1	row + row - 1	row
3	1	totalRows - row	5	row + 2	row + 3 - 1	row + row - 1	row
4	1	totalRows - row	7	row + 3	row + 4 - 1	row + row - 1	row
5	1	totalRows - row	9	row + 4	row + 5 - 1	row + row - 1	row

Column Level Repetitive Action							
row	lower_limit <= col <= upper_limit			upper_limit >= num >= lower_limit			
	lower_limit	upper_limit	upper_limit				lower_limit
1	1	totalRows - row	1	row + 1 - 1	row + row - 1	2*row - 1	row
2	1	totalRows - row	3	row + 2 - 1	row + row - 1	2*row - 1	row
3	1	totalRows - row	5	row + 3 - 1	row + row - 1	2*row - 1	row
4	1	totalRows - row	7	row + 4 - 1	row + row - 1	2*row - 1	row
5	1	totalRows - row	9	row + 5 - 1	row + row - 1	2*row - 1	row

Column Level Repetitive Action

row	lower_limit <= col <= upper_limit		upper_limit >= num >= lower_limit	
	lower_limit	upper_limit	upper_limit	lower_limit
1	1	totalRows - row	2*row - 1	row
2	1	totalRows - row	2*row - 1	row
3	1	1 totalRows - row	2*row - 1 2*row - 1	row row
4	1	totalRows - row	2*row - 1	row
5	1	totalRows - row	2*row - 1	row

So the upper_limit of col & num variables are generalized. The lower_limit & upper_limit are consistent for all the rows. It's now generalized for any number of rows.

So our loop-condition for space count on each row using variable col becomes

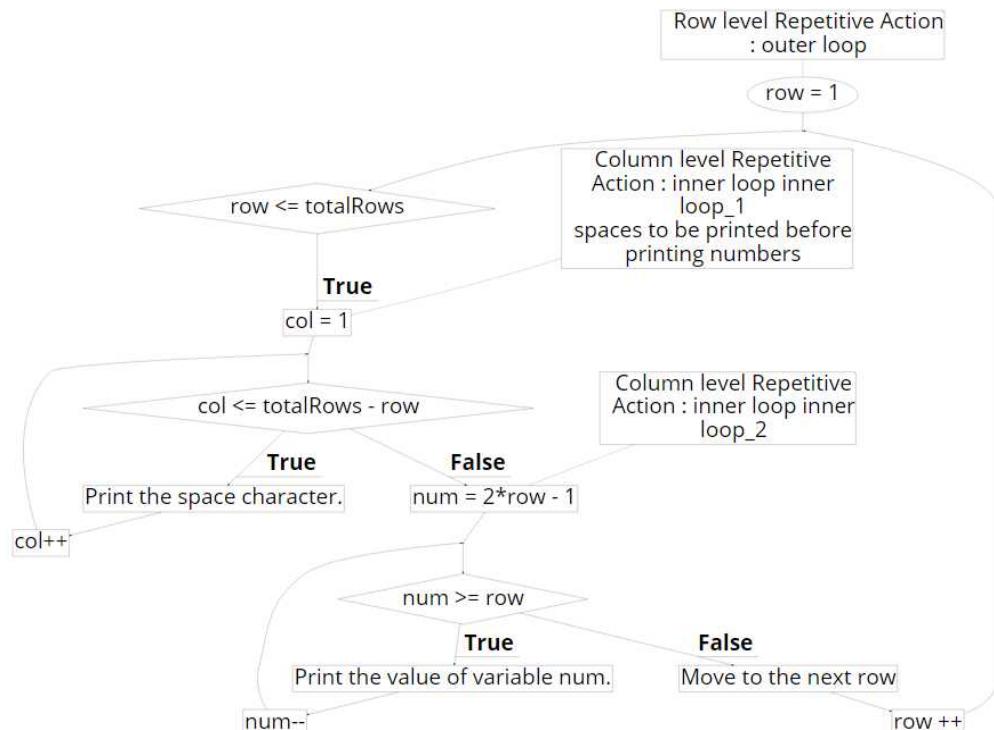
loop-condition	lower_limit <= col <= upper_limit
Mathematical expression	1 <= col <= (totalRows - row)
syntax	for col in range(1, totalRows - row + 1):

So our loop-condition for printing numbers on each row using variable:num becomes

loop-condition	upper_limit >= num >= lower_limit
Mathematical expression	2 * row - 1 >= num >= row
syntax	for num in range(2 * row - 1, row - 1, -1):

So it has two repetition processes, one for printing spaces and the second for printing numeric values. It means it requires 2 inner for-loops (nested for – loops).

The flowchart can be given as follows to capture the details.



```
NumberPatternBBB1.py
totalRows = 5 # number of rows to display
    # Row level Repetitive Action :
for row in range(1, totalRows + 1):
    # Column level Repetitive Action:
    # print character in the same row.
    # spaces to be printed before printing numbers
    for col in range(1, totalRows - row + 1):
        print("", end ="")

    for num in range(2 * row - 1, row - 1,-1):
        # added extra space for clarity in output.
        print(num , end ="")

    print()# moves control to the next line
```

Output

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

Alternative solution

Let's devise a solution by merging two inner for-loops into a single by applying the if-else conditional statement. We can get some idea from the for-loop's condition part of both the inner for-loops. The loop-initialization needs to do it once. If we take care of the loop-initialization and step-decrementing part, the loop-condition of both the inner-loops can be converted into **if-condition**.

NumberPatternBBB2.py

```
totalRows = 5 # number of rows to display
    # Row level Repetitive Action :
for row in range(1, totalRows + 1):

    # Column level Repetitive Action :
    # print character in the same row.

    # spaces to be printed before printing numbers
num = 2 * row - 1 # horizontal movements

# spaces to be printed before printing numbers
for col in range(1, totalRows + 1):

    if (col <=(totalRows - row)):
        print("", end ="")
    elif (num >= row):
        # added extra space for clarity in output.
    print(num, end ="")
    num = num - 1

print()# moves control to the next line

# Ends outer for-loop
```

NumberPattern CCC

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will be to assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique (**row, column**) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

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

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	5	6	7	8	9
Row=2	4	5	6	7	
Row=3	3	4	5		
Row=4	2	3			
Row=5	1				

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (**row, column**) numeric paired values. **For example;** (4, 3) means printing numeric characters at the assumed screen location 4th row and 3rd column.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	5,1	5,2	5,3	5,4	5,5
Row=2	4,1	4,2	4,3	4,4	
Row=3	3,1	3,2	3,3		
Row=4	2,1	2,2			
Row=5	1,1				

Analysis of output

When Row = 1, Col = 1,2,3,4,5 (1 to 5): repetition will be 5 times horizontally ($1 \leq \text{Col} \leq 5$)

When Row = 2, Col = 1,2,3,4 (1 to 4): repetition will be 4 times horizontally ($1 \leq \text{Col} \leq 4$)

When Row = 3, Col = 1,2,3 (1 to 3): repetition will be 3 times horizontally ($1 \leq \text{Col} \leq 3$)

When Row = 4, Col = 1,2 (1 to 2): repetition will be 2 times horizontally ($1 \leq \text{Col} \leq 2$)

When Row = 5, Col = 1 : will be 1 time ($1 \leq \text{Col} \leq 1$)

We need to take care of numeric output logic only and not take care of the logic required to print spaces. The logic for spaces is only to be considered if spaces come before numeric output.

So, it's irrelevant in this case.

row	5					
col	5	6	7	8	9	
output	5	6	7	8	9	

row = 5 ,($5 \leq \text{Col} \leq 9$)(print till upper_limit=9)

row	4					
col	4	5	6	7		
output	4	5	6	7		

row = 4 ,($4 \leq \text{Col} \leq 7$)(print till upper_limit=7)

row	3					
col	3	4	5			
output	3	4	5			

row = 3 ,($3 \leq \text{Col} \leq 5$)(print till upper_limit=5)

row	2					
col	2	3				
output	2	3				

row = 2 ,($2 \leq \text{Col} \leq 2$)(print till upper_limit=3)

row	1					
col	1					
output	1					

row = 1 ,($1 \leq \text{Col} \leq 1$)(print till upper_limit=1)

The row value equals the lower_limit of the col variable. The upper_limit value of the col variable varies in each row.

Let's devise a solution by **decrementing the outer for-loop using variable row** (5 to 1) and determining the loop-condition($\text{lower_limit} \leq \text{col} \leq \text{upper_limit}$) of col variable to give the desired numeric output. Let's try to get a generalized expression using the variable **row**.

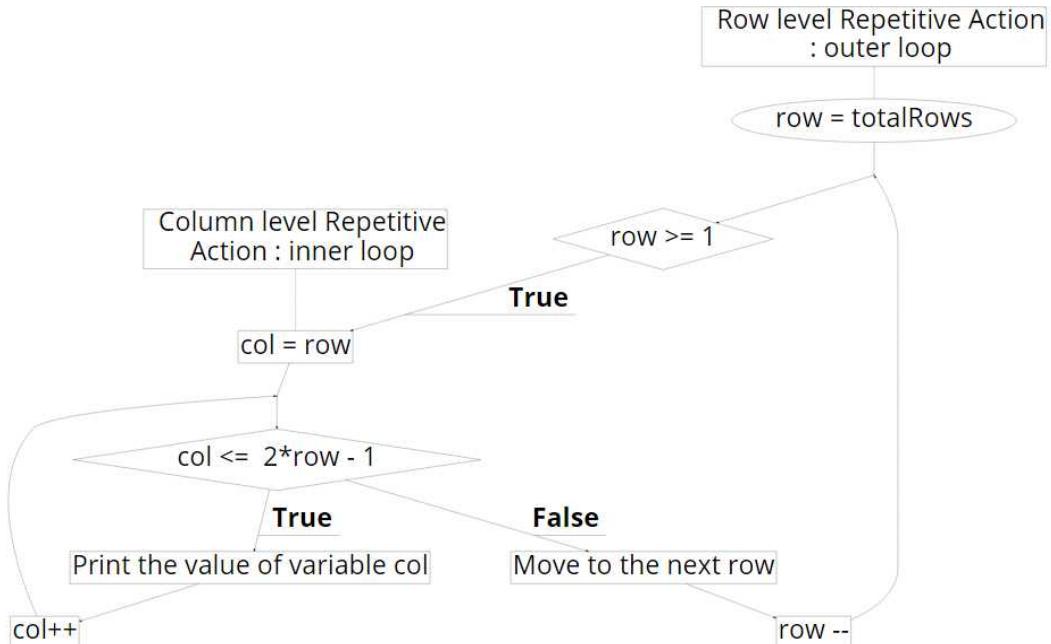
COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action(decrementing loop)									Expected Output
row	Column Level Repetitive Action								
	loop-condition (lower_limit <=col<=upper_limit)								
	lower_limit		upper_limit						
	1	1	2	3	4	5			
5	5	row	9	5+4	row + 4	row + 5-1	row + row - 1	2* row - 1	5 6 7 8 9
4	4	row	7	4+3	row + 3	row + 4-1	row + row - 1	2* row - 1	4 5 6 7
3	3	row	5	3+2	row + 2	row + 3-1	row + row - 1	2* row - 1	3 4 5
2	2	row	3	2+1	row + 1	row + 2-1	row + row - 1	2* row - 1	2 3
1	1	row	1	1+0	row + 0	row + 1-1	row + row - 1	2* row - 1	1

So our inner loop-condition to give numeric output on each row using variable col becomes

loop-condition	lower_limit <=col<= upper_limit
Mathematical expression	$\text{row} \leq \text{col} \leq (2 * \text{row} - 1)$
syntax	<code>for col in range(row, 2 * row):</code>

The flowchart can be given as follows.



Let execute the following program to verify the output.

NumberPatternCCC1.py

```
totalRows = 5 # number of rows to display

# Row level Repetitive Action :
for row in range(totalRows, 0,-1):

    # Column level Repetitive Action :
    # print character in the same row.
    # spaces to be printed before printing numbers
    for col in range(row, 2 * row):
        print(col, end ="")
        # Ends inner for-loop

    print()# moves control to the next line
# Ends outer for-loop
```

Output

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

ProblemGBCC1: Modify program NumberPatternCCC1.py so that it should give the following output.

Output

```
9
7
5
3
1
```

Alternative solution

Let's try to devise a solution using incrementing for-loop for row variable (1 to 5) and determining the loop-condition for col variable.

The row value, in this case, doesn't match the lower_limit of col variable on each row. So row variable; we can't use it directly to express lower_limit of col variable.

We shall try other variable totalRows = 5 (default number of rows) to generalize it.

Row level Repetitive Action(incrementing loop)						Expected Output
row	Column Level Repetitive Action					
	loop-condition(lower_limit <=col<= upper_limit)					
lower_limit		upper_limit				
default		1	2	default	1	
1	5	totalRows	totalRows +1 - 1	9	2* totalRows - 1	5 6 7 8 9
2	4	totalRows -1	totalRows +1 - 2	7	2* totalRows - 3	4 5 6 7
3	3	totalRows -2	totalRows +1 - 3	5	2* totalRows - 5	3 4 5
4	2	totalRows -3	totalRows +1 - 4	3	2* totalRows - 7	2 3
5	1	totalRows -4	totalRows +1 - 5	1	2* totalRows - 9	1

Part of lower_limit's logical expression matches with the row value. Since all numeric values match the row variable's value in each row, we should use the row variable to remove numeric values occurring in the lower_limit's logical expression.

Column Level Repetitive Action					Expected Output
row	loop-condition(lower_limit <=col<= upper_limit)				
	lower_limit		upper_limit		
default		3	default	1	
1	5	totalRows +1 - row	9	2* totalRows - 1	5 6 7 8 9
2	4	totalRows +1 - row	7	2* totalRows - 3	4 5 6 7
3	3	totalRows +1 - row	5	2* totalRows - 5	3 4 5
4	2	totalRows +1 - row	3	2* totalRows - 7	2 3
5	1	totalRows +1 - row	1	2* totalRows - 9	1

Now it's the turn to generalize the upper_limit of the col variable.

Column Level Repetitive Action					Expected Output
row	loop-condition(lower_limit <=col<= upper_limit)				
	lower_limit		upper_limit		
3		default	1	2	
1	totalRows +1 - row	9	2* totalRows - 1	2* totalRows - 1	5 6 7 8 9
2	totalRows +1 - row	7	2* totalRows - 3	2* totalRows - 2 -1	4 5 6 7
3	totalRows +1 - row	5	2* totalRows - 5	2* totalRows - 3 -2	3 4 5
4	totalRows +1 - row	3	2* totalRows - 7	2* totalRows - 4 -3	2 3
5	totalRows +1 - row	1	2* totalRows - 9	2* totalRows - 5 -4	1

Let's try to generalize part of the upper_limit using row variable and can be given as;

Column Level Repetitive Action				Expected Output	
row	loop-condition(lower_limit <=col<= upper_limit)				
	lower_limit	upper_limit			
	3	4	5		
1	totalRows +1 - row	2* totalRows - row	2* totalRows - row -1 +1	5 6 7 8 9	
2	totalRows +1 - row	2* totalRows - row - 1	2* totalRows - row -2+1	4 5 6 7	
3	totalRows +1 - row	2* totalRows - row - 2	2* totalRows - row - 3+1	3 4 5	
4	totalRows +1 - row	2* totalRows - row - 3	2* totalRows - row - 4+1	2 3	
5	totalRows +1 - row	2* totalRows - row - 4	2* totalRows - row - 5+1	1	

Again, we can use the row variable in the upper_limit to get rid of hard-coded numeric values.

Column Level Repetitive Action				Expected Output	
row	loop-condition(lower_limit <=col<= upper_limit)				
	lower_limit	upper_limit			
	3	6	7		
1	totalRows +1 - row	2* totalRows - row -row +1	2*(totalRows - row)+ 1	5 6 7 8 9	
2	totalRows +1 - row	2* totalRows - row -row +1	2*(totalRows - row)+ 1	4 5 6 7	
3	totalRows +1 - row	2* totalRows - row -row +1	2*(totalRows - row)+ 1	3 4 5	
4	totalRows +1 - row	2* totalRows - row -row +1	2*(totalRows - row)+ 1	2 3	
5	totalRows +1 - row	2* totalRows - row -row +1	2*(totalRows - row)+ 1	1	

The lower_limit and upper_limit expressions both have +1.

Moreover, If we run outer loop through [1 <=row<= totalRows] or [0 <=row< totalRows],in either case it is the same.

By deducting 1 from a row, lower_limit and upper_limit expressions will make it simpler.

Column Level Repetitive Action			Expected Output
row	loop -condition(lower_limit <=col< upper_limit)		
	lower_limit expression	upper_limit expression	
1 -1	totalRows +1 – row - 1	2*(totalRows - row)+ 1 - 1	5 6 7 8 9
2 -1	totalRows +1 – row - 1	2*(totalRows - row)+ 1 - 1	4 5 6 7
3 -1	totalRows +1 – row - 1	2*(totalRows - row)+ 1 - 1	3 4 5
4 -1	totalRows +1 – row - 1	2*(totalRows - row)+ 1 - 1	2 3
5-1	totalRows +1 – row - 1	2*(totalRows - row)+ 1 - 1	1

Column Level Repetitive Action				Expected Output
row	loop-condition(lower_limit <= col < upper_limit)			
	lower_limit expression		upper_limit expression	
0	totalRows - row		2*(totalRows - row)	5 6 7 8 9
1	totalRows - row		2*(totalRows - row)	4 5 6 7
2	totalRows - row	totalRows - row	2*(totalRows - row)	3 4 5
3	totalRows - row		2*(totalRows - row)	2 3
4	totalRows - row		2*(totalRows - row)	1

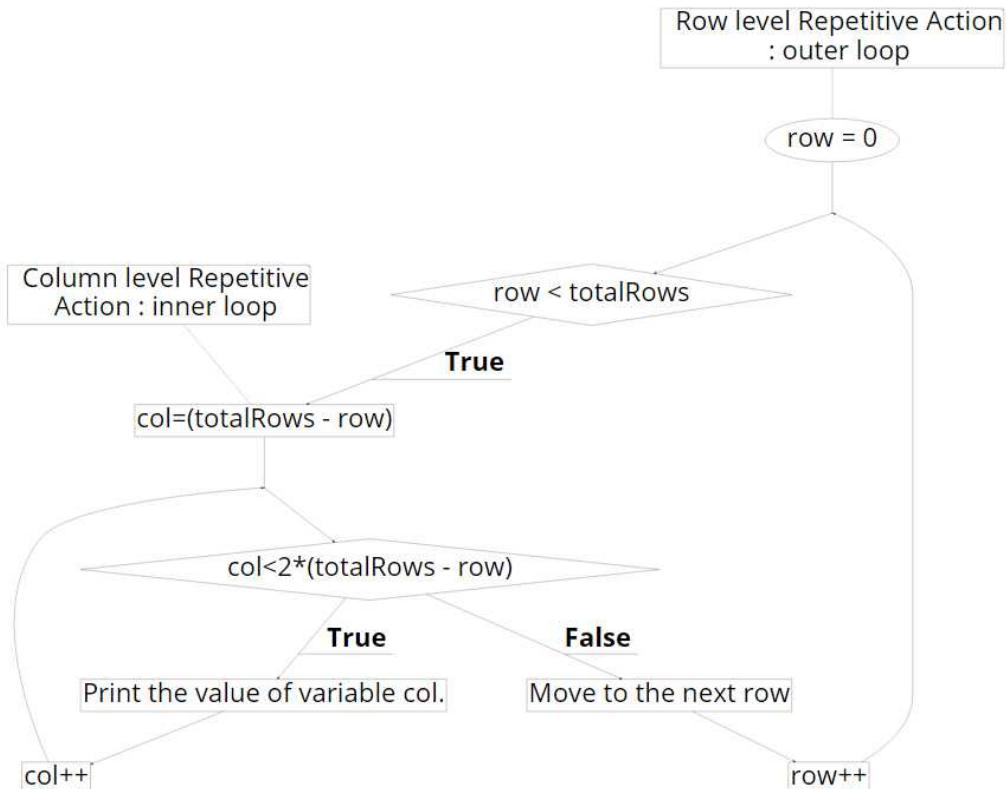
So our **outer for-loop** condition becomes

loop-condition	lower_limit <= row < upper_limit
Mathematical expression	$0 \leq \text{row} < \text{totalRows}$
syntax	<code>for row in range(0, totalRows):</code>

So our **inner for-loop** condition becomes;

loop-condition	lower_limit <= col < upper_limit
Mathematical expression	$(\text{totalRows} - \text{row}) \leq \text{col} < 2 * (\text{totalRows} - \text{row})$
syntax	<code>for col in range((totalRows - row), 2 * (totalRows - row)):</code>

The flowchart can be given as follows.



NumberPatternCCC2.py

```
totalRows = 5 # number of rows to display

# Row level Repetitive Action :
for row in range(0, totalRows):

    # Column level Repetitive Action :
    # print character in the same row.
    lower_limit =(totalRows - row)
    upper_limit = 2 *(totalRows - row)
    for col in range(lower_limit, upper_limit):
        print(col, end="")
        # Ends inner for-loop

    print()# moves control to the next line
# Ends outer for-loop
```

Output

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

NumberPattern DDD

Our programming job is to print this particular logic-based pattern on one part of the computer screen. Our approach will be to assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique (**row, column**) numeric combination. With this assumption, the computer screen will simply be a unique combination of (row, column) numeric value paired boxes that are uniformly spread all across the screen.

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

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

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	9	8	7	6	5
Row=2		7	6	5	4
Row=3			5	4	3
Row=4				3	2
Row=5					1

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable (**row, column**) numeric paired values. For example; (2, 3) means printing numeric characters at the assumed screen location 2nd row and 3rd column.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5
Row=1	1,1	1,2	1,3	1,4	1,5
Row=2		2,2	2,3	2,4	2,5
Row=3			3,3	3,4	3,5
Row=4				4,4	4,5
Row=5					5,5

Analysis of output

In this case, we need to take care of printing two different types of characters at a particular column in a particular row:

1) Space character

2) Numeric character

row	5				
Col	1	2	3	4	5
Output	9	8	7	6	5

For row =1, $0 \leq \text{spaces} \leq 0, 9 \geq \text{num} \geq 5$

row	4				
Col	1	2	3	4	5
Output	space	7	6	5	4

For row =2, $1 \leq \text{spaces} \leq 1, 9 \geq \text{num} \geq 5$

row	3				
Col	1	2	3	4	5
Output	space	space	5	4	3

For row =3, $1 \leq \text{spaces} \leq 2, 5 \geq \text{num} \geq 3$

row	2				
Col	1	2	3	4	5
Output	space	space	space	3	2

For row =4, $1 \leq \text{spaces} \leq 3, 3 \geq \text{num} \geq 2$

row	1				
Col	1	2	3	4	5
Output	space	space	space	space	1

For row =5, $1 \leq \text{spaces} \leq 4, 1 \geq \text{num} \geq 1$

Observations

- There is repetition at row level vertically, i.e., Row = 1 to 5.
- There is repetition at col level horizontally., i.e., Col = 1 to 5.
- These get divided into two parts—one for repetitively printing spaces and the remaining column for printing numeric characters.
- In the table given below marked as **Unknown**, the logical expressions which are not discovered. We need to explore the remaining logical expressions to define the following given inner-loops.

For Loop	Lower limit	Upper limit
Outer loop(decrementing)	row=1	Row>=5
Inner Loop - spaces	col=1	Unknown
Inner Loop – Number (decrementing)	num = row	Unknown

COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action						Expected Output
row	Column level Repetitive Action					
	lower_limit <= col <= upper_limit		upper_limit >= num >= lower_limit			
	lower_limit upper_limit		upper_limit	lower_limit		
row = 5	1	0	9	5	row	9 8 7 6 5
row = 4	1	1	7	4	row	7 6 5 4
row = 3	1	2	5	3	row	5 4 3
row = 2	1	3	3	2	row	3 2
row = 1	1	4	1	1	row	1

The lower_limit for spaces & num variable has the same numeric value expression for all the rows. It's consistent for all the rows. Likewise, we need to develop such a consistent expression for upper_limit, for which it should suit any number of rows.

The other remaining variables are row & totalRows=5. Let's try to generalize using these variables.

Column level Repetitive Action						
row	lower_limit <= col <= upper_limit			upper_limit >= num >= lower_limit		
	lower_limit	upper_limit		upper_limit	lower_limit	
row = 5	1	0	5-5	totalRows - row	9	row
row = 4		1	5-4	totalRows - row	7	
row = 3		2	5-3	totalRows - row	5	
row = 2		3	5-2	totalRows - row	3	
row = 1		4	5-1	totalRows - row	1	

The upper_limit of row variable related inner for-loop is a generalized, consistent expression on all the rows. The upper_limit of num variable related inner for-loop is not generalized as not having a consistent numeric or logical expression on all the rows.

Column level Repetitive Action								
row	lower_limit <= col <= upper_limit			upper_limit >= num >= lower_limit				
	lower_limit	upper_limit		upper_limit	lower_limit			
row = 5	1	0	totalRows - row	totalRows - row	9	10-1	2*5-1	row
row = 4		1	totalRows - row		7	8-1	2*4-1	
row = 3		2	totalRows - row		5	6-1	2*3-1	
row = 2		3	totalRows - row		3	4-1	2*2-1	
row = 1		4	totalRows - row		1	2-1	2*1-1	

Column level Repetitive Action						
row	lower_limit <= col <= upper_limit			upper_limit >= num >= lower_limit		
	lower_limit	upper_limit		upper_limit	lower_limit	
row = 5	1	totalRows - row	9	10-1 = 2* 5 -1	2*row -1	row
row = 4			7	8-1 = 2* 4 -1	2*row -1	
row = 3			5	6-1 = 2* 3 -1	2*row -1	
row = 2			3	4-1 = 2* 2 -1	2*row -1	
row = 1			1	2-1 = 2* 1 -1	2*row -1	

row	Column level Repetitive Action					
	lower_limit <= col <= upper_limit		upper_limit >= num >= lower_limit			
	lower_limit	upper_limit	upper_limit			lower_limit
row = 5	1	totalRows - row	2*	5	-1	2*row - 1
row = 4			2*	4	-1	2*row - 1
row = 3			2*	3	-1	2*row - 1
row = 2			2*	2	-1	2*row - 1
row = 1			2*	1	-1	2*row - 1

So the upper_limit of col & num variables are generalized. The lower_limit & upper_limit are consistent for all the rows. It's now generalized for any number of rows. So it has two repetition processes, one for printing spaces and the second for printing numeric values. It means requiring two inner for-loops to Print characters at the column level and one outer loop for iterating rows.

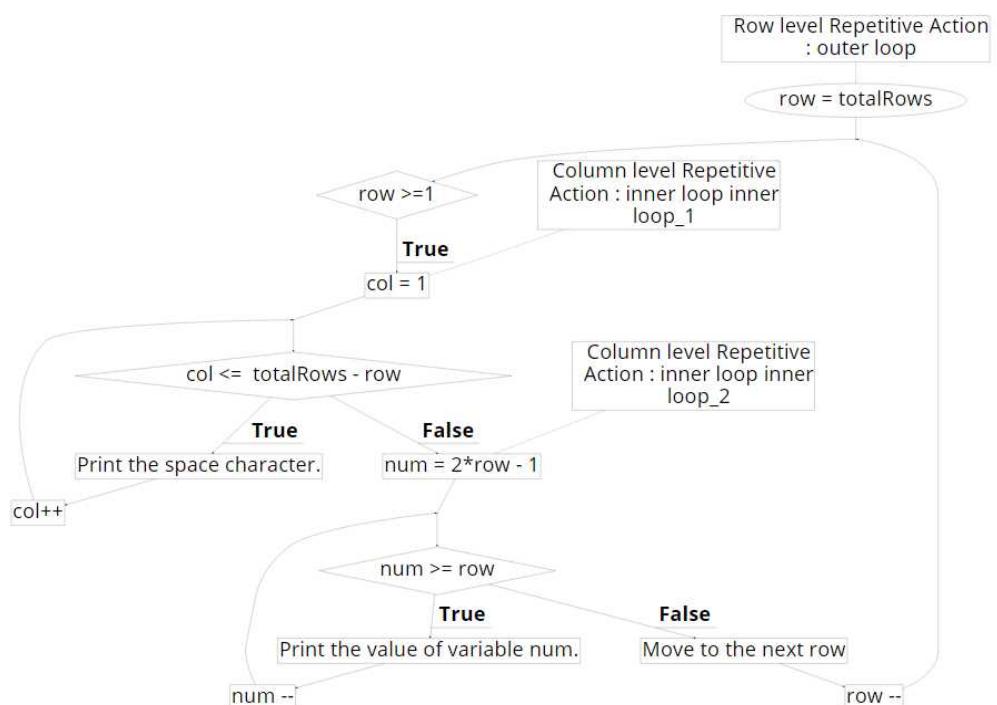
So our inner for-loop condition for printing spaces becomes

loop-condition	lower_limit <= col <= upper_limit
Mathematical expression	$1 \leq col \leq (\text{totalRows} - \text{row})$
syntax	<code>for col in range(1, totalRows - row + 1):</code>

So our inner for-loop condition for printing numbers becomes;

loop-condition	upper_limit >= num >= lower_limit
Mathematical expression	$(2 * \text{row} - 1) \geq num \geq \text{row}$
syntax	<code>for num in range(2 * row - 1, row - 1, -1):</code>

We can summarize all the related information in a flowchart as give.



Let's execute the following program.

```
NumberPatternDDD1.py
totalRows = 5 # number of rows to display

    # Row level Repetitive Action :
for row in range(totalRows, 0,-1):

    # Column level Repetitive Action :
    # print character in the same row.

    # spaces to be printed before printing numbers
for col in range(1, totalRows - row + 1):
print("", end="")# 2 spaces
    # inner for-loop_1

    # num value to be printed and ends with a space
for num in range(2 * row - 1, row - 1,-1):
print(num, end ="")
    # inner for-loop_2

print()# moves control to the next line
    # Ends outer for-loop
```

Output

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

Alternative solution

Since the column values get a clear split between printing spaces and numbers, we can try to reproduce the same result using the **if-else condition** construct and use the **loop-condition** for spaces as an if-condition and printing number in **else-condition**.

NumberPatternDDD2.py

```
totalRows = 5 # number of rows to display

# Row level Repetitive Action :
for row in range(totalRows, 0,-1):

    num = 2 * row - 1

        # Column level Repetitive Action :
    for col in range(1, totalRows + 1):

        if (col <=(totalRows - row)):
            print("", end ="")# when spaces are to be printed
        else:
            print(num,end ="")# when numbers are to be printed
        num -= 1

            # Ends inner for-loop

    print()# Move to the next row

    # Ends outer for-loop
```

Output

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

NumberPatternEEE

Our programming job is to print this particular logic-based pattern on one part of the computer screen. **Our approach will be to assume that the computer screen is a combination of rows and columns, virtually dividing it into smaller boxes**—the smallest box treated as a resident of just a single character only. The address in each smallest box on the display screen will logically become a unique **(row, column)** numeric combination. With this assumption, the computer screen will simply be a unique combination of **(row, column)** numeric value paired boxes that are uniformly spread all across the screen.

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

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1					1				
Row=2				3	2	3			
Row=3			5	4	3	4	5		
Row=4		7	6	5	4	5	6	7	
Row=5	9	8	7	6	5	6	7	8	9

Data Representation

Although it is logical, we can exactly locate the position of each printable character on the display screen by knowing its uniquely identifiable **(row, column)** numeric paired values. **For example;** **(4, 3)** means printing numeric characters at the assumed screen location **4th row and 3rd column**.

(Row, Col) value pairs representation

	Col=1	Col=2	Col=3	Col=4	Col=5	Col=6	Col=7	Col=8	Col=9
Row=1					1 , 5				
Row=2				2 , 4	2 , 5	2 , 6			
Row=3			3 , 3	3 , 4	3 , 5	3 , 6	3 , 7		
Row=4		4 , 2	4 , 3	4 , 4	4 , 5	4 , 6	4 , 7	4 , 8	
Row=5	5 , 1	5 , 2	5 , 3	5 , 4	5 , 5	5 , 6	5 , 7	5 , 8	5 , 9

Analysis of output

row	1								
col	1	2	3	4	5	6	7	8	9
output					1				

For row = 1, $5 \leq Col \leq 5$

row	2								
col	1	2	3	4	5	6	7	8	9
output				3	2	3			

For row = 2, $4 \leq Col \leq 6$

row	3								
col	1	2	3	4	5	6	7	8	9
output			5	4	3	4	5		

For row = 3, $3 \leq Col \leq 7$,

row	4								
col	1	2	3	4	5	6	7	8	9
output		7	6	5	4	5	6	7	

For row = 4, $2 \leq Col \leq 8$,

row	5								
col	1	2	3	4	5	6	7	8	9
output	9	8	7	6	5	6	7	8	9

For row = 5, $1 \leq Col \leq 9$

COLUMN LEVEL REPETITIVE ACTION

Row level Repetitive Action			Expected Output		
row	Column level Repetitive Action				
	<i>lower_limit <= col <= upper_limit</i>				
	<i>lower_limit</i>	<i>upper_limit</i>			
1	5	5	1		
2	4	6	3	2	3
3	3	7	5	4	3
4	2	8	7	6	5
5	1	9	9	8	7

Both the *lower_limit* & *upper_limit* are not consistent for all the rows. We need to develop consistent expressions for both *upper_limit* and *lower_limit*, which should suit n number of rows. The other remaining variables are row, totalRows=5. Let's try to generalize using these variables.

Column level Repetitive Action					
row	<i>lower_limit <= col <= upper_limit</i>				
	<i>lower_limit</i>		<i>upper_limit</i>		
	<i>default</i>	1	2	<i>default</i>	1
1	5	totalRows	totalRows +1 - 1	5	totalRows
2	4	totalRows -1	totalRows +1 - 2	6	totalRows + 1
3	3	totalRows -2	totalRows +1 - 3	7	totalRows + 2
4	2	totalRows -3	totalRows +1 - 4	8	totalRows + 3
5	1	totalRows -4	totalRows +1 - 5	9	totalRows + 4

Column level Repetitive Action				
row	lower_limit <= col <= upper_limit			
	lower_limit		upper_limit	
	default	3	default	3
1	5	totalRows +1 - row	5	totalRows + row - 1
2	4	totalRows +1 - row	6	totalRows + row - 1
3	3	totalRows +1 - row	7	totalRows + row - 1
4	2	totalRows +1 - row	8	totalRows + row - 1
5	1	totalRows +1 - row	9	totalRows + row - 1

The lower_limit & upper_limit are consistent for all the rows. It's now generalized for any number of rows, which gives the exact position where to print numeric value. So our If-condition becomes;

if-condition	lower_limit <= col <= upper_limit
Mathematical expression	(totalRows +1 - row) <= col <= (totalRows + row - 1)
syntax	if (col >= ((totalRows - row)+ 1) and col <= ((totalRows + row)- 1)):

When the if-condition is satisfied, it will print the numeric value at that position at a particular row or else will print the space character. We can observe two things from the given output, which can be given as;

- a) The values at each row start with the $(2 * \text{row} - 1)$ value and goes on decreasing till row value and further increases till it reaches the $(2 * \text{row} - 1)$ value. The value is basically oscillating like a pendulum of a wall clock. Let's choose a variable **num** that represents a $(2 * \text{row} - 1)$ value every time a new row starts. The value of variable num then oscillates to give output in a single row. It can be shown by the following table.

Table for **num** variable starting at $(2 * \text{row} - 1)$ value.

row	num	col < totalRows	col = totalRows	col > totalRows
1	1		1	
2	2	3	2	3
3	3	5 4	3	4 5
4	4	7 6 5	4	5 6 7
5	5	9 8 7 6	5	6 7 8 9

col < totalRows	col >= totalRows
if(col < totalRows) num = $(2 * \text{row} - 1)$ value gets printed. num value get decreased by 1.	if(col >= totalRows) num value gets printed. num value get increased by 1 till $(2 * \text{row} - 1)$

b) The column counts are more than the **totalRows** count other than the first row. The **totalRows** value is 5. The column count is different on each row. The upper_limit of inner **for-loop** need to be generalized.

loop-condition	<code>lower_limit <= col <= upper_limit</code>
Mathematical expression	$1 \leq col \leq 9$ $1 \leq col \leq totalRows + 4$ $1 \leq col \leq totalRows + (totalRows - 1)$ $1 \leq col \leq 2 * totalRows - 1$
syntax	<code>for col in range(1, 2 * totalRows):</code>

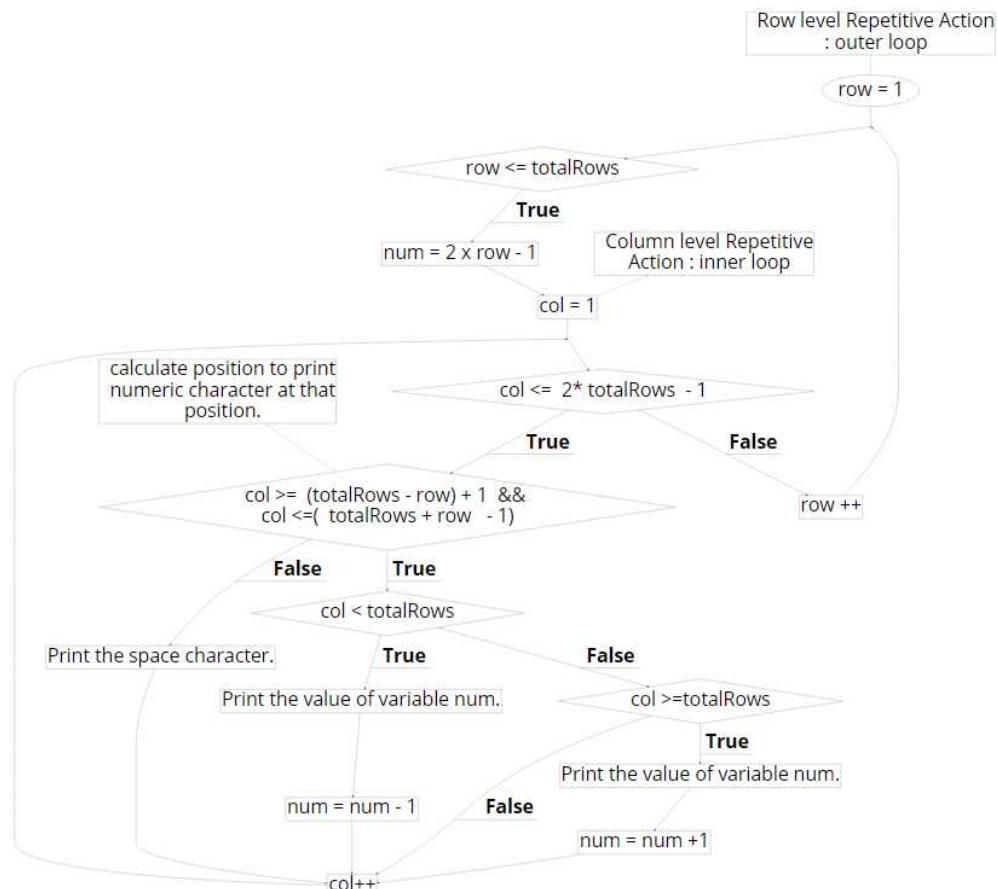
Let's verify the upper limit for col variables or total columns which will be used to display each row

for totalRows = 5 ,the col ≤ 9 equals ($2^5 - 1$)

for totalRows = 4 ,the col ≤ 7 equals ($2^4 - 1$)

So this generalization to calculate total columns for a given row value is satisfactory.

The flowchart can be given as follows.



Let write the program based on our findings and flow-chart.

NumberPatternEEE.py

```
totalRows = 5 # number of rows to display

# Row level Repetitive Action :
for row in range(1, totalRows):
    num = 2 * row - 1

        # Column level Repetitive Action :
    for col in range(1, 2 * totalRows):

        # calculates position to print numeric character
        if (col >=((totalRows - row)+ 1) and col <=((totalRows + row)- 1)):
            # print a number
        if (col < totalRows):
            print(num , end ="")
        num = num - 1
        elif (col >= totalRows):
            print(num, end ="")
        num = num + 1

    else:
        print("", end ="")# print 2 spaces

        #Ends inner loop

    # Move to next row
print()

#Ends outer loop
```

Output

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

ProblemGBBB2:

Modify program NumberPatternBBB2.py to show the following output.

```
1
2
3
4
5
```

ProblemGBDD1:

Modify program NumberPatternDDD1.py to show the following output.

```
9
7
5
3
1
```

<p>ProblemGBEEE:</p> <p>Modify program NumberPatternEEE.py to show the following output.</p> <pre> 1 3 3 5 5 7 7 9 9 </pre>	<p>ProblemGBEEE1:</p> <p>Modify program PatternEEE.py to show the following output.</p> <pre> 1 3 2 3 5 3 5 7 4 7 9 5 9 </pre>
<p>ProblemGB. Write programs to generate the following outputs.</p>	
<pre> 1>=1 2>=1 2>=2 3>=1 3>=2 3>=3 4>=1 4>=2 4>=3 4>=4 5>=1 5>=2 5>=3 5>=4 5>=5 </pre>	<pre> <<~><<~>><<~>><<~>><<~>><<~>><<~>><<~>> <<~><<~>><<~>><<~>><<~>><<~>><<~>> <<~><<~>><<~>><<~>><<~>><<~>><<~>> <<~><<~>><<~>><<~>><<~>><<~>> <<~><<~>><<~>><<~>><<~>> </pre>
<pre> 2 3 4 4 5 6 5 6 7 8 6 7 8 9 10 7 8 9 10 11 12 </pre>	<pre> 1 2 1 3 2 1 4 3 2 1 5 4 3 2 1 6 5 4 3 2 1 7 6 5 4 3 2 1 </pre>
<pre> 0 1 0 2 1 0 3 2 1 0 4 3 2 1 0 5 4 3 2 1 0 </pre>	<pre> 1<2 1<3 1<4 1<5 1<6 1<7 1<8 1<9 1<10 2<3 2<4 2<5 2<6 2<7 2<8 2<9 2<10 3<4 3<5 3<6 3<7 3<8 3<9 3<10 4<5 4<6 4<7 4<8 4<9 4<10 5<6 5<7 5<8 5<9 5<10 6<7 6<8 6<9 6<10 7<8 7<9 7<10 8<9 8<10 9<10 </pre>

Strings

It is a very common coding activity to modify or manipulate text at runtime like displaying text or part of the text in some way, dynamically creating web-page URL addresses or file paths, extracting specific information from a text like many developers use web-scraping tools especially to extract text from web-pages. Every programming language provides in-built string libraries for string manipulation. We will experiment with some of the commonly used in-built string methods or functions.

String concatenation

We can output strings by joining one another using the in-built `join()` method or using the `+(plus)` concatenation operator. We had already used the concatenation operator `(+)` many times in our early exercises.

```
name1 ="Steve"  
name2 ="Jobs"  
"".join([name1, name2])
```

It's a wrong assumption that both variables `name` and `name1` will give the same output.

It's one of the most common mistake which many of beginner programmers does .

StringConcat.py

```
name1 ="Steve"  
name2 ="Jobs"  
  
print("X -> Common mistake while using join() and concatenation operator +")  
# join() method is used to combine  
# the two given strings with space("") as a separator.  
# no assignment operator is being used to store the final result  
  
# common mistake  
"".join([name1, name2])  
print(name1)  
name1 +" "+ name2  
print(name1)  
print("Correct use of join() and concatenation operator +")  
# don't forget to use assignment operator just like  
# we had used in case of numeric calculations  
name3 = "".join([name1, name2])  
print(name3)  
  
name4 = name1 +" "+ name2  
print(name4)
```

Output

```
X -> Common mistake while using join() and concatenation operator +  
Steve  
Steve  
Correct use of join() and concatenation operator +  
Steve Jobs  
Steve Jobs
```

Substrings or Slicing

Choosing **start** and **end** as some positions in a given string. To get a copy of a given string based on the **start** and **end** positions in a given string is commonly known as substring or slicing. One substring could be a part of another substring. Let **string** be the name of a variable that stores a string content "Programming," please notice the following results;

```
string ="Programming"
```

Character	P	r	o	g	r	a	m	m	i	n	g
Index	0	1	2	3	4	5	6	7	8	9	10

string[start: end]→ substring made up of characters from index **start to end-1**.

string[0: 7]→ substring made up of characters from index **0 to 6** is "Program."

string[: end]→ substring made up of characters from index **start to end-1**.

string[: 3]→ substring made up of characters from index **0 to 2** is "Pro."

string[-1]→ For getting the last character of a given string which is "g."

string[:-2]→ For getting 2nd last character of a given string which is "n."

string[start:]→ substring made up of all characters from index **start to end of the string**.

string[3:]→ substring made up of all characters from index **start to end of the string** is "gramming."

string[start: end:step]→ substring made up of characters from index **start to end-1** that includes the character from the current character's index + step positioned character. By default, the 'step' value is 1.

Hence, **string[start: end:1]** and **string[start: end]** mean the same, follows the default behavior.

Note:- **len(string)**→ It is an inbuilt function that returns the length of the given string.

SubStringResults1.py

```
string ="Programming"
subStr =""
LENGTH = len(string)

print("Length of the String =", LENGTH)
print()
print("substring demo:"+string[start : end ])
print()
for index in range(0, LENGTH + 1):
    subStr = string[0: index]
    print("string[ 0 : ",index, "]=",+subStr)
```

Output

Length of the String = 11	string[0 : 5]= Progr
substring demo:string[start : end]	string[0 : 6]= Progra
string[0 : 0]=	string[0 : 7]= Program
string[0 : 1]= P	string[0 : 8]= Programm
string[0 : 2]= Pr	string[0 : 9]= Programmi
string[0 : 3]= Pro	string[0 : 10]= Programmin
string[0 : 4]= Prog	string[0 : 11]= Programming

SubStringResults2.py

```
string ="Programming"
subStr =""
LENGTH = len(string)
print("Length of the String =", LENGTH)
print()
print("substring demo:"+string[start : end ])
print()
for index in range(0, LENGTH + 1):
    subStr = string[index : LENGTH ]
    print("string[", index,":",LENGTH ,"]="+ subStr)
```

Output

Length of the String = 11	string[5 : 11]= amming
substring demo:string[start : end]	string[6 : 11]= mming
string[0 : 11]= Programming	string[7 : 11]= ming
string[1 : 11]= rogramming	string[8 : 11]= ing
string[2 : 11]= ogramming	string[9 : 11]= ng
string[3 : 11]= gramming	string[10 : 11]= g
string[4 : 11]= ramming	string[11 : 11]=

SubStringResults3.py

```
string ="Programming"
subStr =""
LENGTH = len(string)
end =-1
print("Length of the String =", LENGTH)
print()
print("substring demo:"+string[start : end ])
print()
# reducing characters from end of the given string to extract the substring.
for index in range(0, LENGTH + 1):
    end =-1 *(index + 1) if index > 0 else -1
    subStr = string[0 : end ]
    print("string[", 0,":",end ,"]="+ subStr)
```

Output

Length of the String = 11	string[0 :-6]= Progr
substring demo:string[start : end]	string[0 :-7]= Prog
string[0 :-1]= Programmin	string[0 :-8]= Pro
string[0 :-2]= Programmi	string[0 :-9]= Pr
string[0 :-3]= Programm	string[0 :-10]= P
string[0 :-4]= Program	string[0 :-11]=
string[0 :-5]= Progra	string[0 :-12]=

```
SubStringResults4.py
string ="Programming"
LENGTH = len(string)
print("Length of the String =", LENGTH)
print()
print("substring demo:"+string[start : end ])
print()
# reducing characters from both the ends of the given string to extract the
# substring.
for index in range(0, LENGTH + 1):
    end = -1 * (index + 1) if index > 0 else -1
    subStr = string[index : end ]
    print("string[", index, ":" ,end , "]=" + subStr)
```

Output

```
Length of the String = 11

substring demo:string[start : end ]

string[ 0 :-1 ]= Programmin
string[ 1 :-2 ]= rogrammi
string[ 2 :-3 ]= ogramm
string[ 3 :-4 ]= gram
string[ 4 :-5 ]= ra
string[ 5 :-6 ]=
string[ 6 :-7 ]=
string[ 7 :-8 ]=
string[ 8 :-9 ]=
string[ 9 :-10 ]=
string[ 10 :-11 ]=
string[ 11 :-12 ]=
```

Problem: Modify the program **SubStringResults4.py** so that it should only produce a substring that is not blank. The empty substrings should not get printed for the modified version of the program

SubStringResults4.py.

Expected Output	Blank substrings
<pre>Length of the String = 11 substring demo:string[start : end] string[0 :-1]= Programmin string[1 :-2]= rogrammi string[2 :-3]= ogramm string[3 :-4]= gram string[4 :-5]= ra</pre>	<pre>string[5 :-6]= string[6 :-7]= string[7 :-8]= string[8 :-9]= string[9 :-10]= string[10 :-11]= string[11 :-12]=</pre>

Patterns by Single Looping

We can output some of the patterns that we have earlier achieved using a nested loop (one for-loop within another for-loop) by single **for-loop** using the **join()** method.

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

Pattern A1

StringPatternA1.py

```
totalRows = 5 # number of rows to display  
strPlus = ""# initialize with blank string.  
strJoin = ""# initialize with blank string.  
  
print("Print using operator +")  
print()  
for row in range(1, totalRows + 1):  
    # Column level Repetitive Action : without for-loop  
    # append to the existing string using concatenation operator +  
    strPlus = strPlus + "*"  
    # Print the updated string stored in variable:line  
# and move to the next row  
    print(strPlus)  
  
print()  
print("Print using in-built function join()")  
print()  
for row in range(1, totalRows + 1):  
    # Column level Repetitive Action : without for-loop  
    # append to the existing string using single space("") character  
# as a separator.  
    strJoin = "".join(["*",strJoin])  
    # Print the updated string stored in variable:line  
# and move to the next row  
    print(strJoin)
```

Output

```
*
```



```
**
```



```
***
```



```
****
```



```
*****
```

```

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

```

Pattern G1

StringPatternG1.py

```

totalRows = 5 # number of rows to display
line = ""# initialize with blank string.

for row in range(1, 2 * totalRows):
    # Column level Repetitive Action :
    # when row <= totalRows, build the string using '*' character.
    if (row <= totalRows):
        line = line + "*"
    else:
        # when row > totalRows, trim the string by single '*' from the last
        line = line[0:-1]
print(line)

```

Output

```

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

```

ProblemSPG1: Write an appropriate logical expression for the start attribute when **row > totalRows**. The string variable **line = line[start:]** has only a single attribute, “**start.**” Write the appropriate logical expression for the attribute **start**.

StringPatternG2.py

```

totalRows = 5 # number of rows to display
line = ""# initialize with blank string.
for row in range(1, 2 * totalRows):
    # Column level Repetitive Action :
    # when row <= totalRows
    if (row <= totalRows):
        line = line + "*"
    else:
        # when row > totalRows
        line = line[start:]
print(line)

```

Output

```

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

```

ProblemSPG2: In the program `StringPatternG1.py`, the star “*” characters are printed quite close to each other. In order to get better output, suppose an extra space added as shown in the following program. The output doesn’t come as per our expectations. What modifications in the following program should be applied so that we should be able to get the expected output?

StringPatternG3.py

```
totalRows = 5 # number of rows to display
line = ""# initialize with blank string.

for row in range(1, 2 * totalRows):
    # Column level Repetitive Action :
    # when row <= totalRows, build the string using '*' character.
    if (row <= totalRows):
        #index = index + 1
        line = line + "*"# a single space appended to '*'
    else:
        # when row > totalRows, trim the string by single '*' from the last
        line = line[0:-1]
print(line)
```

Output by this program

```
*
```

$$\begin{array}{c} ** \\ *** \\ **** \\ ***** \\ ***** \\ *** \\ *** \\ ** \\ * \end{array}$$

Expected Output

```
*
```

$$\begin{array}{c} ** \\ *** \\ **** \\ ***** \\ *** \\ ** \\ * \end{array}$$

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

NumberPattern AA1

StringPatternAA1.py

```
1  totalRows = 5 # number of rows to display
2  line = ""      # initialize with blank string.
3
4  for row in range(1, totalRows + 1):
5      # Column level Repetitive Action : without for-loop
6      line = line + row
7      print(line)
```

Output

```
Traceback (most recent call last):
  File "..\StringPatternAA1.py", line 6, in <module>
    line = line + row
TypeError: can only concatenate str (not "int") to str
```

From the above error, we can conclude that concatenate operator + does not give the expected results when operands are of different data types other than string data-type. In our case, variable: line is of string data-type, and variable: row is of type integer. Hence, the following programming statement throws the error while running the program.

```
line = line + row
```

We can use the in-built **function str()** to convert an integer value to a string. For example, to convert integer value 5 to string value "5", we need to implement the following code snippet.

```
# converting integer data-type to string data-type
row = 5
rowStr = str(row)
```

StringPatternAA1.py

```
1 totalRows = 5 # number of rows to display
2 line = ""      # initialize with blank string.
3
4 for row in range(1,totalRows + 1):
5     # Column level Repetitive Action : without for-loop
6     rowStr = str(row)# updated code
7     line = line + rowStr
8     print(line)
```

Output

```
1
12
123
1234
12345
```

Since strings can be given in single and double quotes, we shall try inbuilt method **chr()** to print the given pattern. We have to consider Unicode of integers also which points to following hint;

```
chr(48 + 1)='1', chr(48 + 2)='2', chr(48 + 3)='3', chr(48 + 4)='4'
```

StringPatternAA1.py

```
1 totalRows = 5 # number of rows to display
2 line = ""      # initialize with blank string.
3
4 for row in range(1,totalRows + 1):
5     # Column level Repetitive Action : without for-loop
6     line = line + chr(48 + row)+" "# a space for formatting the output
7     print(line)
8
```

Output

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

```

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

```

NumberPatternGG1

StringPatternGG1.py

```

totalRows = 5 # number of rows to display
line = ""# initialize with blank string.
for row in range(1, 2 * totalRows):
    # Column level Repetitive Action : without for-loop
    # when row <= totalRows, build the string using integer values.
    if (row <= totalRows):
        line = line + str(row)+" "# a single space appended
    else:
        # when row > totalRows, trim the string from the end
        line = line[0:-2]
    print(line)

```

Output

```

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

```

Accessing each string's character

In order to know the composition of string, we can use string name followed by [index]. The “index” means the position of a character in a given string. Let us suppose “Program” is a string; then we can give it in a program as;

string = "Program"

What shall we expect from the string [index]? It can be given as follows:

string[index]	Character at index position
string[0]	P
string[1]	r
string[2]	o
string[3]	g
string[4]	r
string[5]	a
string[6]	m

Let's confirm our understanding using a given program.

```
StringCharacters.py
string ="Program"
length = len(string)

for index in range (0, length):

    output ="string["+ str(index)+"]="+ string[index]

    print(output)
```

Output

```
string[ 0]= P
string[ 1]= r
string[ 2]= o
string[ 3]= g
string[ 4]= r
string[ 5]= a
string[ 6]= m
```

ZERO is a valid index value in the **string[0]**, a character accessing method, and it points to the first positioned character in a given string.

Exercise: Write a program for the following staircase look alike picture.

```
P_|_P
r_|_r
o_|_o
g_|_g
r_|_r
a_|_a
m_|_m
m_|_m
i_|_i
n_|_n
g_|_g
```

Staircase.py

```
string ="Programming"
stairStep =[""]

for index in range(0,len(string)):
    stairStep.append("_")# append strings
    ch = string[index]
    print(ch,end="")
    for i in range(len(stairStep)):
        print(stairStep[i],end="")
        print("|",end="")
    for i in range(len(stairStep)):
        print(stairStep[i],end="")
    # Print last char and then move to the next row
    print(ch)
```

Problem A: Rewrite the program **Staircase.py** without using the **join()** method.
The first step will be to write the logic trace table with appropriate rows & columns.
From the analysis of the logic trace table, a programmer can be able to find out a generalized way to get the solution to print series of "_" characters. A different character could also be taken like "*", "=" or "#" to get the count. It could be for any other required analysis.

Problem B. If there is a variable `national_id` in a program and you want to assign a 5-digit length value. For example, **78901**.

It should output in digits made up of "##" characters.

```
# # # #      # # # # #      # # # #      # # # #      #
#      #      #      #      #      #      #      #
#      # # # # #      # # # #      #      #      #
#      #      #      #      #      #      #      #
#      # # # # #      #      # # # #      # # # # #
```

Write a generic program to print any such 5-digit number.

Problem C: Generate the following output for a given string "Programming"

```
P
Pr
Pro
Prog
Progr
Progra
Program
Programm
Programmi
Programmin
Programming
Programmin
Programmi
Programm
Program
Progra
Progr
Prog
Pro
Pr
P
```

Problem1. Print the following output using in-built string functions.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1											P										
2								r	P	r											
3								o	r	P	r	o									
4								g	o	r	P	r	o	g							
5								r	g	o	r	P	r	o	g	r					
6								a	r	g	o	r	P	r	o	g	r	a			
7								m	a	r	g	o	r	P	r	o	g	r	a	m	
8								m	m	a	r	g	o	r	P	r	o	g	r	a	m
9								i	m	m	a	r	g	o	r	P	r	o	g	r	a
10								n	i	m	m	a	r	g	o	r	P	r	o	g	r
11	g	n	i	m	m	a	r	g	o	r	P	r	o	g	r	a	m	m	i	n	g

Problem2. Print the following output using in-built string functions.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1											P										
2									r		r										
3									o				o								
4									g				g								
5									r				r								
6									a				a								
7								m					m								
8								m					m								
9								i					i								
10								n					n								
11	g																				g
12		n																			n
13			i																	i	
14				m															m		
15					n														n		
16						a												a			
17							r										r				
18								g					g								
19									o				o								
20									r				r								
21										P											

Problem3. Please verify the wavelength of your logical thinking capabilities. Write a generalized program so that for different size=1,2,3 inputs, the size of the wave should get doubled or triple. You can write a separate program for each different type of wave.

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

SIZE = 1

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

SIZE = 2

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

SIZE = 1

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

SIZE = 1

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

SIZE = 1

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

SIZE = 1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	1	1	1	1			0	0	0	0			1	1	1	1						
2	1		1				0			0			1			1						
3	1		1				0			0			1			1						
4	1		1				0			0			1			1			1			
5			1				1			0			0			1			1			
6			1				1			0			0			1			1			
7			1	1	1	1				0	0	0	0			1	1	1	1			

SIZE = 1

Problem 4. Write a program for each of the following different wave designs.

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

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1		<	>					<	>					<	>							
2	<			>				<		>			<			>						
3	<		>				<		>			<			>			>				
4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
5			<				>			<			>			<			>			
6			<				>			<			>			<			>			
7				<	>					<	>					<	>					

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1		<	>					<	>					<	>							
2	<			>				<		>			<			>						
3	<		>				<		>			<			>			>				
4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
5			[]			[]			[]		[]			
6			[]			[]			[]		[]			
7			[]						[]						[]				

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1		<	>					[]					<	>							
2	<			>				[]			<			>						
3	<		>				<		>			<			>			>				
4	<		>				<		>			<			>			>				
5			<				>			[]			<			>				
6			<				>			[]			<			>				
7				<	>					[]						<	>				

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1		3	4					15	16					27	28							
2	2			5			14			17			26			29						
3	1			6			13			18			25			30						
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5				7			12			19			24			31			36			
6				8			11			20			23			32			35			
7				9	10					21	22					33	34					
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1		1	2					5	6					9	10							
2	1		2				5			6			9			10						
3	1		2				5			6			9			10						
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5			3				4			7			8			11			12			
6			3				4			7			8			11			12			
7				3	4					7	8					11	12					

List

A list is a **mutable** collection commonly used to store data-items of the same data type, such as integers or strings. It can also store data-items of mixed data types. You can add, remove, and modify the data-items or elements of a list.

An example of a list of numbers:

```
list_number = [1,2,3,4,5]
```

To create a list

We can create a list by comma separating each data items inside square brackets.

For instance:

```
weekdays_num = [1,2,3] # integer data-items
```

```
weekdays_name = ["mon", "tue", "wed"] # string data-items
```

```
mixed_list = ["mon", 1, 2, "tue"] # mixed of integer and string data-items
```

To count the number of items in the list

Since the list is a mutable collection, it may grow or decrease and may call for identifying its size dynamically during the execution of a certain program. To count the number of elements in a given list, use built-in `len()` function. For example, let's create a list of numbers, and use this function.

```
num_list = [1,2,3]
```

```
length = len(num_list)
```

```
print(length)
```

Result:

3

To access a list item

We can access to a particular element from a list by using its index value.

List Index

The first element in the list has an index of zero. And the second element has an index of one, and so on. It means that list indexing starts from zero value. To get a access to list element, use an index operator `[]` and mention element's index value as follows:

```
num_list = [1,2,3]
```

```
second_element = num_list[1]
```

```
third_element = num_list[2]
```

```
print(second_element)  
print(third_element)
```

Output:

2

3

Negative Index

It is possible to use the negative index values to access elements in a list. The value of index, -1 refers to the last element, -2 to the second last element, and so on.

Let's see an example of this as well:

```
num_list = [1,2,3]  
last_element = num_list[-1]  
second_last_element = num_list[-2]  
print(last_element)  
print(second_last_element)
```

Output:

3

2

Range of Elements

We can access not just one element at a time, but also to a range of elements on a list using `:` operator. For example, let's create a list of names on weekdays, and get the first three and then the last two:

```
weekdays = ["Mon", "Tue", "Wed", "Thus", "Fri"]  
first_three_days = weekdays[0:3]  
last_two_days = weekdays[3:5]  
print(first_three_days)  
print(last_two_days)
```

Output:

```
['Mon', 'Tue', 'Wed']  
['Thus', 'Fri']
```

To modify an item in the list

If you want to modify the specific element in a list, do the following two actions:

- First, access that element using `[]` operator with it's index.
- Second, using the assignment operator `=`, assign a value to it.

For example:

```
num_list = [1,2,3]
```

```
num_list[0] = 11
```

```
print(num_list)
```

Output:

```
[11, 2, 3]
```

To add items to a list

Given following are the four approaches to add an element to an existing list:

1. `append()` method
2. `insert()` method
3. `extend()` method
4. List concatenation: Use of `+` operator to concatenate multiple lists

Let's examine each of these approaches.

1. `append()` method

Using method `append()` to add an element to the end of the existing list :

```
num_list = [1, 2, 3]
```

```
num_list.append(44)
```

```
print(num_list)
```

Output:

```
[1, 2, 3, 44]
```

2. `insert()` method

Using method `insert()` we can add an element at a mentioned index in the existing list . We know that the first element's index is zero and the maximum value of the index will always be less than the size of the list.

```
num_list = [1,2,3]
```

```
num_list.insert(0,11)
```

```
print(num_list)
```

Output:

```
[11, 1, 2, 3]
```

3. extend() method

Use of extend() method is to add not just a single element but another list to the end of an existing list. For example:

```
num_list = [1,2,3]
num_list.extend([11, 22, 33])
num_list.extend("python")
print(num_list)
```

Output:

```
[1, 2, 3, 11, 22, 33, 'p', 'y', 't', 'h', 'o', 'n']
```

So why the result is [1, 2, 3, 11, 22, 33, 'p', 'y', 't', 'h', 'o', 'n']

and not [1, 2, 3, 11, 22, 33, "python"]

Reason: When the extend() method gets an argument as a string like “python” it loops through each character in a sequence and appends it to the list.

To have the result like this: [1, 2, 3, 11, 22, 33, 'python']

We need to pass “python” as a list having one string into the extend() method:

```
num_list = [1,2,3]
num_list .extend([11, 22, 33])
num_list .extend(["python"]) # notice the square brackets of the list
print(num_list )
```

The extend() method now loops through as a list element and adds it to the end of the list.

4. list concatenation using + operator

We can use + operator to join separate lists. For example;

```
num_list1 = [1, 2, 3]
num_list2 = [11, 22, 33]
num_list = num_list1 + num_list2
print(num_list)
```

Output:

```
[1, 2, 3, 11, 22, 33]
```

We can merge multiple lists and not stop to use + operator with two lists only.

For example;

```
one_digits = [1, 2, 3]
two_digits = [11, 22, 33]
three_digits = [111, 222, 333]
num_list = one_digits + two_digits + three_digits
print(num_list)
```

Output:

```
[1, 2, 3, 11, 22, 33 111, 222, 333]
```

To remove item from a list

Let assume that the list is not empty, and it has few elements to be eliminated from the list. There are four in-built facilities to remove an item from a non-empty list:

clear() method

pop() method

remove() method

del statement

1. clear() method

Use the clear() method to remove all the elements from a list.

For instance:

```
num_list = [1,2,3]
num_list.clear()
print(num_list)
```

Output:

```
[]
```

2. pop() method

We can remove a particular element using its value index by applying the pop() method. For example, try removing the first element of a list. The index of the first element is always zero and we can accept this index value as an argument in the pop method.

```
working_days = ["Sun", "Mon", "Tue"]
working_days.pop(0)
print(working_days)
```

Output:

```
['Mon', 'Tue']
```

3. remove() method

We can use remove() method to remove the first occurrence of a specific element in the list.

For example, let's remove the first occurrence of number 1 on the list:

```
num_list = [1,2,2,3,3]
num_list.remove(1)
print(num_list)
```

Output:

[2,2,3,3]

4. del statement

We can also remove elements from a list by specifying the value index using **del statement**:

```
num_list = [1,2,3]
del num_list[0]
print(num_list)
```

Output:

[2,3]

To reverse a list

We can use the reverse() method to reverse the order of list of elements:

```
num_list = [1,2,3]
num_list.reverse()
print(num_list)
```

Output:

[3,2,1]

To sort a list

It is important to sort randomly ordered data for observation purposes. You can sort a list by calling its sort() function. It will sort the list in ascending order by default if you don't provide any parameters. For example, you could list flight schedules in ascending order or alphabetize the passenger menu.

Example 1: Sort a list of numbers in ascending order:

```
numbers = [2,3,1]
numbers.sort()
print(numbers)
```

Output:

[1,2,3]

Example 2: Sort a list of numbers in descending order using method sort() with an argument, reverse=True.

```
numbers = [2,3,1]
numbers.sort(reverse=True)
print(numbers)
```

Output:

```
[3,2,1]
```

Example 3: Sort strings alphabetically in ascending order.

```
names = ['Home', 'Soup', 'Apple', 'Mango']
names.sort()
print(names)
```

Output:

```
['Apple', 'Home', 'Mango', 'Soup']
```

Example 4: You can sort strings in reversed alphabetic order by specifying reverse=True inside the sort() method:

```
names = ['Harry', 'Suzy', 'Al', 'Mark']
names.sort(reverse=True)
print(names)
```

Output:

```
['Suzy', 'Mark', 'Harry', 'Al']
```

Appendix-A: Installation of Python and project set-up

<https://www.javatpoint.com/how-to-install-python>

<https://www.javatpoint.com/python-example>

References:

1. <https://www.analyticsvidhya.com/blog/2013/06/art-structured-thinking-analyzing/>
2. JavaScript Programming Pattern Looping intelligence.[2019] Mohmad Yakub
<https://www.amazon.com/dp/1096466090/>

Acknowledgments

I wish to personally thank everyone that helped me to make my first book possible.

My sincere thanks to:

My family who supported me throughout its completion.

My youngest brother, Sadique S. Naikwadi, helped me tremendously in contributing technical reviews. He is a freelance developer and has always been famous for his programming skills since college times. He has worked on many different languages and never hesitates in trying programming heroics.

He is always reached out by college mates and beginner programmers for improving their programming skills. His suggestions helped me easily write technical writings and discussed in detail the issues of a struggling programmer. There was a time when programming was alien to him, and the set of exercises covered in this book was once a test-bed for him, and he was able to pick programming very nicely. With all these important factors, he is the best person to give me an honest technical review. Moreover, he runs a test-bed based on this book's exercises to help programmatically weak learners whenever he gets the chance.

I like to thank Mrs. Shabana Mursal and her family. She is a teacher, and she made me understand how different teaching styles are important to teach students of different mindsets.

Her inputs compelled me to put a completely new section which I would have never thought of including.

Thank you all so much for taking an interest in helping me.

*I am not teaching you
anything.*

*I just help you to explore
yourself.*

- Bruce Lee