Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Indian Institute of Technology Kanpur

Lecture: 10

This is the fifth lecture of week two, Introduction to Python. In the fourth lecture, we looked at array operations, matrix operations, and conditional statements. We will start with loops in this class.

Suppose you want to perform an operation repeatedly; that is when you use loops. For example, if you want to print the numbers from 1 to 100 one by one, you could write print(1), print(2), print(3), up to print(100). However, this is an inefficient way to do it. You would want to automate this process.

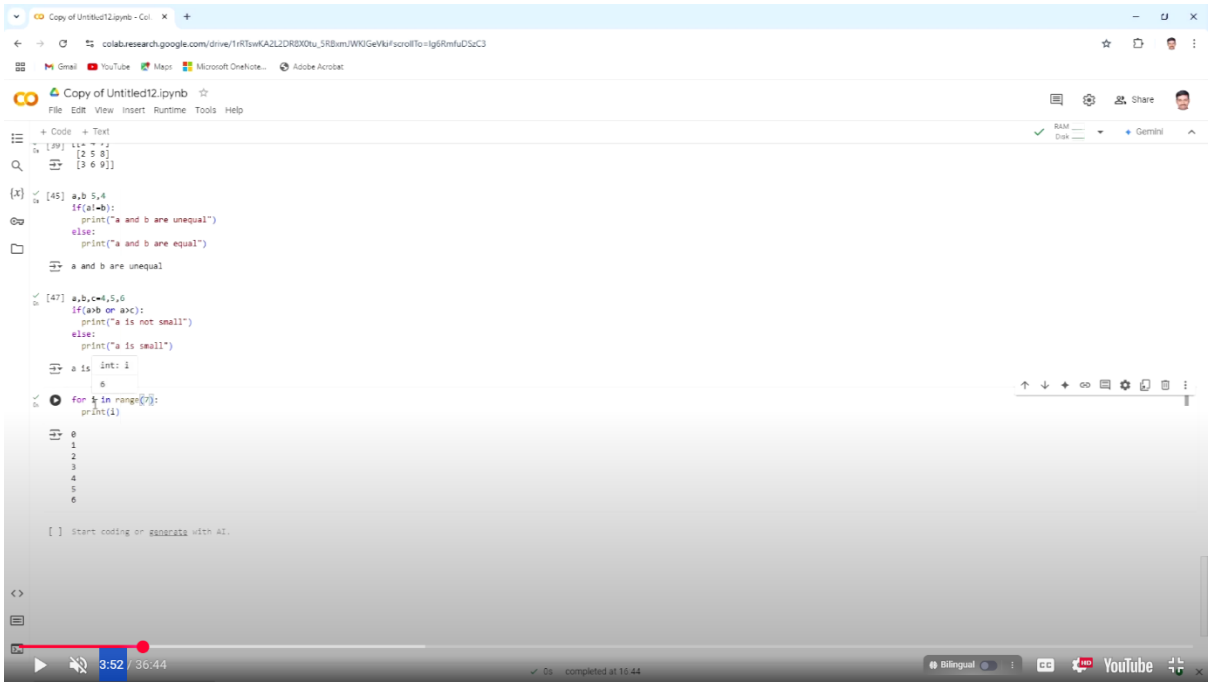
What you do is use something called loops. There are two commands that can be used for loops: for and while. The syntax for these two is slightly different.

Let us take the problem of printing numbers from 0 to 99. We will use a smaller range for demonstration. Writing for i in range(10): print(i) prints numbers from 0 to 9.

This statement means that the value of i is initialized to have values in the range of 10. The range(10) function generates numbers starting from 0 up to, but not including, 10, which is 9.

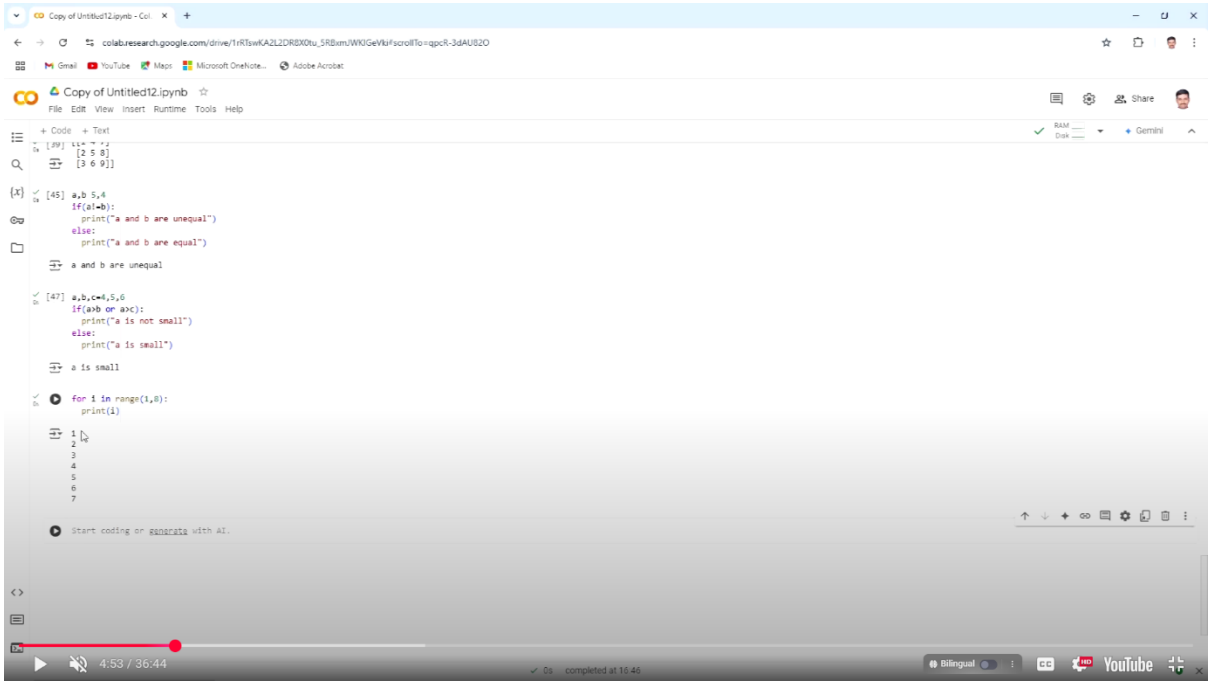
If you write for i in range(100): print(i), it will print from 0 to 99. If you change this to for i in range(7): print(i), this will print numbers from 0 to 6.

(Refer Slide Time 3:52)



The range() function generates values starting from 0 up to, but not including, the specified number. To print numbers starting from 1 instead of 0, you can modify the range parameters. For example, to print numbers from 1 to 7, you would use range(1, 8), which starts at 1 and stops before 8.

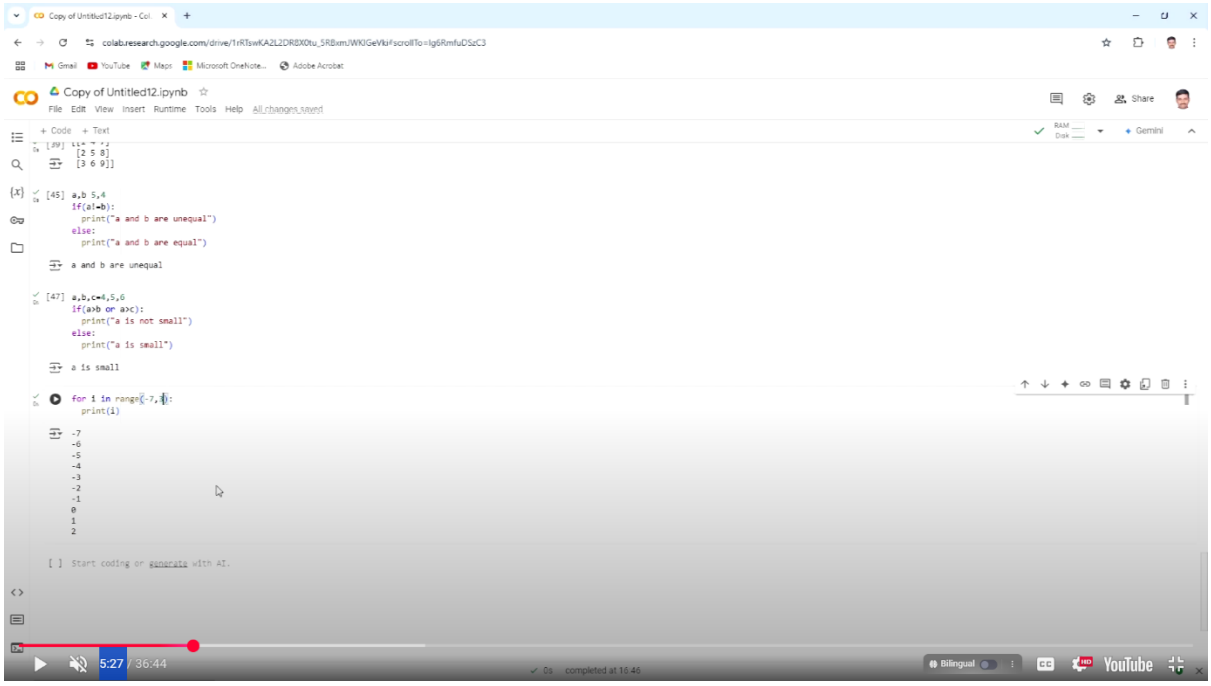
(Refer Slide Time 4:53)



You can specify a starting point and an endpoint in the range function.

For instance, to print numbers from -7 to 2, use range(-7, 3) since the function stops before the second parameter. This would print: -7, -6, -5, -4, -3, -2, -1, 0, 1, 2.

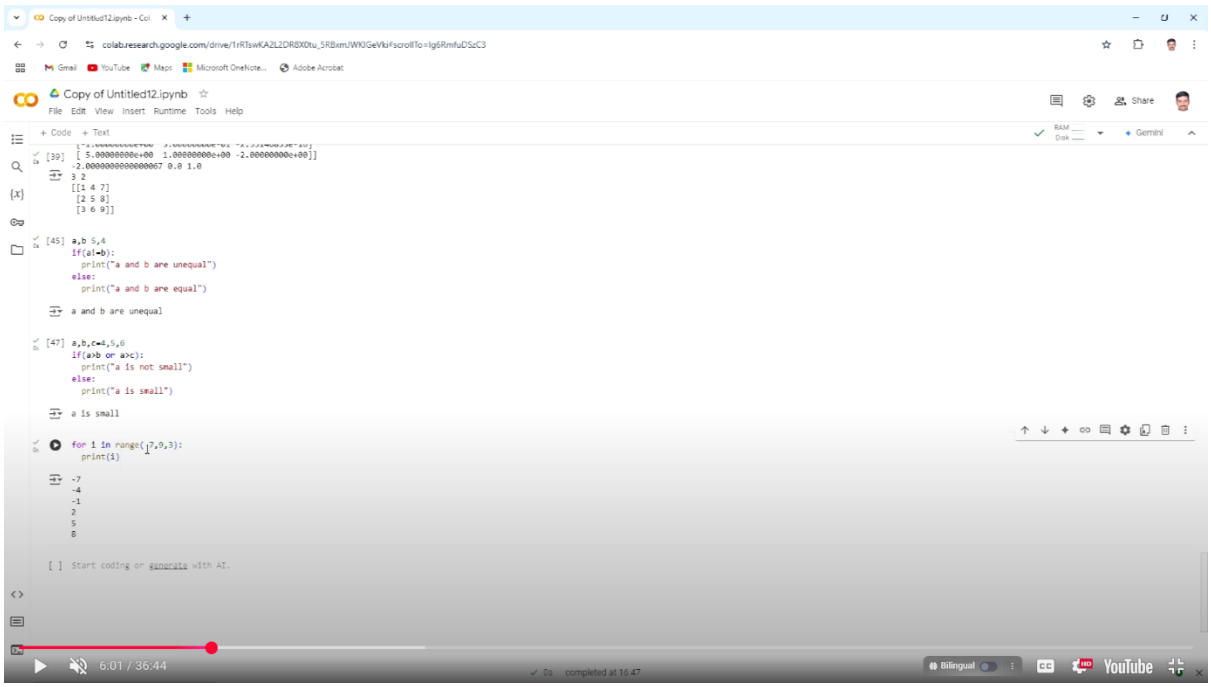
(Refer Slide Time 5:27)



To move in steps, add a third parameter for the step size.

For example, range(-7, 9, 3) starts at -7, ends before 9 (at 8), and moves in steps of 3, printing: -7, -4, -1, 2, 5, 8.

(Refer Slide Time 6:01)



The for loop is used when you want to repeat an operation a specific number of times, such as summing 100 numbers or printing 100 values.

The while loop is used when you want to continue looping until a certain condition is no longer met.

For example, to approximate $\log_2(40000)$ without using np.log2, you can repeatedly divide by 2 and count the divisions until the number is less than or equal to 1:

python

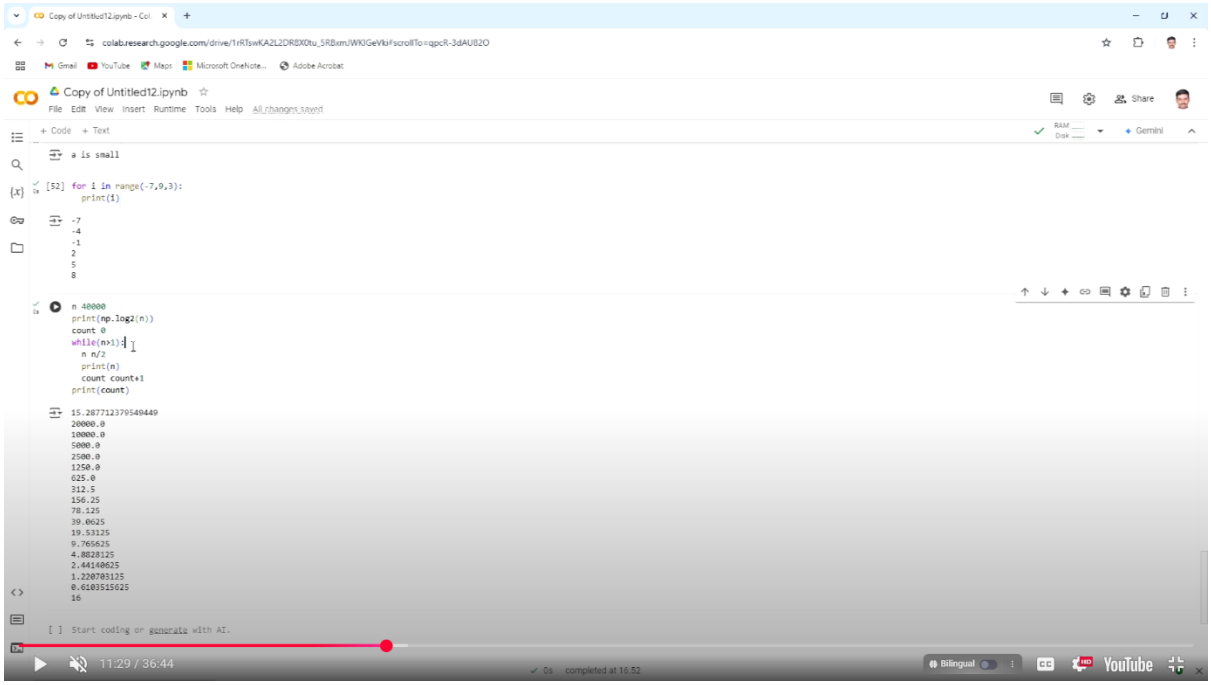
```
n = 40000

count = 0

while n > 1:

    n = n / 2

    count = count + 1

print(count)
```

This will output 16, indicating that 40000 divided by 2 sixteen times results in a value less than or equal to 1, showing that $\log_2(40000)$ is between 15 and 16.

To understand the process clearly, you can print the value of n at each step. Starting with n = 40000, each division by 2 produces the following sequence:

40000, 20000, 10000, 5000, 2500, 1250, 625, 312.5, 156.25, 78.125, 39.0625, 19.53125, 9.765625, 4.8828125, 2.44140625, 1.220703125, and finally 0.6103515625.

(Refer Slide Time 11:29)



After the 15th division, n is approximately 1.22.

After the 16th division, it becomes less than 1 (0.61), and the loop stops because the condition n > 1 is no longer true. This is a basic method to approximate the logarithm base 2 of a number.

The syntax for a while loop is:

python

while condition:

    # routine to perform

The loop continues executing the routine as long as the condition remains true. It exits only when the condition fails.

In contrast, a for loop is used when the number of iterations is known or fixed (e.g., iterating over a range of numbers). A while loop is used when the number of iterations depends on a dynamic condition (e.g., until a value reaches a threshold).

Nested loops (loops within loops) are also possible, following the same indentation rules. Both for and while loops use a colon and indentation to define the block of code to be repeated.

The while statement, like if and for, concludes with a colon. The block of code to be repeated must be indented beneath it. The loop's scope is defined by this indentation; the loop is considered complete when the indentation level returns to that of the while statement.

We will now proceed to nested loops. A nested loop is a loop placed inside the body of another loop. The classic example used to illustrate this concept is manual matrix multiplication. While the @ operator in NumPy performs this operation efficiently, implementing it manually demonstrates the structure of nested loops.

Consider a matrix A of size m × n and a matrix B of size n × p. Their product, matrix C, will be of size m × p.

Each element c_{ij} in the resulting matrix C is computed by taking the dot product of the i-th row of matrix A and the j-th column of matrix B. This calculation is expressed by the formula:

$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + ... + a_{in}b_{nj}$

This can be written more concisely using summation notation:

$c_{ij} = \Sigma\ a_{ik}b_{kj}$ for k = 1 to n

To compute a single element c_{ij}, a loop is required to iterate through the index k and sum the products:

for k in range(n): c[i, j] = c[i, j] + a[i, k] b[k, j]

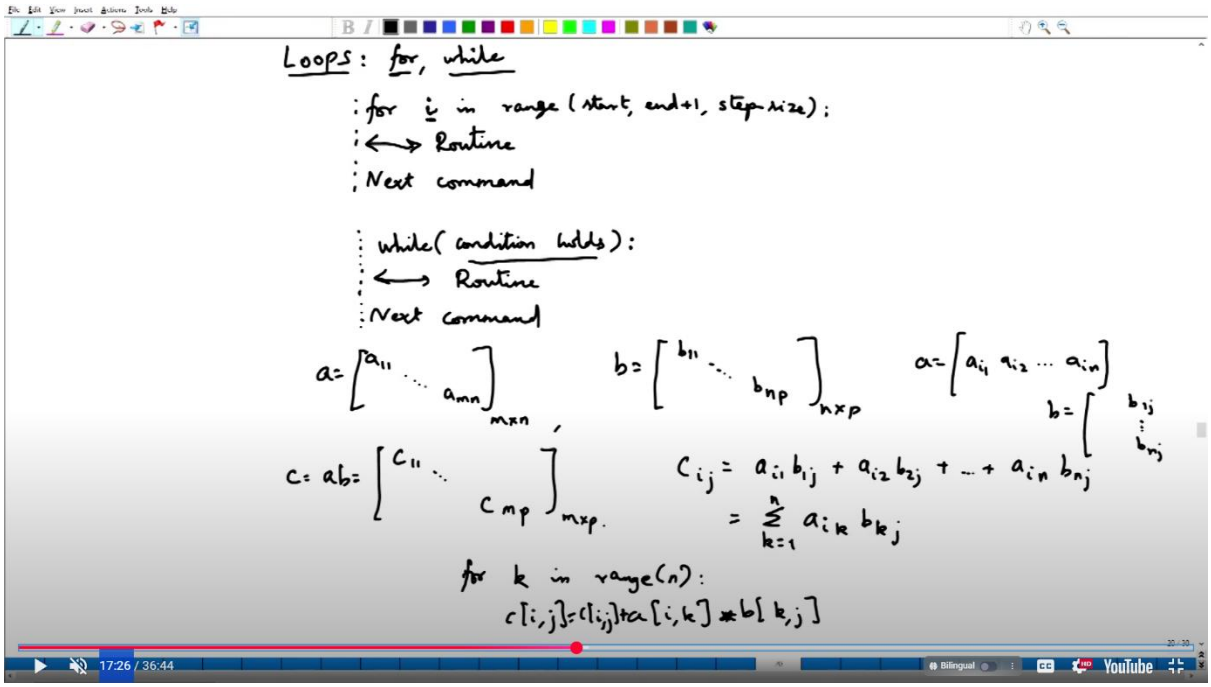
Before this summation, the element c[i, j] should be initialized to zero.

However, this calculation is only for one specific pair of indices i and j. To compute the entire matrix C, this process must be executed for every row index i (from 0 to m-1) and every column

index j (from 0 to p-1). This necessitates the use of two additional outer loops, resulting in a structure of three nested loops.

To demonstrate the nested loop structure for matrix multiplication, we will use predefined matrices. Let us assume we have two matrices, m and n. For this example, we will consider them to be 3x3 matrices.

(Refer Slide Time 17:26)



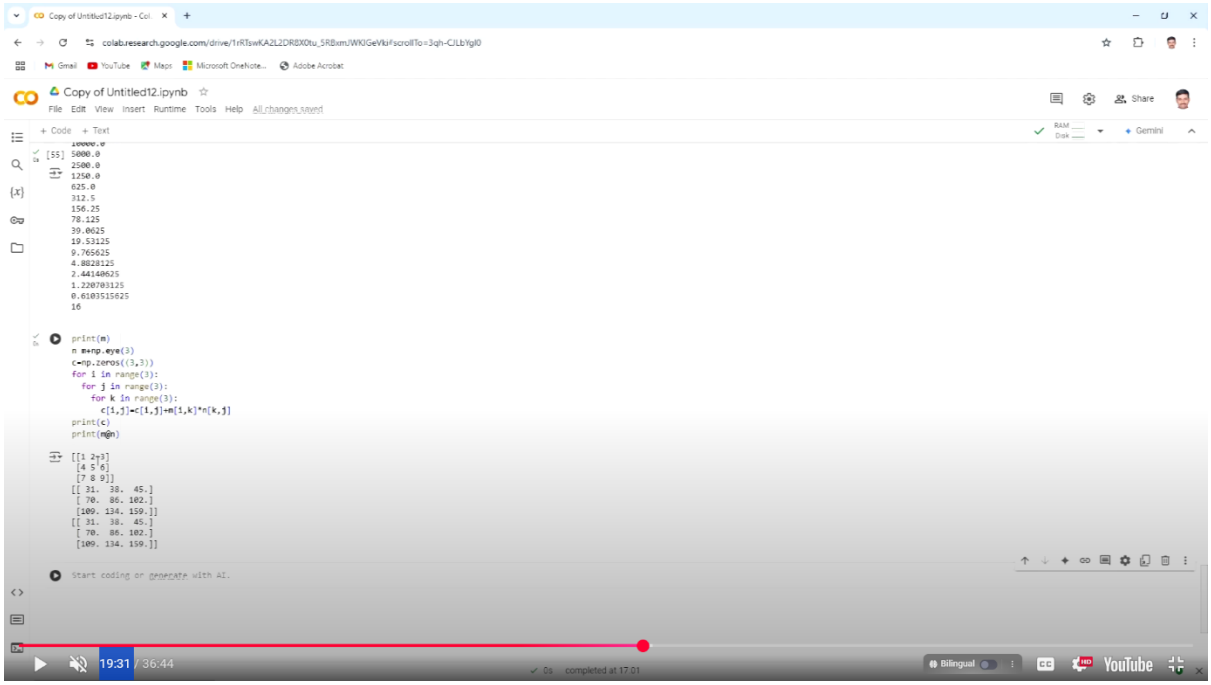
The code for manual multiplication is as follows:

Python

```
# Initialize the result matrix C with zeros

c = np.zeros((3, 3))

# Perform matrix multiplication using nested loops

for i in range(3):

    for j in range(3):

        for k in range(3):

            c[i, j] = c[i, j] + m[i, k]  n[k, j]

# Print the manually computed result and the result from the @ operator for verification

print("Manual multiplication (c):\n", c)

print("Built-in operator (m @ n):\n", m @ n)
```

The output will show that every element in the manually computed matrix c is exactly the same as the result from the built-in operator m @ n, confirming the correctness of the nested loop implementation.

(Refer Slide Time 19:31)



The structure of the nested loops is defined by their indentation. The entire block of code for the inner loops is contained within the outer loop. Specifically:

All commands for the j and k loops are within the first i loop.

All commands for the k loop are within the j loop.

Only the innermost summation statement is within the k loop.

The innermost statement c[i, j] = c[i, j] + m[i, k] n[k, j] is executed 3 3 3 = 27 times.

In general, for multiplying a matrix of size m × n with another of size n × p, this operation runs m × n × p times.

The outermost i loop runs m times, the middle j loop runs p times for each i, and the innermost k loop runs n times for each j.

This serves as an illustration of how to structure nested loops.

We will now move on to user-defined functions. Thus far, you have utilized many built-in commands from libraries like NumPy, such as sum, product, mean, median, dot, insert, and delete. These functions were defined by the developers of those libraries. User-defined functions are custom functions that you create yourself to perform specific tasks.

For example, suppose you need to sum the elements of a vector but are unaware of the np.sum function. You can define your own function, named add, to perform this operation.

To define a function, you use the def keyword. This keyword signals that you are defining a new function not already present in the language. The function is defined as follows:

Python

```
def add(v):
    count = 0
    for i in range(np.shape(v)[0]):
        count = count + v[i]
    return count
```

This function works by first initializing a variable count to zero. It then iterates through each index i of the vector v, adding the value at each index to count. Finally, the function returns the total sum stored in count.

To use this function, you first define a vector, for instance:

python

```
v = np.array([3, 5, 1, 9, 7])
```

Calling add(v) will then return the sum of its elements, which is 25 (3 + 5 + 1 + 9 + 7).

This demonstrates how to create a user-defined function. The syntax is straightforward: the def keyword is followed by the function's name and its parameters in parentheses. The function's body is indented beneath this declaration. The end of the indentation signifies the end of the function's definition. The return statement specifies the value that the function outputs.

The return statement is used to specify the value that a function sends back to the section of code that called it. This is a fundamental aspect of creating reusable and modular code.

User-defined functions are highly useful for encapsulating specific operations. For instance, you can define a function for a mathematical expression, such as:

python

```
def f(x):
    return x*np.exp(x) – x**2 + np.log(x)
```

This function, f(x), can then be evaluated for any value, for example, f(1.34689). It is important to use the correct NumPy functions: np.exp(x) for e^x and np.log(x) for the natural logarithm (ln x). For logarithms of other bases, use np.log10(x) for base 10 and np.log2(x) for base 2. Logarithms for other bases, such as base 4, are not directly available and must be computed using the change of base formula if needed.

These custom functions can be employed to verify mathematical properties programmatically. To verify if a function like $g(x) = e^{-x^2}$ is even (where $g(x) = g(-x)$), you can test it with a randomly generated value:

Python

```
x = np.random.randn()  # Generates a random number from a Gaussian distribution

print(g(x) == g(-x))   # Should print True for an even function
```

To check for an odd function, you would test if $f(x) == -f(-x)$.

To check for convexity using the midpoint property, you can test if the following condition holds for a function like $h(x) = x \log(x)$ and for random values x_0 and x_1:

$$h((x_0 + x_1)/2) \leq (h(x_0) + h(x_1)) / 2.$$

This test can be implemented in code to verify the property over multiple random pairs.

This test is focused on verifying midpoint convexity.

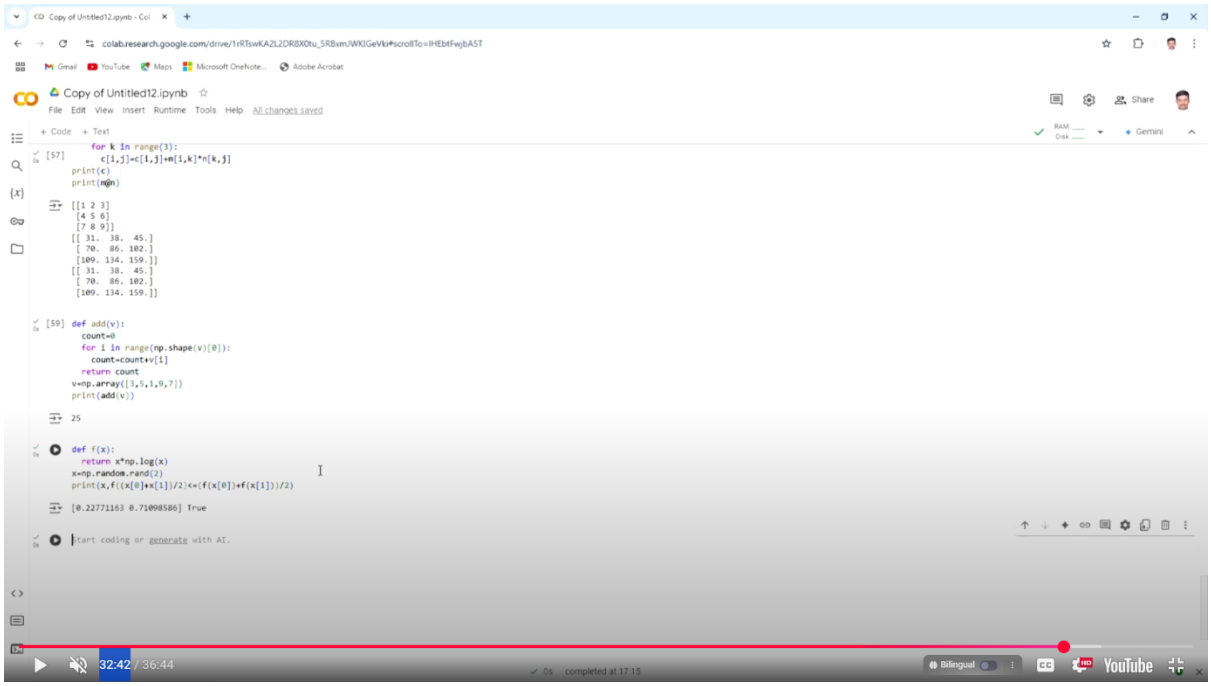
For the function $h(x) = x \log(x)$, which is defined for $x > 0$, we must ensure the input values are positive. The np.random.rand() function is suitable for this as it generates random numbers uniformly from the interval [0, 1).

For instance, with two randomly generated numbers, $x_0 = 0.46857$ and $x_1 = 0.26830$, the condition for midpoint convexity holds:

$$h((x_0 + x_1)/2) \leq (h(x_0) + h(x_1)) / 2$$

This test can be repeated multiple times with new random pairs, such as (0.91, 0.42) or (0.2, 0.71), to gather more evidence supporting the function's convexity.

(Refer Slide Time 32:54)

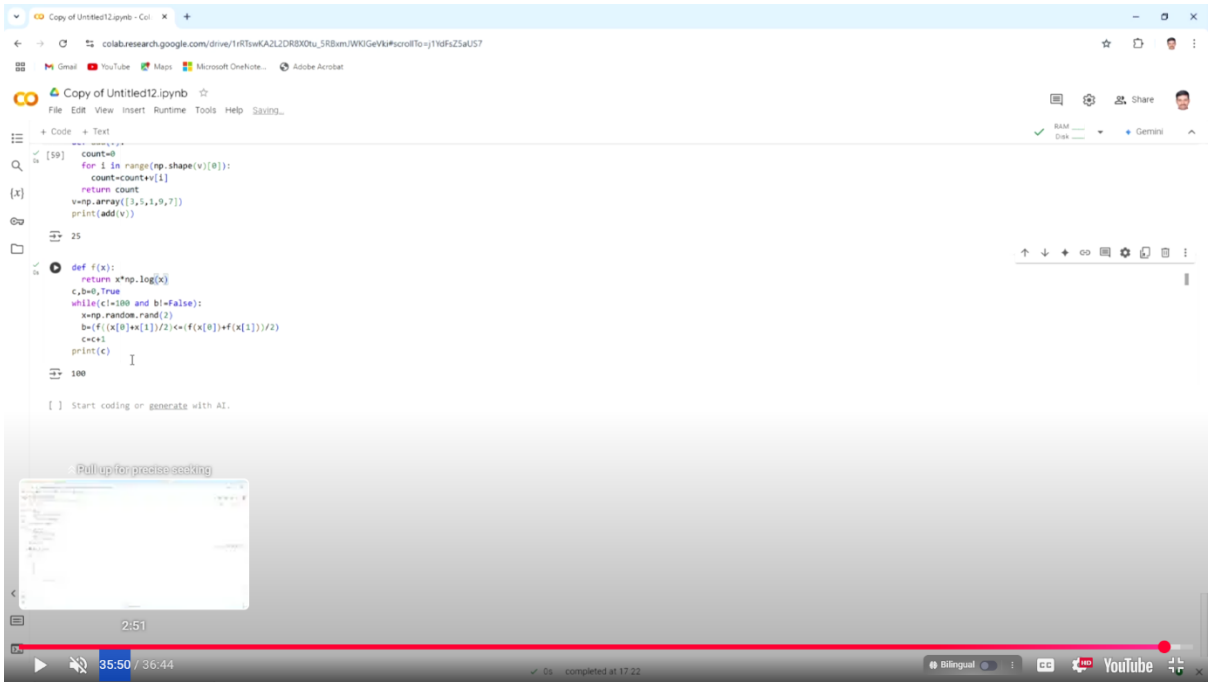


To perform a rigorous verification, this process can be automated to run 100 times using a while loop. The loop is designed to continue running while a counter c is less than 100 and a Boolean flag b remains True.

The flag b is initially set to True and would be set to False if the convexity condition ever failed. The counter c is incremented by one after each test.

After running this loop 100 times, if the flag b remains True, it indicates that the condition for midpoint convexity held for all 100 randomly selected pairs of values. This provides substantial empirical evidence that the function h(x) is convex on the interval (0, ∞).

(Refer Slide Time 35:50)



This concludes our exploration of user-defined functions. This week has covered fundamental Python concepts, starting with basic print statements and variable assignments.

We then progressed to array and matrix operations, followed by conditional statements and loops. The week finished with the creation of user-defined functions.

Next week, we will begin our study of optimization algorithms. Thank you.