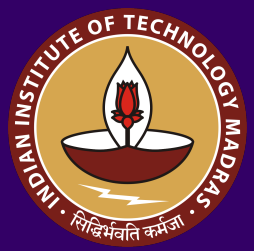
Scientific Computing & Advanced Programming Lab

SAURAV SAMANTARAY

Department of Mathematics
IIT Madras

Lecture-1



sauravsray@iitm.ac.in

Welcome!

- Slides will be available at ”https://sauravsray.github.io/MA-5105-Aug-26/”
- Codes corresponding to each lecture will be available at the same place.

Prerequisites and Demands

Prerequisites

- Some python programming experience
- Basic linear algebra and calculus

Demands

- honesty: don't cheat yourself !
- clocking numerous hours per week in writing / testing codes.
- interact with me and the teaching assistant

sauravsray@iitm.ac.in

Course Policies

Collaboration

- struggle first
- discuss specific doubts over the macroscopic problem
- don't copy anyone's code

LLMS Usage

- don't use to solve the problem
- may be only ask for clarifications

Assignments

- No assignments
- A lot of problem sets

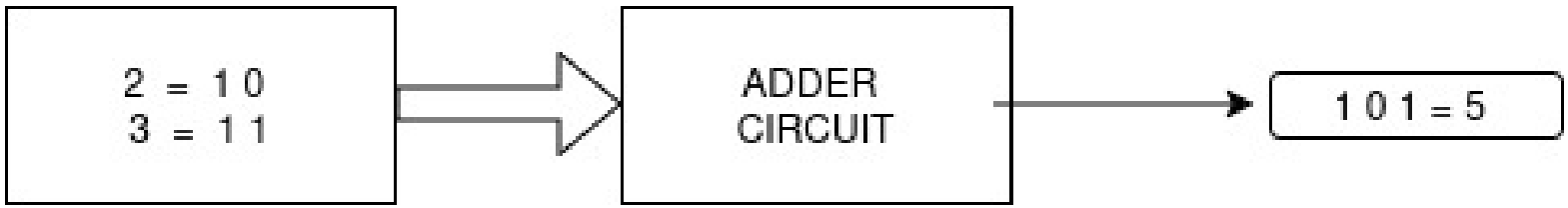
sauravsray@iitm.ac.in

Grading

- 4 surprise quizzes (fairly easy), 10 Marks each.
- 1 viva exam, 20 Marks.
- Class participation: 20 Marks.
- A final project worth 20 Marks.

Computing

- What do computers do ?
- How do they compute ?
  1. Data movement: read / store
  2. logical : Compare / And / Or
  3. add / subtract / Multiply / Divide / Increment / Decrement
  4. control flow: Jump to another instruction
- Example-1: Calculate 2 + 3:



- By itself not very impressive.

Computing

Example-2:

1. Read marks of next student;
2. Add score;
3. Compute Avg;
4. Store result;
5. Repeat steps for every student;
6. Print the final grade list.

- the computer doesn't learn any new operation for each of the instructions
- each operation is broken down to basic operation blocks, performed in some sequence

sauravsray@iitm.ac.in

What is programming?

- Programming is the process of organising basic operations into simple applications which combine to solve a much larger problem.
- can compute anything using 6 primitives (Turing)
- programming language provides a set of primitive operations
- create ”expressions” which are complex but legal combination of primitives
- write down a sequence of declaration / evaluation usually to meet some end goal.
- Complex software is built by combining numerous of these tiny, simple instructions into a carefully organised sequence.

sauravsray@iitm.ac.in

What is a program?

Program

A program is a sequence of definitions and commands

- definitions **evaluated**
- commands **executed**

Scripts/Shells

- can be typed directly in a shell
- typically stored in a file

Install and use Python through IDLE (Ubuntu)

- `python3 --version`
- `sudo apt-get update`
- `sudo apt-get install idle-python3.12 python3-pip`
- `idle-python3.12` (start idle)

Where things may go wrong

Syntactic errors:

- common
- easily caught by compilers
- English:

  “sat cat mat” $\rightarrow$ not syntactically valid

  “cat sat on the mat” $\rightarrow$ syntactically valid
- Programming Language:

  “hi” 5 $\rightarrow$ not syntactically valid

  3.2*5 $\rightarrow$ syntactically valid

Where things may go wrong

Static semantic errors:

- syntactically valid strings
- no semantic errors
  1. program crashes, stops running
  2. program runs forever
  3. program gives an answer but different than expected

Content

sauravsray@iitm.ac.in

OBJECTS

- programs manipulate data objects
- objects have a type
  - Saurav is a human so he can breathe, sleep, etc.
  - An uncle is a relative, so he can give career advice, discuss politics, ask about your grades.
  - Mango is a fruit, so it can be ...
- Objects are:
  1. Scalar (cannot be subdivided)
  2. non-scalar (have internal structure that can be accessed)

Scalar objects

- `int` - represents integers, e.g. 7
- `float` - represents real numbers. e.g. 3.14159265
- `bool` - represents real Boolean values `True` or `False`
- `NoneType` - special and has one value, `None`
- use `type (..)` to see the type of an object

Code1

```
# Scalar Data type

x = 10                  # int
y = 3.14                # float
z = 2 + 5j              # complex
flag = True             # bool
type(x)
print(type(x))
```

Scalar objects

Type conversion (cast)

- can convert object of one type to another
- `float (3)` converts integer `3` to float `3.0`
- `int (3.9)` truncates float `3.9` to integer `3`

Print to console

- `3 + 2`
- `print (3 + 2)`

sauravsray@iitm.ac.in

Variables

Binding Variables and values

- `interest = 1000.0 * 9.8 * 6 / 100.0`
- `roi = 9.8`
- `prin = 1000.0`
- `time = 6`
- `interest = prin * roi * time / 100.0`
- can modify interest by modifying any of the basic components.

Code3

```
# definitions <- This is a comments
pi = 3.14159
radius = 2.2

area = pi * radius # what is this?
print('area = ', area)

# use comments to help others understand what y
radius = radius + 1
print(area)      # area doesn't change

# Recompute
area = pi*(radius**2)
print(area)      # are is updated
```

Math Functions and Number Methods

1. `round()`, for rounding numbers to some number of decimal places
   - ```
     >>> round(2.3)
     >>> round(3.14159, 3)
     >>> round(2.65, 1.4)
     ```
2. `abs()`, for getting the absolute value of a number
   - ```
     >>> abs(3)
     >>> abs(-5.0)
     ```
3. `pow()`, for raising a number to some power
   - ```
     >>> pow(2, 3)
     >>> pow(2, -2)
     >>> pow(2, 3, 2)
     ```
     ```

Strings

- Many programmers deal with text on a daily basis
- example
  - web developers work with text that gets input from web forms
  - Data scientists process text to extract data and perform things like sentiment analysis, to classify opinions in a body of text.
- Collections of text in Python are called strings

sauravsray@iitm.ac.in

Strings

- letters, special characters, spaces, digits
- enclose in

```
hi = 'hello there' or
hi = "hello there" or
```

- concatenate strings

```
name = "saurav"
greet = hi + name
greeting = hi + " " + name
```

- String Indexing

```
>>> flavor = "apple pie"
>>> flavor[1]
'p'
```

- In Python—and most other programming languages—counting always starts at zero.

Strings

- To get the character at the beginning of a string, you need to access the character at position 0:

```
>>> flavor[0]
'p'
```

- String Slicing
  - Suppose you need the string containing just the first three letters of the string "apple pie".

```
>>> first_three_letters = flavor[0] +
flavor[1] + flavor[2]
>>> first_three_letters
'app'
```

  - a portion of a string, called a substring

```
>>> flavor = "apple pie"
>>> flavor[0:3]
'app'
```

Strings

Strings Are Immutable

- we can't change them once we've created them

```
>>> word = "goal"
>>> word[0] = "f"
```

- If we want to alter a string, we must create an entirely new string

Converting Strings to Numbers

- `int()` stands for integer and converts objects into whole numbers,
- `float()` stands for floating-point number and converts objects into numbers with decimal points.

Strings

Converting Numbers to Strings

- Sometimes you need to convert a number to a string.
- ```
  >>> num_pancakes = 10
  >>> "I am going to eat " + num_pancakes
  + " pancakes."
  ```
- Since `num_pancakes` is a number, Python can't concatenate it with the string "I'm going to eat".
- To build the string, you need to convert `num_pancakes` to a string using `str()`:

  ```
  >>> num_pancakes = 10
  >>> "I am going to eat " +
  str(num_pancakes) + " pancakes."
  ```
  ```

Input / Output

- output

```
>>> x = 1
>>> print(3) >>> print(5*text)
```

Interact With User Input

- input

```
>>> text = input("Type anything...  ")
>>> print (text)
```

- Working With Strings and Number:
  user input using the `input()` function, the result is always a string.

Content

sauravsray@iitm.ac.in

Comments

```
import math

principal = 500000
interest = 7.5
duration = 20

r = interest / 12 / 100
n = duration * 12

if r == 0:
    payment = principal / n
else:
    payment = principal * r * (1 + r) ** n / ((1 + r) ** n - 1)

print("Monthly EMI = ", payment, " Ruppes")
```

Comments

A person can understand the syntax, but they may wonder:

- Why is the interest divided by 12 and then by 100?
- Where did this complicated formula come from?
- Why is there a special case when r == 0?

The code works, but the intent is not obvious.

sauravsray@iitm.ac.in

Comments

```python
import math

# Loan details
principal = 500000      # Loan amount in rupees
interest = 7.5          # Annual interest rate (in percent)
duration = 20           # Loan repayment period (in years)

# Convert the annual interest rate into a monthly decimal rate,
# as the EMI formula works with monthly interest.
r = interest / 12 / 100

# Total number of monthly installments.
n = duration * 12

# If there is no interest, each installment is simply an equal
# share of the principal.
if r == 0:
    payment = principal / n
else:
    # Compute the Equated Monthly Installment (EMI) using the
    # standard loan amortization formula used by banks.
    payment = principal * r * (1 + r) ** n / ((1 + r) ** n - 1)

# Display the monthly payment amount.
print("Monthly EMI =", payment, "Rupees")
```

Comments

What the comments add

- Why the interest rate is converted.
- What n represents.
- Why there is a special case for zero interest.
- Where the formula comes from (the standard EMI formula).
- What each input variable means.

sauravsray@iitm.ac.in

Comments

```
# Print "Hello, world"
print("Hello, world")
```

- No comment is needed in this example.
- best used to clarify code that may not be easy to understand
- to explain why something is done a certain way
- comments should be used sparingly
- Use comments only when they add value

Functions

- Functions are the building blocks of almost every Python program.
- we've already seen how to use several functions, including `print()`, `len()`, and `round()`.
- These are all built-in functions because they come built into the Python language itself.
- You can also create user-defined functions that perform specific tasks.
- Functions break code into smaller chunks, and are great for tasks that a program uses repeatedly

Functions Are Values

- One of the most important properties of a function in Python is that functions are values and can be assigned to a variable.
- Like any other variable, you can assign any value you want to len:

```
>>> len= "I'm not the len you're
looking for."
>>> len
```

- Now `len` has a string value
- The variable name `len` is a keyword in Python, and even though you can change it's value, it's usually a bad idea to do so.

Functions Are Values

- Changing the value of `len` can make your code confusing
- because it's easy to mistake the new `len` for the built-in function.
- If you typed in the previous code examples, you no longer have access to the built-in `len` function in IDLE.
- You can get it back with the following code:

```
>>> del len
```

- The del keyword is used to un-assign a variable from a value.
- `del` stands for delete, but it doesn't delete the value. Instead, it detaches the name from the value and deletes the name.

How Python Executes Functions

- Typing just the name doesn't execute the function.
- must call the function to tell Python to actually execute it.

```
>>> len()
```

- Python raises a TypeError when `len()` is called because `len()` expects an argument.
- An argument is a value that gets passed to the function as input.
- Some functions can be called with no arguments, and some can take as many arguments as you like.
- When a function is done executing, it returns a value as output.

sauravsray@iitm.ac.in

How Python Executes Functions

The process for executing a function can be summarized in three steps:

1. The function is called, and any arguments are passed to the function as input.
2. The function executes, and some action is performed with the arguments.
3. The function returns, and the original function call is replaced with the return value.

```
>>> num_letters = len("four")
```

- When a function changes or affects something external to the function itself, it is said to have a side effect.

  ```
  >>> return_value= print("What do I return?")
  ```

User define Functions

- Repetitive code can be a nightmare to maintain.
- If you find a mistake in some code that's been copied and pasted all over the place, you'll end up having to apply the fix everywhere the code was copied.

The Anatomy of a Function

Every function has two parts:

1. The **function signature** defines the name of the function and any inputs it expects.
2. The **function body** contains the code that runs every time the function is used.

Multiplication function

- write a function that takes two numbers as input and returns their product.

```
def multiply(x, y): # Function signature
    # Function body
    product = x * y
    return product
```

- The first line of code in a function is called the function signature.
- It always starts with the `def` keyword, which is short for “define.”

Function Signature

```
def multiply(x, y):
```

The function signature has four parts:

- The `def` keyword
- The function name, `multiply`
- The parameter list, `(x, y)`
- A colon `(:)` at the end of the line

- When Python reads a line beginning with the def keyword, it creates a new function.
- The function is assigned to a variable with the same name as the function name.

Functions

- a function name can only contain numbers, letters, and underscores, and must not begin with a number.
- A function can have any number of parameters, including no parameters at all!

The Function Body

- The function body is the code that gets run whenever the function is used in your program.

```
# Function body
product= x * y
return product
```

- The second line of code is called a return statement.
- It starts with the `return` keyword and is followed by the variable product.

Indentation in Python

What is indentation?

- Indentation means adding spaces (or a tab) at the beginning of a line.
- In most programming languages, indentation is used mainly to improve readability.
- In Python, indentation is part of the language syntax.
- It tells Python which statements belong together as a block.

```
def multiply(x, y):
    product= x * y
    return product

print("Where am I?") # Not in the function body.
```

- In Python, indentation is mandatory, not just a matter of style.

Indentation in Python

Incorrect Indentation

- If `print()` is indented, then it becomes a part of the function body even if there is a blank line between `print()` and the previous line

```
marks = 45

if marks >= 50:
    print("Marks = ", marks)
print("Pass")
```

- There is one rule every line must be indented by the same number of spaces.

Function Return

- Once Python executes a return statement, the function stops running and returns the value.

```
def multiply(x, y):
    product= x * y
    return product

    print("Where am I?") # Not in the function body.
```

- If any code appears below the return statement that is indented so as to be part of the function body, it will never run

Calling a User-defined Function

- You call a user-defined function just like any other function.
- Type the function name followed by a list of arguments in between parentheses.
- to call `multiply()` with the argument 2 and 4, just type:

```
multiply(2, 4)
```

-
```
num = multiply(2, 4)
print(num)

def multiply(x, y):
    product= x * y
    return product
```

EMI Calculator

Write a function to generalise the EMI Calculator and compute the interest for one particular case.

```
import math

# Loan details
principal = 500000      # Loan amount in rupees
interest = 7.5          # Annual interest rate (in percent)
duration = 20           # Loan repayment period (in years)

# Convert the annual interest rate into a monthly decimal rate,
# as the EMI formula works with monthly interest.
r = interest / 12 / 100

# Total number of monthly installments.
n = duration * 12

# If there is no interest, each installment is simply an equal
# share of the principal.
if r == 0:
    payment = principal / n
else:
    # Compute the Equated Monthly Installment (EMI) using the
    # standard loan amortization formula used by banks.
    payment = principal * r * (1 + r) ** n / ((1 + r) ** n - 1)

# Display the monthly payment amount.
print("Monthly EMI =", payment, "Rupees")
```

Functions With No Return Statement

- All functions in Python return a value, even if that value is None.
- However, not all functions need a return statement.

```
def greet(name):
    print(f"Hello, name!")
```

- Even though `greet()` has no return statement, it still returns a value:

```
return_value= greet("Dave")
```

- Notice also that the string `"Hello, Dave!"` is printed even when the result of `greet("Dave")` is assigned to a variable.
- That's because the call to `print()` inside of the `greet()` function body produces the side effect of always printing to the console.

Documenting Your Functions

- ```
  >>> help(len)
  Help on built-in function len in module
  builtins:
  len(obj, /)
  Return the number of items in a
  container.
  ```
- When you pass a variable name or function name to `help()`, it displays some useful information about it.
- Let's see what happens when you call `help()` on the `multiply()` function:
- `help()` displays the function signature, but there isn't any information about what the function does.
  ```

Documenting Your Functions

- To better document `multiply()`, we need to provide a `docstring`.
- A `docstring` is a triple-quoted string literal placed at the top of the function body.
- Docstrings are used to document what a function does and what kinds of parameters it expects.
- Here's what multiply() looks like with a docstring added to it:

```
def multiply(x, y):
    """Return the product of 2 numbers x and y."
    product= x * y
    return product
```

Content

sauravsray@iitm.ac.in

Run in Circles

- A loop is a block of code that gets repeated over and over again either a specified number of times or until some condition is met.
- There are two kinds of loops in Python: `while` loops and `for` loops.

The `while` Loop

`while` loops repeat a section of code while some condition is true. There are two parts to every `while` loop:

1. The `while` statement starts with the `while` keyword, followed by a test condition, and ends with a colon (:).
2. The loop body contains the code that gets repeated at each step of the loop. Each line is indented.

`while` loop

- When a while loop is executed, Python evaluates the test condition and determines if it is true or false.
- If the test condition is true, then the code in the loop body is executed.
- Otherwise, the code in the body is skipped and the rest of the program is executed.
- If the test condition is true and the body of the loop is executed, then once Python reaches the end of the body, it returns to the while statement and re-evaluates the test condition.
- If the test condition is still true, the body is executed again.
- If it is false, the body is skipped.

sauravsray@iitm.ac.in

`while` loop

- This process repeats over and over until the test condition fails, causing Python to loop over the code in the body of the while loop.
- ```
  >>> n = 1
  >>> while n < 5:
          print(n)
          n = n + 1
  ```
- drop the last statement `n = n + 1`
- If you aren't careful, you can create an infinite loop.
- This happens when the test condition is always true.
- An infinite loop never terminates.
  ```

`while` Infinite loops

- Infinite loops aren't inherently bad.
- Sometimes they are exactly the kind of loop you need.
- For example, code that interacts with hardware may use an infinite loop to constantly check whether or not a button or switch has been activated.
- If you run a program that enters an infinite loop, you can force Python to quit by pressing Ctrl+C.
- Python stops running the program and raises a Keyboard Interrupt error:

```
Traceback (most recent call last):
    File "<pyshell#8>", line 2, in <module>
        print(n)
KeyboardInterrupt
```

Example: Estimating e Using a `while` Loop

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

Problem Approximate the value of e by successively adding terms of the series.Continue adding terms until the next term satisfies $\frac{1}{n!} < 10^{-8}$.

- This is a common stopping criterion in numerical mathematics.

Why is a `while` loop appropriate?

- The number of terms required is *not known in advance*.
- The computation stops only when a desired accuracy is achieved.

Python Implementation

```
n = 0
term = 1.0
total = 0.0

while term > 1e-8:
    total += term
    n += 1
    term = term / n

print("Approximation of e =", total)
print("Terms used =", n)
```

Finding a Root Using a `while` Loop

$$f(x) := x^3 - x - 2 = 0.$$

$$f(1) = -2, \qquad f(2) = 4,$$

so there is a root in the interval $[1, 2]$.

Problem

Use the *bisection method* to approximate the root. Continue halving the interval until its length is less than 10^{-6}.

Why use a `while` loop?

- The number of iterations is not known beforehand.
- The algorithm stops only when the desired accuracy is achieved.

Python Implementation

```
def f(x):
    return x**3 - x - 2

a = 1.0
b = 2.0
tol = 1e-6

while (b - a) > tol:
    c = (a + b) / 2

    if f(a) * f(c) < 0:
        b = c
    else:
        a = c

root = (a + b) / 2
print("Approximate root =", root)
```

The `for` Loop

- A `for` loop executes a section of code once for each item in a collection of items.
- The number of times that the code is executed is determined by the number of items in the collection.

Like `while` the `for loop` has two main parts:

1. The `for` statement begins with the `for` keyword, followed by a membership expression, and ends in a colon `(:)`.
2. The loop body contains the code to be executed at each step of the loop, and is indented.

The `for` Loop

Type:

```
for letter in "Python":
.       print(letter)
```

How to do the same with `while` loop?

```
word= "Python"
index= 0
while index < len(word):
    print(word[index])
    index= index + 1
```

Numerical Integration Using a `for` Loop

$$I = \int_0^1 x^2 \, dx.$$

- approximate the integral using the **Left Rectangle Rule**.
- Divide the interval $[0, 1]$ into N equal subintervals of width $\Delta x = \frac{1}{N}$

$$I \approx \sum_{i=0}^{N-1} f(x_i) \, \Delta x, \quad x_i = i \, \Delta x, \qquad f(x) = x^2.$$

Task: Approximate the integral using $N = 100$ subintervals.

Why use a `for` loop?

- The number of rectangles (N) is known in advance.
- The same computation is repeated exactly N times.

Python Implementation

```
def f(x):
    return x**2

N = 100
dx = 1.0 / N

integral = 0.0

for i in range(N):
    x = i * dx
    integral += f(x) * dx

print("Approximate integral =", integral)
print("Exact value =", 1.0/3.0)
```

Nested `for` Loops

A **nested loop** is a loop inside another loop.

General Syntax

```
for i in range(...):
    # Statements in the outer loop
    for j in range(...):
        # Statements in the inner loop
    # More statements in the outer loop
```

How it works

- The outer loop executes once.
- For each iteration of the outer loop, the inner loop runs from start to finish.
- If the outer loop runs m times and the inner loop runs n times, then the inner loop executes a total of $m \times n$ times.

Numerical Integration over a Rectangle

$$f(x, y) = x + y, \qquad 0 \leq x \leq 1, \qquad 0 \leq y \leq 1.$$

We wish to approximate the double integral

$$I = \int_0^1 \int_0^1 f(x, y)\, dy\, dx.$$

Divide the square into an $N \times N$ grid of equal cells, where

$$\Delta x = \Delta y = \frac{1}{N}.$$

Approximate the integral using

$$I \approx \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} f(x_i, y_j)\, \Delta x\, \Delta y, \quad x_i = i\Delta x, \qquad y_j = j\Delta y.$$

Task

- approximate the integral using $N = 100$.
- Use **nested `for` loops** to compute the double summation.
- Compare your answer with the exact value of the integral.

Scope Hierarchies

- Scopes have a hierarchy.

```
x = "Hello World"
def func():
    x = 2
    print("Inside 'func', x has the value ", x)

func()
print("Outside 'func', x has the value", x)
```

- How does x still have the value ”Hello World” after calling `func()`, which changes the value of x to 2?
- the function `func()` has a different scope than the code that exists outside of the function.

Scope Hierarchies

- can name an object inside `func()` the same name as something outside `func()`
- Python can keep the two separated.
- The function body has what is known as a local scope, with its own set of names available to it.
- Code outside of the function body is in the global scope.

```
x = 5
def outer_func():
    y = 3
    def inner_func():
        z = x + y
        return z
    return inner_func()
```

sauravsray@iitm.ac.in

The LEGB Rule

- This rule is an acronym for Local, Enclosing, Global, Built-in.
- Python resolves scope in the order in which each scope appears in the list LEGB.
- Local (L): The local, or current, scope; could be the body of a function or the top-level scope of a script.
- It always represents the scope that the Python interpreter is currently working in.
- Enclosing (E): The enclosing scope. This is the scope one level up from the local scope.
- If the local scope is an inner function, the enclosing scope is the scope of the outer function.

sauravsray@iitm.ac.in

The LEGB Rule

- Global (G): The global scope, which is the top-most scope in the script.
- This contains all of the names defined in the script that are not contained in a function body.
- Built-in (B): The built-in scope contains all of the names, such as keywords, that are built-in to Python.
- Functions such as `round()` and `abs()` are in the built-in scope.
- Anything that you can use without first defining yourself is contained in the built-in scope.

Break the Rules

```
total= 0
def add_to_total(n):
    total= total + n
add_to_total(5)
print(total)
```

- Something unexpected occurs. You get an error!
- LEGB rule $\Longrightarrow$, Python should have recognized that the name total doesn't exist in the `add_to_total()` function's local scope and moved up to the global scope to resolve the name, right?
- The problem here is that the script attempts to make an assignment to the variable total, which creates a new name in the local scope.

Break the Rules

- when Python executes the right-hand side of the assignment it finds the name `total` in the local scope with nothing assigned to it yet.
- These kinds of errors are tricky and are one of the reasons it is best to use unique variable and function names no matter what scope you are in.
- can get around this issue with the global keyword

```
def add_to_total(n):
    global total
```

- Although this “fixes” the script, the use of the `global` keyword is considered bad form in general.

sauravsray@iitm.ac.in

Content

sauravsray@iitm.ac.in

Conditional Logic

- Programs often need to make decisions based on certain conditions.
- The basic syntax of an `if` statement is

```
if condition:
    statement(s)
```

Example:

```
x = -3
if x < 0:
    print("x is negative")
```

The condition is a logical expression that evaluates to either `True` or `False`.

The `if-else` Statement

- Use an `else` block when there are two possible outcomes.
- Example: Determine whether an integer is even or odd.

```
n = 17
if n % 2 == 0:
    print("Even")
else:
    print("Odd")
```

Flow of execution

$$\boxed{\text{Condition}} \longrightarrow \begin{cases} \text{True} & \rightarrow \text{Execute if-block} \\ \text{False} & \rightarrow \text{Execute else-block} \end{cases}$$

Multiple Conditions

- Use `elif` when there are more than two possible cases.
  Example: Determine the sign of a number.

```
x = -2.5
if x > 0:
    print("Positive")
elif x < 0:
    print("Negative")
else:
    print("Zero")
```

Flow of execution

$$\boxed{x > 0?} \longrightarrow \begin{cases} \text{True} & \rightarrow \text{Positive} \\ \text{False} & \boxed{x < 0?} \longrightarrow \begin{cases} \text{True} & \rightarrow \text{Negative} \\ \text{False} & \rightarrow \text{Zero} \end{cases} \end{cases}$$

Classifying the Roots of a Quadratic

For the quadratic equation

$$ax^2 + bx + c = 0,$$

the discriminant is

$$D = b^2 - 4ac.$$

The nature of the roots depends on the value of D.

```
D = b**2 - 4*a*c
if D > 0:
    print("Two distinct real roots")
elif D == 0:
    print("One repeated real root")
else:
    print("Two complex roots")
```

In-Class Exercise: Classifying the Roots of a Quadratic

Consider the quadratic equation **Write a Python program that:**

1. Prompts the user to enter the coefficients a, b, and c.
2. Computes the discriminant D.
3. Uses `if-elif-else` statements to determine whether the equation has
   - Two distinct real roots,
   - One repeated real root, or
   - Two complex roots.
4. Prints the value of the discriminant and the corresponding classification.

Test your program with: $(1, -3, 2)$, $(1, -2, 1)$, $(1, 2, 5)$.

sauravsray@iitm.ac.in

Boolean Operations on Strings

Strings can be compared just like numbers.

Examples

```
name = "Alice"
print(name == "Alice")        # True
print(name != "Bob")          # True

print("cat" == "Cat")         # False
print("apple" < "banana")     # True

print("py" in "python")       # True
print("Java" in "python")     # False

print("cat" not in "dog")     # True
```


sauravsray@iitm.ac.in

Comparison Operators

Common Boolean Operators

`==`	Equal to
`!=`	Not equal to
`<, >, <=, >=`	Lexicographical comparison
`in`	Check if a substring exists
`not in`	Check if a substring does not exist

Lexicographical Comparison of Strings

Python compares strings character by character using the alphabetical (lexicographical) order.

Examples

```
print("apple" < "banana")      # True
print("cat" < "dog")           # True
print("apple" < "Apple")       # False
print("10" < "2")              # True
```

Notes

- Uppercase and lowercase letters are different.
- Comparison starts from the first character.
- Digits are compared as characters, not as numbers.

sauravsray@iitm.ac.in

The `in` and `not in` Operators

- The `in` operator checks whether a substring is present in a string.
- The `not in` operator checks whether a substring is absent.

Examples

```
text = "Industrial Mathematics"
print("Math" in text)
print("math" in text)
print("Data" in text)
print("Data" not in text)
print("Industrial" in text)
print("ics" in text)
```

- String matching is **case-sensitive**.
- The substring may appear anywhere within the string.

sauravsray@iitm.ac.in

Combining Boolean Expressions

- Python provides three logical operators for combining Boolean expressions.

Operator	Meaning
`and`	True if *both* conditions are True
`or`	True if *at least one* condition is True
`not`	Reverses the truth value

Examples

```
x = 5
print(x > 0 and x < 10)      # True
print(x < 0 or x > 10)       # False
print(not(x == 5))           # False
```

- These operators allow us to write more complex conditions.

Examples Using Logical Operators

```
temperature = 25
humidity = 60
```

Examples

```
# Is the weather warm and dry?
temperature > 20 and humidity < 70
# Is it freezing or very hot?
temperature < 0 or temperature > 35
# Is the temperature NOT below zero?
not (temperature < 0)
# Is x outside the interval [0,10]?
x < 0 or x > 10
# Is x inside the interval [0,10]?
x >= 0 and x <= 10
```

- Use parentheses to make complex logical expressions easier to read.

Why Do We Need the `not` Operator?

Suppose a student is eligible for a scholarship if

- GPA ≥ 8.0, and
- Attendance $\geq 75\%$.

Without `not`

```
if gpa < 8.0 or attendance < 75:
    print("Not eligible")
```

Using `not`

```
if not (gpa >= 8.0 and attendance >= 75):
    print("Not eligible")
```

- The `not` operator lets us negate an entire condition without having to rewrite the logic.
- For compound conditions, this makes the code easier to read and less error-prone.