**Introduction To Python**

Python is a high-level, interpreted, object-oriented programming language known for its simplicity and readability, making it a popular choice for beginners and experienced developers alike; it is often used for web development, data analysis, machine learning, and automation tasks, with a key feature being its reliance on **whitespace indentation** to define code blocks, rather than curly brackets like many other languages; developed by Guido van Rossum, the name "Python" is a reference to the British comedy group Monty Python , and to any Snake type.

**Some convincing key points about Python are:**

* Python is a language that can be used for a wide range of applications, from simple scripting to complex data analysis and machine learning i.e from cooking Maggie to cooking something difficult.
* Code execution is no complex just simple line after line without any extr-ordinary compilation steps which is a benefit for testing quick and development.
* Also it supports Object – Oriented concepts such as classes and objects which are quiet easy.
* Python is a chill guy and hence do not need to explicitly declare variables with data-types, which makes it more concise.
* Readability of python with clean syntax along with clear keywords and significant whitespaces for code indentation, makes is easy to read and fun to understand.
* Python has a motherly nature and hence has a vast collection of pre-built modules and libraries for several tasks.

**Declare:**

To declare something means to specify or introduce an entity, such as a variable, without necessarily assigning it a value.Variables are declared simply by assigning a value to them. There is no need to specify the type explicitly because Python is dynamically typed.

**Define:**

To define something involves creating it with specific functionality, such as defining a function, class, or variable.

Variables are defined when you assign a value to them.

Functions are defined using the def keyword.

**Call:**

Calling refers to executing a function or method that has been defined earlier.

You "call" a function by using its name followed by parentheses (). If the function takes arguments, you include them inside the parentheses.

**Indentation:**

Indentation is the space (or tab) at the beginning of a line of code that determines the structure and grouping of code.

Indentation is mandatory and replaces braces {} found in Python and in other languages like C++ or Java.

It defines blocks of code, such as loops, functions, or conditionals

**Condition:**

A condition is a statement that evaluates to True or False and determines whether specific blocks of code will execute..

**Basic Syntax**

**Variables**

A "variable" in Python is a named container that stores a data value, essentially acting as a placeholder for information that can be accessed and modified throughout a program; in simpler terms,Variables are the cabinates which we see inthe kitchen, which stores the containers in it.Python creates a variable the moment you assign a value to it, without needing a separate declaration step.

**Significant points about Python variables:-**

* To create a variable , we simply assign a value to name using “The Equal sign (=) “ , example:

age = 13

* Every variable holds a specific data-type like an integer or string. Float , list, etc
* Python being a unique model , we don’t need to declare a variable before using it , as soon as we assign a value it gets created.

**For example :-**

Name = “alice” #stores the string ‘alice’ in variable named ‘name’

age = 25 #stores the integer 25 in variable named ‘age’

is\_student = True #stores the Boolean value ‘True’ in variable

**Important points to remember while creating a variable:-**

* Names created are case sensitive, which means ‘age’ and ‘Age’ are considered as two different variables.
* Characters used to create a variable includes alphabets(a-z , A-Z) , numbers(0-9) and only underscore ( \_ ) as a special character.
* Practicing to create good and understandable variable names is a sign of a good programmer and hence creating names like “user\_name” or “total\_score” helps your code more readable.
* It is a good practice to avoid reserved keywords by python such as “if” , “for” , “while” as variable names.

**Print Function**

In Python, “print” refers to an in-built function print(), Which is used to display data or output(standard) on the screen , in which the provided information is automatically converted to a string before being printed.

**Points to remember about Print() Function:**

* print("My name is Chen") , this will print “My name is chen” to the console.
* Python allows to print multiple values in a single line by separating them with comma’s( , ) within print() function, which will automatically add spaces in between.
* Data type passed to print() can be converted to a string before printing.

**For example:-**

Name = “Hoodle\_bee”

Print(“Hellooww , my name is ”,name)

Output : Hellooww , my name is Hoodle\_bee

**Input Function**

In Python, "input" refers to the process of getting data from a user by using the **built-in "input()"function**, which allows the program to pause and wait for the user to type something on the keyboard, storing that input as a string within a variable; essentially, it's how a Python program can actively receive information from the user during execution.

**Key points to be remembered:-**

- Function: The primary way to get user input in Python is through the input() function.

- String return type: Regardless of what the user enters (number, text, etc.), the input() function always returns the input as a string.

- Prompting: You can include a message within the input() function to guide the user on what to enter, called a "prompt".

For Example:

name = input("What is your name? ")

print("Hello,", name)

**Important Points to Remember:**

- Type conversion: If you want to use the input as a number (integer or float), you need to explicitly convert it using the int() or float() functions respectively. Basically type conversion is converting a specific data-type to other specific data-type.

- Error handling: Since input() always returns a string, you may need to validate user input to ensure it is in the correct format (e.g., checking if a number is entered when expected).

**Operators in Python**

**Introduction to operators:**

Python operators are special symbols or keywords used to perform operations on variables and values. They allow you to carry out mathematical calculations, comparisons, logical checks, and other manipulations on data within your code.

**Types of Operators**

1. **Arithmetic Operators**

These are used to perform basic mathematical operations such as

**= >Addition (+)**

a = 4

b = 3

print(“sum = “ , a + b)

**Output:** sum = 7

**= > Subtraction ( - )**

a = 4

b = 3

print(“sub = “ , a - b)

**Output:** sub = 1

**= > Multiplication ( \* )**

a = 4

b = 3

print(“multiply = “ , a \* b)

**Output:** multiply = 12

**= > Division ( / )**

a = 4

b = 2

print(“division = “ , a / b)

**Output:** division = 2

**= > Floor division ( // )**

a = 13

b = 4

print(“Floor division = “ , a // b)

**Output:** Floor division = 3

**= > Modulo ( % )**

a = 13

b = 4

print(“Modulo = “ , a % b)

**Output:** Modulo = 1

**= > Exponent ( \*\* )**

a = 4

b = 5

print(“Exponent = “ , a \*\* b)

**Output:** multiply = 1024

1. **Assignment Operators**

Assignment operators are used to assign values to variables.

**= > Basic assignment ( = )**

r = 8

print(r)

**Output:** 8

**= >Addition assignment ( += )**

f = 8

f += 2 # is f = f + 2

print(f)

**Output:**  10

**= > Subtraction assignment ( -= )**

f = 8

f -= 2 # is f = f - 2

print(f)

**Output:**  6

**= > Multiplication assignment ( \*= )**

f = 8

f \*= 2 # is f = f \* 2

print(f)

**Output:**  16

**= > Division assignment ( /= )**

f = 8

f /= 2 # is f = f / 2

print(f)

**Output:**  4

**= > Floor division assignment ( //= )**

f = 8

f //= 2 # is f = f // 2

print(f)

**Output:**  4 # rounds the division to the nearest number

**= > Modulus assignment ( %= )**

f = 8

f %= 2 # is f = f % 2

print(f)

**Output:**  0

**= > Exponentiation assignment ( \*\*= )**

f = 8

f \*\*= 2 # is f = f \*\* 2

print(f)

**Output:**  64

**= > Bitwise AND assignment ( &= )**

f = 8

f &= 2 # is f = f & 2

print(f)

**Output:**  0

**= > Bitwise OR assignment ( |= )**

f = 8

f |= 2 # is f = f | 2

print(f)

**Output:**  10

**= > Bitwise XOR assignment ( ^= )**

f = 8

f ^= 2 # is f = f ^ 2

print(f)

**Output:**  10

**= > Right shift assignment ( >>= )**

f = 8

f >>= 2 # is f = f >> 2

print(f)

**Output:**  2

**= > Left shift assignment ( <<= )**

f = 8

f <<= 2 # is f = f << 2

print(f)

**Output:**  32

**= > Walrus operator ( := )**

print(f := 8)

**Output:**  8

1. **Bitwise Operators**

Bitwise operators are used to manipulate individual bits in a binary pattern

**= > Bitwise AND ( & )**

print(8 & 2)

#This performs AND operation i.e compares each bit and set to 1 if both are 1 , else 0.

**Output:**  0

**= >Bitwise OR ( | )**

print(8 & 2)

#This performs OR operation i.e compares each bit and set to 1 if both or even one bit is 1 , else 0.

**Output:**  10

**= > Bitwise Not ( ~ )**

Print(~ 8 )

#This converts all 1’s to 0’s and all 0’s to 1’s

**Output:**  - 9

**= > Bitwise XOR ( ^ )**

Print( 2 ^ 8 )

#compares each bit and set it to 1 if only one is 1, otherwise 0

**Output:** 10

**= > Right shift ( >> )**

Print( 8 >> 2 )

#moves each bit the specified number of times to the right. Empty holes at the left are filled with 0's

**Output:** 2

**= > Left shift ( << )**

Print ( 10 << 2 )

#inserts the specified number of 0's (in this case 2) from the right and let the same amount of leftmost bits fall off

**Output:** 40

1. **Comparison Operators**

Comparison operators are used to compare two values and return a Boolean result.

**= > Equal to ( == )**

r = 4

s = 10

print( r == s )

# This prints Boolean value False as 4 and 10 are not equal.

**Output:** False

**= > Not equal to ( != )**

r = 4

s = 7

print( r != s)

#This returns True as 4 is not equal to 7

**Output:** True

**= > Greater than ( > )**

r = 4

s = 7

print( r > s)

#This returns False as 4 is less than 7

**Output:** False

**= > Less than ( < )**

r = 4

s = 7

print( r < s)

#This returns True as 4 less than 7

**Output:** True

**= > Greater than Equal to ( >= )**

r = 4

s = 4

print( r >= s)

**Output:** True

**= > Less than Equal to ( <= )**

r = 4

s = 7

print( r <= s)

**Output:** False

1. **Logical Operators**

Logical Operators are used to combine multiple conditions within a single Expression.

**= > Logical AND**

Returns true if both Statements are true

**Example:** x > 1 and x < 10

**= > Logical OR**

Returns true if one of the Statement among two is true

**Example:** x > 1 or x < 10

**= > Logical NOT**

Reverse the result , i.e if the output is true it returns False

**Example:** not(x > 1 && x < 10)

1. **Ternary Operator:**

A ternary operator in Python is a one-liner shorthand for an if-else statement. It’s used to evaluate a condition and return one of two values based on whether the condition is True or False.

**Example:**

age = 20

status = "Child" if age < 18 else "Adult" if age < 65 else "Senior"

print(status)

**Output**: Adult

1. **Membership Operator**

Membership operators in Python are used to check whether a value is a member of a sequence, such as a string, list, tuple, set, or dictionary.

**= > in Operator**

* Checks if a value is present in a sequence.
* Returns True if the value is found, otherwise False.

**Example:**

# Using 'in' with a list

horror\_movies = ["hereditary", "smile", "longlegs"]

print("longlegs" in fruits) # Output: True

print("trap" in fruits) # Output: False

# Using 'in' with a string

text = "Hello, Scoobiesss!"

print("scoobiesss!" in text) # Output: True

print("geez" in text) # Output: False

**= > not in Operator**

* Checks if a value is **not present** in a sequence.
* Returns True if the value is not found, otherwise False.

**Example:**

# Using 'not in' with a list

movies = ["Sabrina", "Casablanca", "Grease"]

print("holiday" not in fruits) # Output: True

print("Sabrina" not in fruits) # Output: False

# Using 'not in' with a string

text = "Hello, ChottaBheem!"

print("ChottaBheem" not in text) # Output: True

print("nobita" not in text) # Output: False

**Identity Operator**

Identity operators are used to compare the **memory locations** of two objects. They check whether two objects are the same object, not just if they are equal in value.

**= > is Operator**

The **is** operator evaluates to True if two objects refer to the **samememory location**(i.e., they are the same object).

x = [1, 2, 3]

y = x # y points to the same object as x

z = [1, 2, 3] # z is a new object with the same value as x

print(x is y) # Output: True (x and y are the same object)

print(x is z) # Output: False (x and z are different objects)

**= > is not Operator**

The **is not** operator evaluates to True if two objects do **not** refer to the same memory location.

x = [1, 2, 3]

z = [1, 2, 3] # z is a new object

print(x is not z) # Output: True (x and z are different objects)

**Comments**

In Python, a comment is a piece of text added within your code that explains the logic or purpose of a specific line or section of code. Comments are completely ignored by the Python interpreter when running the program. And hence it does not get printed on the Output Screen.

Python ignores them just as you ignore your preparations for exam whole year.

Syntax: A single line comment starts with a "#" symbol followed by the comment text.

Example: #This is MEEE

Multi-line Comments: While not a dedicated syntax, you can use triple quotes (""" or ''') to create multi-line comments by not assigning them to a variable.

Example: “”” This is MEee

NOT YOUUUU

“””

**Data Types in Python**

A data type is a classification of data that specifies the kind of value a variable can hold.

1. **Numeric Types** 
   1. **Integers (int) :**

int stores numbers without decimal i.e positive or negative numbers, for example 2 , 3 , -4 , 5 , -6 etc,.

example: - int\_num = 3

* 1. **Floating – point ( float )**

Floats stores positive and negative decimal numbers for example 3.14 , 2.5 , 4.13 etc.

example: - float\_num = 4.16

* 1. **Complex numbers ( complex )**

Complexe numbers are (real number) + (imaginary number), just like you living in between a real and delusional world, where your present is the real part and imaginary thoughts are your delusional part. It can be written as "4 + 13i" where i stands for imaginary , or just can be written as 3i or 4j etc

example : - complex\_number = 4j

1. **Boolean (bool)**

This is used to know if an operation’s result is true or false. Boolean values are either TRUE or FALSE.

in an operation like ' a > b ' ' a < b ' ' a = b ' or other such expressions , these values are used.

r = 4

s = 10

print( r == s )

# This prints Boolean value False as 4 and 10 are not equal.

**Output:** False

1. **Sequence Types** 
   1. **List**

Ordered Collection of items within Square brackets. allowing modification.

list in simple words can be defined as just as grocerey list? Where you collect many type of things together, list is ordered i.e one after another. Items in list need not to be of same data type, it can be integer or float or string , anything. List is more like an array of things.

It can be written inside a Square "[]" bracket .

A list can be **" INDEXED "** just as an array , starting by index '0' , it can also be sliced .

**For example :-** list = [3 , 4.14 , "harper"]

here '3' stands on 0th index while '4.14' is on 1st index and 'harper' is on 2nd index position.

There are different types of listing methods which includes:-

* sort() : This sorts the list i.e arranging the data accordingly or we can say ascending to descending order.

example : level = [2 ,3 , 1 ,6 , 8 ]

level.sort()

* append() : This listing method is used to add last min items which people forget . But it appends items at the last position , not in between nor first , because discipline should be maintained. It can append only one item at a time.

example : lists = [2 , 5 , "hellow" ]

list.append("people")

* insert() : It helps inserting /adding an item in a list to an exact position with the help of an index. syntax for same will can be list\_name.insert(index\_num , adding\_element)

example : marks = [50 , 23 , 45 , 65, 33]

marks.insert(2 , 35)

* remove() : This method helps removing an item from the list, its just like removing an over-ripped mango from your mango box.

example : length = [10 , 22 , 3 , 41 , 2]

length.remove(41)

* extend() : It also helps adding items but multiple items at a time which is therefore called extend. Just as expanding slime , but adds at the last on a list.

example : bread = [12 , 5]

bread.extend([11,6,13])

* 1. Tuple

Ordered collection within parentheses, but immutable.

Tuple can be described as same as list i.e to store multiple items in a single variable but instead of "[]" , we use "()" round brackets.

similar to list, it can accept any data types i.e mixed as well, tuple is also ordered, indexing is seen but cannot be changed i.e immutable. Tuple allows duplicate values to be written .

**For example :-** tuples = (4 , "peace" , 4.2 , "science” )

* 1. **String**

Sequence of characters enclosed in single or double quotes.

string should be defined as a collection of characters which in simple terms we called them as words, it can be anyones name, object name or anything , example "Tim" , "Lion" , etc

For example :- str = "Hellowww"

1. **Dictionary**

Key-value pairs enclosed in curly braces, allowing quick access to data using keys.

Dictionary is often described as collection of key : value pairs . value in dictionary can be of any datatype and can be changed while the keys cannot be changed and is always preferred to be something diferrent . Even dictionaries are portrayed in "{}" curly brackets.

Example : dictionary = { 'name' : "grey" , "age" : 20 , "weight" : 52 }

Dictionary can be made related to a pair of socks which cannot be separated i.e A pair is neccesarryyy.

-To change a specific item in dictionary , we use key-name to refer , similarly we can even add a pair of item for the same, example :

student = {"name" : "Ren" , "Year" : 2 , "Rollnum" : 12}

student["year"] = 3

1. **Set**

Unordered collection of unique items within curly braces.

Set is an unordered collection of data types , just as list and tuples but is iterable , doesn't allows duplicate items and is unindexed . Sets are protrayed in "{}" curly brackets.

Each element in a set must be unique, so if you try to add the same value twice, it will only be stored once.

For example :- sets = {4 , 6 , 8 , 10}

sets have few methods similar to listing method :

* remove() : know for removing the item from the set simliar to that in list.

example : set = {4, 5 , 6 , 7 , 8 , 9}

set.remove(6)

* add() : adds a single element to the set.

example : sets = {2 , 3 , 4 , 5 , 6}

sets.add(8)

**Type Casting**

In Python, type casting refers to the process of explicitly converting a value from one data type to another.

**Key Points About Type Casting**

- **Explicit Conversion:** Type casting in Python is always explicit, meaning you need to use the relevant function to convert a value to the desired data type.

- Common Functions for Type Casting:

- int(x): Converts a value to an integer.

- float(x): Converts a value to a floating-point number.

- str(x): Converts a value to a string.

This can be referred to an example of currency conversion from any currency type to other, rupee to dollar .

- Data Loss Potential: When converting between data types, information might be lost depending on the conversion.

Example Usage

age\_string = "30" # String representing an age

age\_int = int(age\_string) # Convert string to integer

print(type(age\_string)) # Output: <class 'str'>

print(type(age\_int)) # Output: <class 'int'>

print(age\_int + 5) # This will now work as expected, adding 5 to the integer age

**Important Considerations**

- Implicit Type Conversion: While Python primarily uses explicit type casting, in certain situations, it might automatically perform type conversions during operations involving mixed data types.

- Error Handling: Always check if a value can be converted to the desired data type before performing the conversion to avoid potential errors.

**Conditional Statements**

In Python, conditionals refer to programming statements that allow a program to execute different blocks of code depending on whether a specific condition is true or false.

For example imagine you want to cook " Enchiladas " but you have certain conditions only after which you can start cooking.

-These conditions can be ingredients, gas stove or oven and even can be your cooking skills.

-To be more clear if you have all the ingredient along with gas stove or oven whichever you prefer also you got some cooking skills you can easily Cook and eat BUT if even one of these is missing it becomes difficult for you to Cook.

-Similarly , conditional statements in python works .

**Key Points About Conditionals**

- To control the flow of a program by selectively executing code based on whether a condition is met.

-Conditional statements use the "if" keyword followed by the condition to be checked, a colon, and then the code block to execute if the condition is true.

- Components:

- if statement: Executes a block of code only if the condition is true.

- else statement: Executes a block of code if the "if" condition is false.

- elif statement: Allows checking multiple conditions sequentially, executing the code block only if the current "elif" condition is true.

for example : lets assume we have to check for Ages below 10 , so when user enters a certain age when asked it enters into :

if age<10:

print("valid age")

* imagine we created a variable named age and took input for age as "5" , now the "if" condition checks whether age is less than 10 i.e "5<10" then it completes the condition and prints "valid age".
* here few things are to be taken note of :-

mainly "INDENTATION" , python is a simple language but needs proper indentation for the compiler to understand the task.

**If – else**

if the condition don't match it automatically executes the else condition.

for example : You got confused between which outfit to select for your birthday party and you Got Two outfits which confuses you so you check few conditions , if one of those outfit fits on those condition you wear the outfit but if it does not your only option remains is the second outfit . So,

if outfit == perfect :

print("To wear tonight")

else :

print("second option")

**Elif**

Just like "if - else" statement , elif allows you to check if one condition fits and if does not , it goes to the next condition , again if it doesn't fit again it goes to next condition , and this continues until it meets it's matched condition and if there's no matched condition it reciprocates the default else condition.

for example:-

if a>b:

print("A is greater")

elif a<b:

print("A is less")

elif a==b:

print("A and B are equal")

else:

print("Invalid")

These are simple conditions with simple syntax just needs attention to "indentation" accordingly.

**Important Aspects of Conditionals**

-Operators like == (equal to), != (not equal to), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to) are used to compare values within conditions.

- Operators like and, or, not can be used to combine multiple conditions.

-In Python, the code block associated with an "if", "elif", or "else" statement is defined by indentation.

**Loops in python**

In Python, a loop is a programming construct that allows a block of code to be executed repeatedly until a specific condition is met.

1. **For Loop**

For loops are used to iterate over a sequence (such as a list, tuple, or string) and execute a block of code for each item in the sequence

**For Example:**

When you go through a daily routine like brushing your teeth, taking a shower, and getting dressed, you might have a fixed number of steps to complete. Each of these steps happens once, in a specific order.

You repeat the same set of actions a specific number of times, such as brushing each side of your teeth for a certain number of seconds.

fruits = ['apple', 'banana', 'cherry']

for fruit in fruits:

print(fruit)

**Output:**

apple

banana

cherry

In this example, the for loop iterates over the fruits list and prints each fruit.

1. **While Loops**

While loops are used to execute a block of code as long as a given condition remains true.

Think about waiting in line at a coffee shop. You check if the line is moving forward, and if it is not your turn yet, you keep waiting. Once it's your turn, you leave the line.

Keep waiting (loop) until it's your turn (the condition is met).

Example:

i = 0

while i < 5:

print(i)

i += 1

Output:

0

1

2

3

4

In this example, the while loop executes as long as i is less than 5. The loop prints the current value of i and then increments i by 1.

1. **Nested loops**

Nested loops are used to iterate over multiple sequences or to perform a task multiple times.

Suppose you’re preparing a multi-course meal. For each course, you need to prepare different ingredients, and for each ingredient, you might have to chop, boil, or sauté it.

For each course (outer loop), for each ingredient in that course (inner loop), you repeat the preparation steps**.**

**Example:**

for i in range(3):

for j in range(4):

print(f"({i}, {j})")

Output:

(0, 0)

(0, 1)

(0, 2)

(0, 3)

(1, 0)

(1, 1)

(1, 2)

(1, 3)

(2, 0)

(2, 1)

(2, 2)

(2, 3)

In this example, the outer loop iterates over the range 0-2, and the inner loop iterates over the range 0-3. The loop prints the coordinates of each point in the 3x4 grid.

**Loop Iterators**

Loop iterators are used to iterate over iterables such as lists, tuples, and strings.

Imagine you're reading a book. You start at the first page and read through each page sequentially until you reach the end of the book.

The iterator here is like the page number that keeps track of your current position in the book. Each page you read is like the next item the iterator gives you, and you keep moving through the pages (the loop) until you reach the end.

Example:

fruits = ['apple', 'banana', 'cherry']

fruit\_iterator = iter(fruits)

print(next(fruit\_iterator)) # prints 'apple'

print(next(fruit\_iterator)) # prints 'banana'

print(next(fruit\_iterator)) # prints 'cherry'

Output:

apple

banana

cherry

In this example, the iter() function creates an iterator object from the fruits list. The next() function returns the next item from the iterator.

**Enumerate Iterator**

The enumerate iterator returns both the index and value of each item in an iterable.

When organizing a shelf, you might want to place books in order. As you place each book, you note its position (1st, 2nd, 3rd, etc.) on the shelf along with its title.

The index is the position of the book on the shelf (1st, 2nd, etc.), and the value is the title of the book. As you arrange the books, you keep track of both the position and the title, just like an enumerator iterator does.

Example:

fruits = ['apple', 'banana', 'cherry']

for index, fruit in enumerate(fruits):

print(f"{index}: {fruit}")

Output:

0: apple

1: banana

2: cherry

In this example, the enumerate() function returns an iterator that produces tuples containing the index and value of each item in the fruits list.

**Zip Iterator**

The zip iterator returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables.

You have a list of names and a corresponding list of phone numbers. To create a directory, you pair each name with the correct phone number.

The names and phonenumbers are two lists. A zipiterator pairs the first name with the first phone number, the second name with the second phone number, and so on.

Example:

fruits = ['apple', 'banana', 'cherry']

colors = ['red', 'yellow', 'pink']

for fruit, color in zip(fruits, colors):

print(f"{fruit}: {color}")

Output:

apple: red

banana: yellow

cherry: pink

In this example, the zip() function returns an iterator that produces tuples containing one element from each of the fruits and colors lists.

**Exceptions in Python**

An exception refers to an unexpected event that occurs during the execution of a program**.**

Exceptions are errors that happen while the program is running.

Different types of exceptions exist, each representing a specific error condition.

To manage exceptions, Python provides the try...except block.

Imagine you’re buying something online. You add an item to your cart, proceed to checkout, and then enter your payment details. However, there are a few things that could go wrong during the payment process, such as:

1. CardDeclined**:** You might enter an expired or incorrect card.
2. InsufficientFunds**:** Your bank balance might not be enough to complete the purchase.
3. ServerError**:** The payment gateway might be temporarily down.

In such cases, instead of the aborting entire shopping expense , the system needs to handle these issues gracefully. The payment system might show a message like "Card declined" or "Insufficient funds" and prompt you to try again or choose another payment method.

**Example of Exception Handling**

try:

result = 10 / 0

except ZeroDivisionError:

print("Error: Cannot divide by zero!")

**The “ Try ”:**

The **try** statement is used to test a block of code for potential errors during execution. It allows you to handle exceptions (errors) gracefully instead of letting them crash the program.

Imagine you're playing a video game where you have to jump over a hole. Sometimes, you might fall into the hole, but you don't want the whole game to crash just because you made a mistake. So, there's a trampoline at the bottom of the hole to bounce you back up, and you can keep playing.

In Python, the **try** block is like that trampoline. It says, "Hey, I’ll try to do something risky, it will be fun. If something goes wrong, I won't let the program crash; I’ll handle it nicely!"

**Key Components:**

1. **try Block**: Contains the code that might raise an exception.
2. **except Block**: Contains the code to execute if an exception occurs in the try block. You can specify the type of exception to handle specific errors.

**The “except” Keyword**

The "except" keyword is used within a "try...except" block to specify a block of code that will be executed if an exception is raised.

**Key Points About "except"**

- The "except" keyword always appears alongside a "try" block.

- You can specify the exact type of exception to catch within the "except" block.

- If you want to catch any kind of exception, you can simply use except Exception.

Now, think of a situation where you are baking a cake, and you have to follow the recipe. In the recipe, it eplains "add 2 cups of sugar," but you only have 1 cup left.

In this case:

* The bakingprocess is like the code running.
* Findingout you don't haveenough sugar is like encountering an exception (error).
* The exceptblock allows you to handle the error, so instead of giving up, you might decide to use a substitute, like honey, or adjust the recipe.

try:

sugar = int(input("Enter amount of sugar (in cups): "))

if sugar < 2:

raise ValueError("Not enough sugar for the cake.")

except ValueError as e:

print(f"Error: {e}. Using honey as a substitute instead.")

**The “Finally” Keyword**

The "finally" keyword is used within a "try...except" block to define a code block that will always be executed.

Lets imagine you're leaving your house and locking the door. You might face two scenarios:

1. You lock the door and leave without issues.
2. Maybe you drop your keys or the lock breaks while you're trying to lock the door.

Regardless of what happens, after you're done trying to lock the door, you always make sure the door is locked before leaving and if you belong to Indian household then ofcourse you got this habit. This is a real-world analogy for the finally block. Even if an error occurs (like dropping your keys), you still want to ensure that the door is locked before you leave.

Key Points About "finally"

- The code within the "finally" block will run no matter if the "try" block throws an exception or completes normally.

- The most common use case for "finally" is to perform necessary cleanup actions.

Example of "finally"

try:

door\_locked = input("Did you lock the door? (yes/no): ")

if door\_locked.lower() != "yes":

raise ValueError("The door is not locked!")

print("You are safe. The door is locked.")

except ValueError as e:

print(f"Error: {e}")

finally:

print("Locking the door (if not already done).")

**The “else”**

 The "else" clause within a "try...except" block is used to execute a specific code block only if no exceptions are raised during the execution of the code within the "try" block; essentially, it acts as a way to run code when the "try" block successfully completes without encountering any errors.

Lets imagine you're going to buy a gift at the store for your soulmate whose delulu. Here's how the process might work:

* You check if you have enough money. If you do, the transaction process smoothly.
* If you don't have enough money, an error occurs (like the transaction failing), and you can handle it by informing yourself that you need more money.
* If you have enough money, the purchase completes, and you might want to print a message saying the transaction was successful.

**Key points about the "else" clause in exception handling**:

Placement:

The "else" clause must always come after all "except" clauses within a "try...except" block.

Functionality:

If the code within the "try" block runs without raising any exceptions, the code within the "else" block will be executed.

Use cases:

Verifying successful operations: You can use the "else" block to perform additional checks or actions that only make sense if the main operation in the "try" block succeeded.

Improving code readability: By separating error handling logic from successful execution logic, the "else" clause can make your code more organized and easier to understand.

Example:

money = 50 # Amount of money you have

price = 40 # Price of the gift

try:

if money >= price:

# starting the purchase process

money -= price # Deduct money from your account

print(f"Purchase successful! You bought the gift for {price} dollars.")

else:

raise ValueError("Not enough money to buy the gift!")

except ValueError as e:

print(f"Error: {e}. Please withdraw more money.")

else:

print("Enjoy your new gift!")

**Important considerations:**

Avoid redundant checks:

Since the "else" block only executes when no exceptions are raised, avoid adding checks within it that are already implied by the "try" block.

**Functions in Python**

A function is a block of reusable code that performs a specific task.

Imagine you're preparing a sandwich. To make the process easier and more organized, you can think of the process as a function.

1. **Making the sandwich**: The process of making a sandwich involves multiple steps (e.g., getting ingredients, spreading butter, placing veggies, etc.).
2. **Reusable process**: Once you understood the steps to make a sandwich, you can reuse this process every time you want a sandwich, without repeating the instructions over and over.

**Key Points About Functions**

- **Purpose**: Functions help break down complex programs into smaller, manageable units.

- **Syntax**: def function\_name(parameter1, parameter2):

- **Calling a Function**: To use a function, you write its name followed by parentheses and any necessary arguments.

Types of Functions in Python

- **Built-in Functions:** Functions already available in Python like print(), len(), max().

- **User-defined Functions**: Functions created by the programmer using the def keyword.

- **Lambda Functions (Anonymous Functions)**: Small, one-line functions defined without a name using the lambda keyword.

**For Example:**

# Define the function to make a sandwich

def make\_sandwich(bread, filling, sauce):

print("Taking two slices of {bread}.")

print("Adding {filling} to the sandwich.")

print("Spreading some {sauce}.")

print("Your sandwich is ready!")

# Call the function to make a sandwich

make\_sandwich("whole wheat bread", "cheese and tomato", "mayonnaise")

**Recursive Functions in Python**

A recursive function is a function that calls itself during its execution.

A real-life example of recursion could be **Matryoshka dolls**. These are a set of wooden dolls, each of which contains a smaller one inside. The process of opening each doll and finding another inside is recursive because you keep performing the same action—opening a doll—until you reach the smallest one.

Here’s how the analogy works:

* You start by opening the largest doll.
* Inside it, you find a smaller doll, which you then open.
* Inside that doll, you find another smaller one, and you keep going until there are no more dolls inside (the smallest doll, which doesn't contain any others

**For Example:**

def open\_doll(doll\_number):

if doll\_number == 1:

print("You have reached the smallest doll. There are no more dolls inside!")

else:

print(f"Opening doll number {doll\_number}...")

open\_doll(doll\_number - 1) #Call the function again with a smaller doll number

# Start the process with the largest doll (5th doll)

open\_doll(5)

Recursion helps simplify problems that involve repetitive tasks or nested structures. The Matryoshka dolls analogy is a fun and clear way to understand how recursion works: open one doll, find a smaller one, and keep going until you reach the smallest one, at which point the process stops. Similarly, a recursive function breaks down the task into smaller, manageable sub-tasks until it hits a base case.

**Important Considerations for Recursive Functions**

- Efficiency: Recursive functions can be less efficient than iterative solutions for certain problems.

- Infinite Recursion: If you don't define a proper base case, the function will keep calling itself indefinitely, leading to an error.

**Lambda Functions in Python**

The "lambda" keyword is used to define an anonymous function, meaning a small, single-line function without a name.

Lets assume, you work at a store and often need to apply a discount to the price of items. You might want a quick way to apply different discounts depending on the scenario, such as a 10% discount or a 20% discount.

Instead of defining a whole function each time, you could use a lambda function to calculate the price after applying the discount.

**For Example:**

apply\_discount = lambda price, discount: price - (price \* discount)

# Example prices and discounts

price1 = 100 # Original price

discount1 = 0.10 # 10% discount

price2 = 200

discount2 = 0.20 # 20% discount

# Applying the discounts

final\_price1 = apply\_discount(price1, discount1)

final\_price2 = apply\_discount(price2, discount2)

print(f"The final price after discount is: {final\_price1}")

print(f"The final price after discount is: {final\_price2}")

**Key Points**

- Syntax: lambda arguments: expression

- Arguments: The "arguments" part is a list of variables that the function will accept, similar to a regular function definition.

- Expression: The "expression" is the calculation or operation that will be performed on the arguments and returned as the result.

- No need for "return" statement: Unlike regular functions, a lambda function automatically returns the result of the expression.

**Match Statement in Python**

The match statement is a keyword used in the match-case statement, which allows for pattern matching against a value.

* you're at a café and you want to order a drink. The order could be different based on what kind of drink the customer wants (e.g., coffee, tea, juice). The match statement could help you decide which preparation steps to follow depending on the type of drink. This makes it easy and does not give any headacheeee

**Take an example code as**

def order\_drink(drink\_type):

match drink\_type:

case "coffee":

print("Preparing coffee with milk and sugar.")

case "tea":

print("Preparing tea with lemon.")

case "juice":

print("Preparing fresh juice.")

case \_:

print("Sorry, we don't serve that drink. Please choose coffee, tea, or juice.")

# Example orders

order\_drink("coffee") # Expected: Preparing coffee with milk and sugar.

order\_drink("tea") # Expected: Preparing tea with lemon.

order\_drink("juice") # Expected: Preparing fresh juice.

order\_drink("water") # Expected: Sorry, we don't serve that drink. Please choose coffee, tea, or juice.

The **match statement** in Python is a great way to handle multiple possible conditions in a clean and readable way, just like handling different orders at a café or responding to various status codes. It simplifies your code, making it easier to manage and extend when needed.

**Classes and objects**

- A class is a blueprint or template used to create objects.

- An object is a collection of variables and functions that interact with each other.

Now lets first understand what exactly is class?? But first make your selves comfortable as this is going to be mor fun than it seems. Nah, no not like you are going to win any 7 crore like in KBC but you are for sure more than 7 crore ofcourse , I know this seems weird Jalebie types but its simple , Knowledge which is important in life. SOOOOOOO

Imagine you want to create many cars. Instead of describing each car separately with all its features (like color, wheels, and engine), you can create a blueprint or a template for cars. This blueprint will contain all the common features that everycar will have.

In Python, this blueprint is called a class.

* The class defines the properties (attributes) of a car, like color and wheels.
* The class also defines what actions the car can do, like driving or honking.

So, a class is like a template or recipe that says, "Here’s how a car should be built!"

Now lets learn whats object?????

An object is a real car that is created using the blueprint (or class). You can create as many instances of cars (objects) as you want using the same blueprint, but each car can have different properties (like color or model). Each individual car you create from the blueprint is an object.

For example, you can have:

* A redcar with 4 wheels.
* A blue car with 4 wheels.
* A sports car with 4 wheels.

Even though they're all cars, each one is an object created from the same class blueprint.

# This is the Car class (blueprint)

class Car:

# This is the constructor (special function) to set the car's features when it's created

def \_\_init\_\_(self, color, wheels, model):

self.color = color # Every car has a color

self.wheels = wheels # Every car has wheels

self.model = model # Every car has a model

# This is a method (an action) that the car can do

def drive(self):

print(f"The {self.color} {self.model} car with {self.wheels} wheels is driving!")

# Another method for honking the car's horn

def honk(self):

print(f"The {self.color} {self.model} car honked its horn: Beep beep!")

# Creating a car object (an actual car) using the class blueprint

my\_car = Car("red", 4, "Sedan")

my\_sports\_car = Car("blue", 4, "Sports Car")

# Using methods of the car objects

my\_car.drive() # Red sedan drives

my\_sports\_car.drive() # Blue sports car drives

my\_car.honk() # Red sedan honks

my\_sports\_car.honk() # Blue sports car honks

**Encapsulation**

- Encapsulation is the practice of bundling data and methods together within a single class.

- It restricts direct access to the internal state of an object and only allows modification through controlled methods.

Encapsulation is like putting something important inside a box so that people can only access it in the way you want them to.

Also can be said,

Imagine your younger brother or sister have a toy that has a button. When you press the button, the toy does something fun, like making noise or lighting up. But, there’s a secretpartinside the toy (like batteries or circuits) that makes the toy work. You can’t see or touch the inside parts directly, but the button lets you interact with it in a controlled way.

**Key Points of Encapsulation:**

1. **Private Data**: Hide internal details (like the balance) from the outside world.
2. **Controlled Access**: Use methods (like deposit and withdraw) to change or access the data, ensuring no accidental mistakes or unauthorized changes.
3. **Safety and Simplicity**: Makes your code safer and easier to use, because it hides the complexity and provides a clear way to interact with the object.

**Data Abstraction**

- Data abstraction is the practice of hiding internal implementation details of a class or object.

- It only exposes essential features and functionalities to the user.

It hides the complex details of how something works and only exposes the necessary information or functionality to the user. This allows you to use an object without needing to understand its internal workings

Imagine you're using a television at home. You don’t need to know how the television works inside, like how the signals are received or how the screen lights up. All you need is a remote control to turn it on, change channels, or adjust the volume. You interact with the remote control, not the complex internal workings of the TV.

**Key Points of Data Abstraction:**

1. **Hide Complexity**: The user doesn't need to know how things work under the hood.
2. **Expose Only Necessary Information**: Only the important actions or properties are shown to the user.
3. **Simplify**: Makes your code easier to use and understand, hiding unnecessary details.

**Inheritance**

- Inheritance is a core OOP concept that allows a new class to inherit properties and methods from an existing class.

- It creates a hierarchical relationship where the child class can reuse and extend the functionality of the parent class.

Imagine your younger brother have a **base toy** (like a simple **car** toy). Now, you want to create different kinds of **specialized cars** (like a **sports car** or an **electric car**). Instead of starting from scratch for each new type, you can **inherit** the features of the original car and then add more specific features for each type.

In Python, **inheritance** allows a **child class** to use the properties and methods of a **parent class** (base class) while also adding its own unique features or behaviors.

**Key Concepts of Inheritance:**

* **Parent Class**: The base class that contains common features or behavior (e.g., basic Car).
* **Child Class**: A class that **inherits** from the parent class, getting the features of the parent, and can add its own features or change the existing ones (e.g., SportsCar, ElectricCar).

**Real-Life Example: Inheritance in Python**

Let's use a simple example with a **basic car** and different types of cars that **inherit** from it.

**Step 1: The Parent Class (Car)**

The Car class is the basic class that has common features like color and wheels, and common behaviors like drive().

**Step 2: The Child Classes (SportsCar and ElectricCar)**

* A **SportsCar** might have extra features like turbo\_boost().
* An **ElectricCar** might have extra features like charge().

**Example:**

# Parent class (basic car)

class Car:

def \_\_init\_\_(self, color, wheels):

self.color = color

self.wheels = wheels

def drive(self):

print(f"The {self.color} car with {self.wheels} wheels is driving.")

# Child class (sports car) inherits from Car

class SportsCar(Car):

def \_\_init\_\_(self, color, wheels, turbo):

# Inheriting from the parent class (Car)

super().\_\_init\_\_(color, wheels)

self.turbo = turbo

def turbo\_boost(self):

print(f"The {self.color} sports car is using turbo boost!")

# Child class (electric car) inherits from Car

class ElectricCar(Car):

def \_\_init\_\_(self, color, wheels, battery\_capacity):

# Inheriting from the parent class (Car)

super().\_\_init\_\_(color, wheels)

self.battery\_capacity = battery\_capacity

def charge(self):

print(f"The {self.color} electric car is charging with {self.battery\_capacity} kWh battery.")

# Creating objects of each class

my\_car = Car("red", 4)

my\_sports\_car = SportsCar("blue", 4, "super")

my\_electric\_car = ElectricCar("green", 4, 75)

# Using the methods from the parent class (Car)

my\_car.drive()

my\_sports\_car.drive() # Inherited from Car

my\_electric\_car.drive() # Inherited from Car

# Using the methods from the child classes

my\_sports\_car.turbo\_boost() # Only in SportsCar

my\_electric\_car.charge() # Only in ElectricCar

Inheritance have Types as well which are not so confusing but is a little confusing but we can handle that confusion because its not that confusing as much as we are to why? Why? Why? We choose this College??????????

**Single Inheritance**

In single inheritance, a class inherits from only one parent class.

**Example**:

Imagine you have a Car class and a SportsCar class. The SportsCar class inherits from the Car class.

python

class Car:

def drive(self):

print("The car is driving.")

class SportsCar(Car):

def turbo\_boost(self):

print("The sports car is using turbo boost!")

# Create an object of SportsCar

my\_sports\_car = SportsCar()

my\_sports\_car.drive() # Inherited from Car

my\_sports\_car.turbo\_boost() # Defined in SportsCar

**Output**:

The car is driving.

The sports car is using turbo boost!

* **Single Inheritance means the SportsCar inherits from only one class (Car).**

**Multiple Inheritance**

In multiple inheritance, a class can inherit from more than one parent class. This allows the class to get features from more than one parent.

Example:

Imagine a FlyingCar that can both drive and fly. It inherits from both Car and Airplane classes.

Python

class Car:

def drive(self):

print("The car is driving.")

class Airplane:

def fly(self):

print("The airplane is flying.")

class FlyingCar(Car, Airplane): # Inherits from both Car and Airplane

def turbo\_boost(self):

print("The flying car is using turbo boost!")

# Create an object of FlyingCar

my\_flying\_car = FlyingCar()

my\_flying\_car.drive() # Inherited from Car

my\_flying\_car.fly() # Inherited from Airplane

my\_flying\_car.turbo\_boost() # Defined in FlyingCar

**Output**:

The car is driving.

The airplane is flying.

The flying car is using turbo boost!

* **Multiple Inheritance means the FlyingCar inherits from both the Car and Airplane classes.**

**Multilevel Inheritance**

In multilevel inheritance, a class inherits from a parent class, which itself is derived from another class. So, it forms a chain of inheritance.

Example:

Imagine you have a Grandparent class, a Parent class, and a Child class. The Child class inherits from the Parent class, which in turn inherits from the Grandparent class.

python

class Grandparent:

def have\_legacy(self):

print("The grandparent has a legacy.")

class Parent(Grandparent):

def care(self):

print("The parent cares for the child.")

class Child(Parent): # Inherits from Parent, which in turn inherits from Grandparent

def play(self):

print("The child is playing.")

# Create an object of Child

my\_child = Child()

my\_child.have\_legacy() # Inherited from Grandparent

my\_child.care() # Inherited from Parent

my\_child.play() # Defined in Child

**Output**:

The grandparent has a legacy.

The parent cares for the child.

The child is playing.

- **Multilevel Inheritance forms a chain: Child -> Parent -> Grandparent.**

**Hierarchical Inheritance**

In hierarchical inheritance, multiple classes inherit from the same parent class. The parent class is shared by multiple child classes.

Example:

Imagine a Vehicle class that can be inherited by both Car and Bike classes.

python

class Vehicle:

def move(self):

print("The vehicle is moving.")

class Car(Vehicle):

def drive(self):

print("The car is driving.")

class Bike(Vehicle):

def ride(self):

print("The bike is being ridden.")

# Create objects of Car and Bike

my\_car = Car()

my\_car.move() # Inherited from Vehicle

my\_car.drive() # Defined in Car

my\_bike = Bike()

my\_bike.move() # Inherited from Vehicle

my\_bike.ride() # Defined in Bike

**Output**:

The vehicle is moving.

The car is driving.

The vehicle is moving.

The bike is being ridden.

- Hierarchical Inheritance means both Car and Bike inherit from the same parent class Vehicle.

**Hybrid Inheritance**

Hybrid inheritance is a combination of two or more types of inheritance. This could be a mix of multiple inheritance and multilevel inheritance, or single inheritance with other types.

**Example**:

Here, we combine multiple inheritance and multilevel inheritance.

python

class Animal:

def breathe(self):

print("The animal breathes.")

class Bird(Animal):

def fly(self):

print("The bird is flying.")

class Fish(Animal):

def swim(self):

print("The fish is swimming.")

class FlyingFish(Bird, Fish): # Combines both Bird and Fish

def jump(self):

print("The flying fish is jumping.")

# Create an object of FlyingFish

my\_flying\_fish = FlyingFish()

my\_flying\_fish.breathe() # Inherited from Animal

my\_flying\_fish.fly() # Inherited from Bird

my\_flying\_fish.swim() # Inherited from Fish

my\_flying\_fish.jump() # Defined in FlyingFish

**Output**:

The animal breathes.

The bird is flying.

The fish is swimming.

The flying fish is jumping.

* Hybrid Inheritance combines different inheritance types, in this case multiple (from both Bird and Fish) and single inheritance (from Animal).

Inheritance helps in reusing code, organizing your code into logical relationships, and allows for extending functionality in an easy and maintainable way.

**Polymorphism**

- Polymorphism is a programming concept that allows a single type of entity to represent multiple types in different contexts.

- It enables objects of different classes to be treated as members of a single superclass.

Imagine you have a shape like a circle or a square. Both of them can have a method called area(), but the way they calculate their area will be different:

* For a circle, the area is calculated as π \* radius^2.
* For a square, the area is calculated as side \* side.

Even though the method is the same (area()), the implementation depends on the type of shape. This is polymorphism — the same name, but different behaviors depending on the object type.

Key Points:

* Same method name but different implementations.
* The method's behavior changes depending on the type of object.
* Polymorphism allows different objects to respond to the same method or function in their own unique way.

**FOR EXAMPLE:**

# Parent class (Shape)

class Shape:

def area(self):

pass # This is just a placeholder method to be overridden by child classes

# Child class (Circle)

class Circle(Shape):

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return 3.14 \* self.radius \* self.radius # Area of circle: π \* radius^2

# Child class (Square)

class Square(Shape):

def \_\_init\_\_(self, side):

self.side = side

def area(self):

return self.side \* self.side # Area of square: side^2

# Create objects of Circle and Square

circle = Circle(5)

square = Square(4)

# Both call the same method 'area', but they behave differently

print("Area of Circle:", circle.area()) # Calls Circle's area method

print("Area of Square:", square.area()) # Calls Square's area method

Now lets get explained about what happened and what is happening as ou might have no idea but don’t you really worry we Got youuuu buddy

* **Parent class (Shape)**: This class has a method area() but does not define how to calculate the area. It just **acts as a blueprint** for all shapes.
* **Child class (Circle)**: The Circle class overrides the area() method and provides its own implementation to calculate the area of a circle.
* **Child class (Square)**: The Square class also overrides the area() method and provides its own implementation to calculate the area of a square.
* Even though **both classes** have a method named area(), the behavior of this method is **different** for each class based on their own implementation.