PYTHON FUNCTIONS

INTRODUCTION TO FUNCTIONS

What we will Cover ?

1. FUNCTIONS
2. TYPES OF FUNCTIONS
3. PARAMETERS AND ARGUMENTS
4. RECURSIVE FUNCTIONS
5. LOCAL AND GLOBAL VARIABLES
6. LAMBDA
7. FILTER , MAP AND REDUCE ()

PYTHON
FUNCTIONS

PYTHON FUNCTIONS

If a group of statements is repeatedly required then it is not recommended to write these statements everytime seperately. We have to define these statements as a single unit and we can call that unit any number of times based on our requirement without rewriting. This unit is nothing but function.

The main advantage of functions is code Reusability.

PYTHON FUNCTIONS

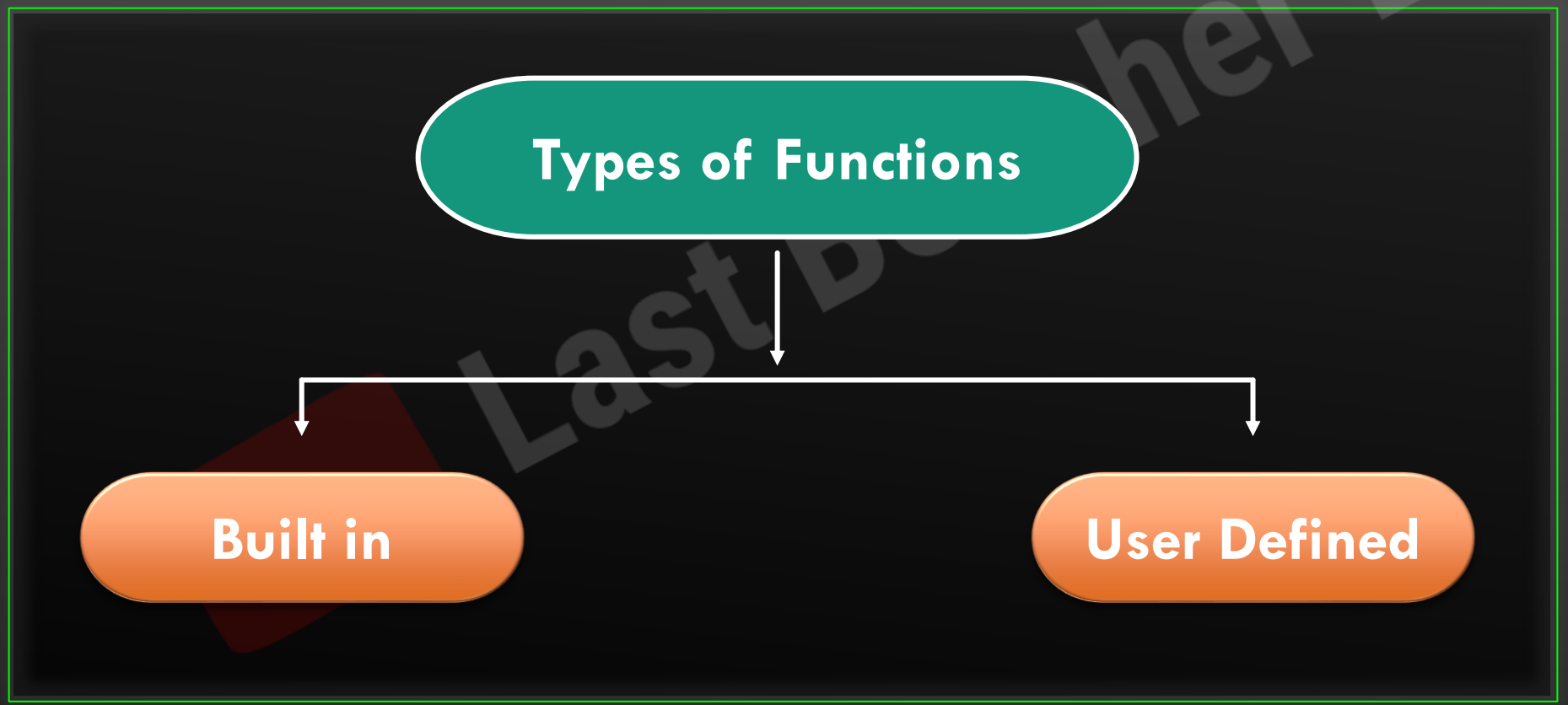
PYTHON FUNCTIONS

Functions in Python are a group of statements that are meant to be run multiple times in a program. Functions allow you to pass parameters into them and based on that return you a result.

A function in Python is enclosed within a def block. The statements that go inside a function body are indented by 4 spaces or 1 tab.

PYTHON FUNCTIONS

TYPES OF FUNCTIONS



BUILT IN FUNCTIONS

The functions which are coming along with Python software automatically , are called built in functions or pre defined functions

Example:

id()

type()

input()

eval()

etc..

Built In

Functions

USER DEFINED FUNCTIONS

The functions which are developed by programmer explicitly according to business requirements ,are called user defined functions

There are no curly braces in Python and so you need to indent this code block otherwise it won't work. You can make the function return a value.

Let's take an example for you.

User Defined

Functions

SYNTAX


```
def function_name(params):
    """
    Docstring
    """
    <body>
    return <something>
```

POINTS TO UNDERSTAND THE USER DEFINED FUNCTION

1. def marks the start of a function.
2. It is followed by a function_name to uniquely identify the function. The rules for writing function names are the same as writing identifiers.
3. Within parenthesis, params or parameters are used to pass values to a function. They are optional.
4. Colon (:) marks the end of the function header.
5. Documentation string (docstring) to describe what the function does. It is optional.
6. A return statement to return a value from the function. It is optional.

USER DEFINED FUNCTIONS

Note: While creating functions we can use 2 keywords

1. *def (mandatory)*
2. *return (optional)*

Write a Function to print Hello World


```
def func():
    print("Hello World")


func()
func()
func()
```

PARAMETERS

Parameters are inputs to the function.
If a function contains parameters ,
then at the time of calling , compulsory we should provide values otherwise we will get error.

Write a Function to take name and greet

Parameters

func()

```
def func(name):
    print(f"Hello {name} Welcome to my Channel")

func("Bhupi")
func("Katrina")
func("Alia")
```

func()

Arguments

Write a Function to print Square

Parameters

func()

```
def squarekro(num):
    print("Square = ",num*num)


squarekro(5)
squarekro(4)
```

func()

Arguments

ADVANTAGES

1. By using functions, we can avoid rewriting same logic/code again and again in a program.
2. We can call functions any number of times in a program and from any place in a program.
3. We can track a large program easily when it is divided into multiple functions.
4. Reusability is the main achievement of Python functions.

Return

Function can take input values as parameters and executes business

logic, and returns output to the caller with return statement.

As soon as a return statement is reached in a function, the function stops executing. Then, the next statement after the function call is executed.

PREDICT THE OUTPUT


```
# What is the output


def hello():
    print("Hello World")
```

CONCLUSION

The Conclusion is that we need to call the Function if we need to execute it. Defining a function and providing a body of the function is not enough so we need to call them as per our requirement

Syntax : hello()

PREDICT THE OUTPUT

What is the value of b ?

```
def hello():
    print("10")

b = hello()
print(b)
```

CONCLUSION

Most of you thinking 10 Right?

But it's not

CONCLUSION

The Answer is None because we have not initialize the value of b and hello() is not returning anything so to overcome this we need to return the value in a func if we want to reuse them

Returning Multiple Values

In other languages like C,C++ and Java, function can return atmost one value. But in Python, a function can return any number of values.

Returning Multiple Values

Example to Understand

```
def sum_sub(a,b):
    sum = a+b
    sub = a-b
    return sum,sub

x,y = sum_sub(10,5)
print("Sum = ",x)
print("Sub = ",y)
```

Returning

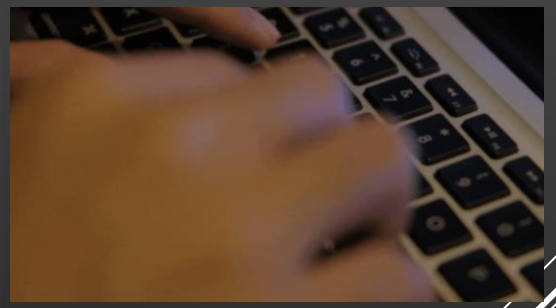
Multiple Values

Programming Lab - 34

86. Write a function to accept 2 numbers as input and return sum.

87. Write a function to check whether the given number is even or odd?

88. Write a function to check whether the given number is prime or not and return



Arguments and its Types

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that are sent to the function when it is called.

Arguments

In Python

Arguments and its Types

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Arguments

In Python

Types of Arguments

1. Positional Arguments
2. Keyword Arguments
3. Default Arguments
4. Variable Length Arguments

Positional Arguments

These are the arguments passed to function in correct positional order.

```
def sub(a,b):
    print(a-b)
sub(100,200)
sub(200,100)
```

The number of arguments and position of arguments must be matched. If we change the order then result may be changed. If we change the number of arguments then we will get error

Keyword Arguments

We can pass argument values by keyword i.e by parameter name.

```
def msg(name,msg):
    print(f"Hello {name}, {msg} ")

msg(name="Bhupi",msg="Kya Haal hai ")
msg(msg="Gulab Jamun Kha liya?",name="Megha")
```

You can Change the order by specifying parameter name at the time of calling a Func()

Keyword Arguments

Here the order of arguments is not important but number of arguments must be matched.

Note:

We can use both positional and keyword arguments simultaneously. But first we have to take positional arguments and then keyword arguments , otherwise we will get syntaxerror.

Default Arguments

Sometimes we can provide default values for our positional arguments.

```
def msg(name="Guest"):
    print(f"Welcome {name}, How May I Help You")

msg("Katto")
msg()
```

If we are not passing any name then only default value will be considered.

Default Arguments

After default arguments we should not take non default arguments

```
def msg(name="Kiara",msg="Good Morning"):
```

```
def msg(name,msg="Good Morning"):
```

```
def msg(name="Shalinee",msg):
```

SyntaxError: non-default argument follows default argument

Variable Length Arguments

Sometimes we can pass variable number of arguments to our function , such type of arguments are called variable length arguments.

```
def func(*n):
```

We can declare a variable length argument with * symbol as follows

def func(*n):

Variable Length Arguments

We can call this function by passing any number of arguments including zero number.
Internally all these values represented in the form of tuple.

Suppose We need to Perform add operation in marks of students
But a student may be absent in a particular subject
So the argument can be of 0 , 2 , 5 or anything else

Variable Length Arguments

Example to Understand

```
def func(*n):
    total = 0
    for i in n:
        total = total + i
    print(total)

func(0)
func(40,50)
func(1,2,3)
func(100,200,300,400)
```

Variable Length Arguments

We can mix variable length arguments with positional arguments

```
def func(i,*n):
    print("i = ",i)
    print(n)

func(0)
func(40,50)
func(1,2,3)
func(100,200,300,400)
```

Variable Length Arguments

Note: After variable length argument , if we are taking any other arguments then we should provide values as keyword arguments.

```
def func(*n,i):
    print("i = ",i)
    print(n)

func(0)
func(40,50)
```

```
def func(*n,i):
    print("i = ",i)
    print(n)

func(0,i=100)
func(40,50,i=200)
```

Variable Length Arguments

Note: We can declare key word variable length arguments also. For this we have to use **.

```
def func(**n):
```

We can call this function by passing any number of key-value arguments. Internally these keyword arguments will be stored inside a dictionary.

Variable Length Arguments

Example to Understand

```
def func(**n):
    for k,v in n.items():
        print(f"Key = {k} , Value = {v}")

func(id=150,name="Simran",Gender="Pta nai")
```

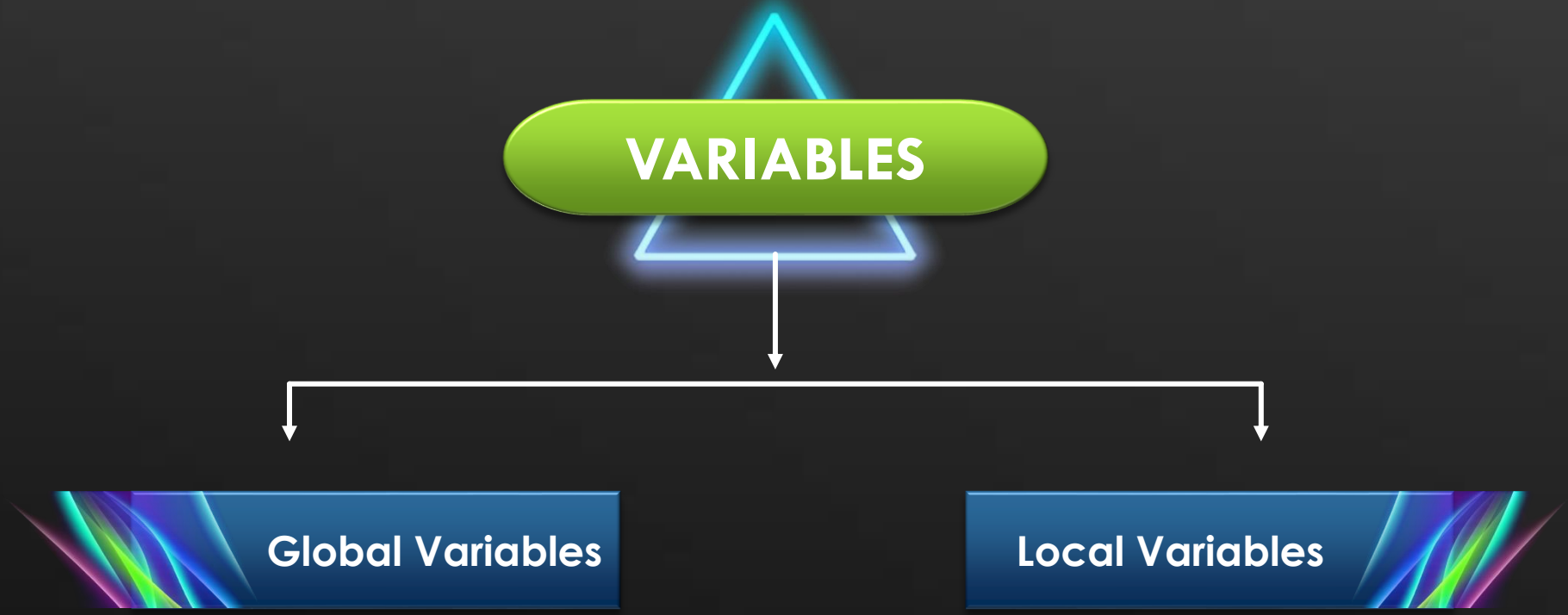
CASE STUDY

FUNCTION DEFINATION

```
def func(num1,num2,num3=4,num4=8):
    print(num1,num2,num3,num4)
```

1	`func(3,2)`	3 2 4 8	RUN
2	`func(10,20,30,40)`	10 20 30 40	RUN
3	`func(25,50,num4=200)`	25 50 4 200	RUN
4	`func(num4=11,num1=22,num2=33)`	22 33 4 11	RUN
5	`func()`	Type Error	RUN
6	`func(num3=10,num4=20,50,100)`	Syntax Error	RUN
7	`func(25,50,num3=200,num5=300)`	Type Error	RUN

Types of Variables



Global Variables

The variables which are declared outside of function are called global variables. These variables can be accessed in all functions of that module.

```
a = 10 ←—— Global Variable
def f1():
    print(a)
def f2():
    print(a)

f1() ——→ 10
f2() ——→ 10
```

Local Variables

The variables which are declared inside a function are called local variables.

Local variables are available only for the function in which we declared it. i.e from outside of function we cannot access.

Local Variables


```
def f1():
    a = 10
    print(a)
def f2():
    print(a)

f1()
f2()
```

Local Variable

f1() ⟶ 10

f2() ⟶ Name Error

PREDICT THE OUTPUT


```
a=100

def f1():
    a = 200

f1()
print(a)
```

A 200

B 100

C Name Error

D Syntax Error

PREDICT THE OUTPUT


```
def f1():
    a = 500

print(a)
```

A None

B 500

C Name Error

D Syntax Error

Global Keyword

On the Basis of previous 2 Problems, The Keyword Global comes in the picture

We can use global keyword for the following 2 purposes:

1. To make global variable available to the function so that we can perform required modifications
2. To declare global variable inside function

Global Keyword


```
a=100

def f1():
    global a
    a = 200

f1()
print(a)
```

Global Variable
'a' available to f1()

200

Global Keyword


```python
a = 50

def f1():
    global a
    a = 200

def f2():
    a = 300

print(a)
```

CONCLUSION

If your answer is 200 or 300

Global Keyword

Now you might be thinking that what happen if global variable and local variable is of same name then how to access and operate

```
a=10 <-- Global Variable

def f1():
    a=555 <-- Local Variable
    print(a)
    print(globals()['a'])

f1()
```

Recursion

A function that calls itself is known as Recursive Function.

In this method, we repeatedly call the function within the same function, and it has a base case and a recursive condition. The recursive condition helps in the repetition of code again and again, and the base case helps in the termination of the condition.

Working of Recursion

Recursion performs a number of repetitive calls to the function from within the function. The recursive condition performs the repeating calls to the function until the base case is met. The base case is present inside the function, and once the condition of the base case is satisfied, it stops the execution.

Why Recursion

Recursion can be used in almost every problem, but there are some cases where the recursion is actually helpful. It is generally used when dealing with complex problems and problems that form a hierarchical pattern; it solves the original problem via the smaller subproblems.

Programming Lab - 35

89. Write a Python Function to find factorial of given number with recursion.

90. WAP to Find No of Digits in a given integer using Recursion

LAMBDA

Sometimes we can declare a function without any name , such type of nameless functions are called anonymous functions or lambda functions.

The main purpose of anonymous function is just for instant use (i.e for one time usage)

We can define by using lambda keyword

λ

LAMBDA

SYNTAX

Lambda argument : expression

For Example

lambda n : n*n

Programming Lab - 36

91. WAP to create a lambda function to find square of given number

92. WAP to create a lambda function to find sum of 2 given numbers

93. WAP to create a Lambda function to find even or odd

94. WAP to create a Lambda function to find Biggest b/w two no

LAMBDA

N O T E

Lambda Function internally returns expression value and we are not required to write return statement explicitly.

Sometimes we can pass function as argument to another function. In such cases lambda functions are best choice.

We can use lambda functions very commonly with filter(),map() and reduce() functions , bcoz these functions expect function as argument.

FILTER()

We can use filter() function to filter values from the given sequence based on some condition.

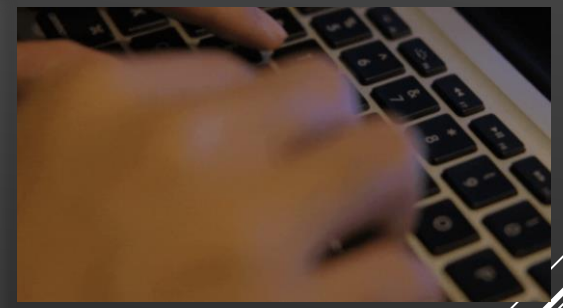
where function argument is responsible to perform conditional check sequence can be list or tuple or string.

FILTER()

SYNTAX

```
filter(function,sequence)
```

Programming Lab - 37



95. WAP to filter even values from a List

96. WAP to filter no which is divisible by 5 from a list

MAP()

For every element present in the given sequence , apply some functionality and generate new element with the required modification. For this requirement we should go for map() function.

Eg: For every element present in the list perform double and generate new list of doubles

MAP()

SYNTAX

```
map(function,sequence)
```

The function can be applied on each element of sequence and generates new sequence.

Programming Lab - 38

97. WAP to double all elements using map()

98. WAP to show square all elements using map()

REDUCE()

reduce() function reduces sequence of elements into a single element by applying the specified function.

reduce() function present in functools module and hence we should write import statement.

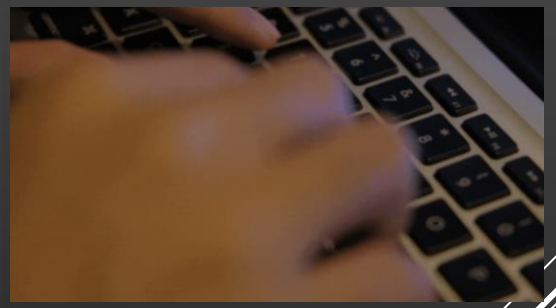
REDUCE()

SYNTAX

```
reduce(function,sequence)
```

Programming Lab - 39

99. WAP to calculate sum of all elements using reduce()



Function Aliasing

For the existing function we can give another name, which is nothing but function aliasing.

```
def jetha(name):
    print(f"Aur {name} kaisi chlri h Python ")

babita = jetha

jetha("Bhupi")  ⟶ Valid
babita("Bhupi") ⟶ Valid
```

Function Aliasing

```
def jetha(name):
    print(f"Aur {name} kaisi chlri h Python ")

babita = jetha
del jetha

jetha("Bhupi")  ⟶ Name Error
babita("Bhupi") ⟶ Valid
```

Nested Functions

We can declare a function inside another function , such type of functions are called Nested functions.

Or

Function within a function is called nested function

Nested Functions


```
def outer():
    print("1")
    def inner():
        print("2")

outer()
```

A. 1 2

B. 1

C. Type Error

D. Syntax Error

Nested Functions

If your answer is 1 2

CONCLUSION

We have declared inner () inside outer () but we have not called it
That's why inner () is not executed at all .

You can conclude from the next slide

Nested Functions

```
def outer():
    print("outer () started")
    def inner():
        print("inner () started")
    print("Calling inner ()")
    inner()
outer()
```

Executes - 1

Executes - 3

Executes - 2

Nested Functions

```
def outer():
    print("outer () started")
    def inner():
        print("inner () started")
    print("Calling inner ()")
    inner()


outer()
inner()
```

Executes - 1

Executes - 3

Executes - 2

Name Error

ACTiViTY TiME


```
def outer():
    print("outer")
```

On the basis of this function What is the difference between

f1 = outer V/s f1 = outer()

DECORATORS

Decorator is a function which can take a function as argument and extend its functionality and returns modified function with extended functionality.

The main objective of decorator functions is we can extend the functionality of existing functions without modifies that function.

DECORATORS

After Makeup

DECORATORS


```
def func(name):
    print(f"Hn bhyi {name} kya hal hai..")
```

This function can always print same output for any name

But we want to modify this function to provide different message if name is Megha.
We can do this without touching func() function by using decorator.


DECORATORS
@newt_sch
aurat ka chakkar babu bhaiyaa,
aurat ka chakkar...

DECORATORS

How to call same function with decorator and without decorator:

We should not use @decor

ACTiViTY TiME

A real life example to understand Decorators

DECORATORS

CHAINING

We can define multiple decorators for the same function and all these decorators will form Decorator Chaining.

Eg:

```
@decor2
@decor1
def num():
```

For num() function we are applying 2 decorator functions.

DECORATORS

CHAINING

Output Function with
extended functionality

num() → decor1 (func()) → decor2 (func())

num function
Will be the parameter for decor 1

Inner function of decor1
Will be the parameter for decor 2

GENERATORS

Generator is a function which is responsible to generate a sequence of values.

We can write generator functions just like ordinary functions, but it uses yield keyword to return values.

GENERATORS

Generator Function

Sequence of values

Yield

GENERATORS

To Create a Simple Generator

```
def mygen():
    yield 10
    yield 20
    yield 30

g = mygen()
print(type(g))
```

10 Added

20 Added

30 Added

class 'generator'

GENERATORS

To access elements in a Generator

```
def mygen():
    yield "A"
    yield "B"
    yield "C"

g = mygen()
print(next(g)) ← A
print(next(g)) ← B
print(next(g)) ← C
print(next(g)) ← Stop Iteration Exception
```

Programming Lab - 40

74. WAP to generate first N numbers using generator

75. WAP to generate even numbers upto N numbers using generator

76. WAP to start countdown from N number to 1using generator

77. WAP to generate Fibonacci series using generator

Difference b/w Normal () and Generator ()

The difference between a normal function and a generator function is that while a return statement terminates a function, yield statement will pause the function, save its current state and later continue from there in the next call.

If you use return in generator (), execution stops completely.

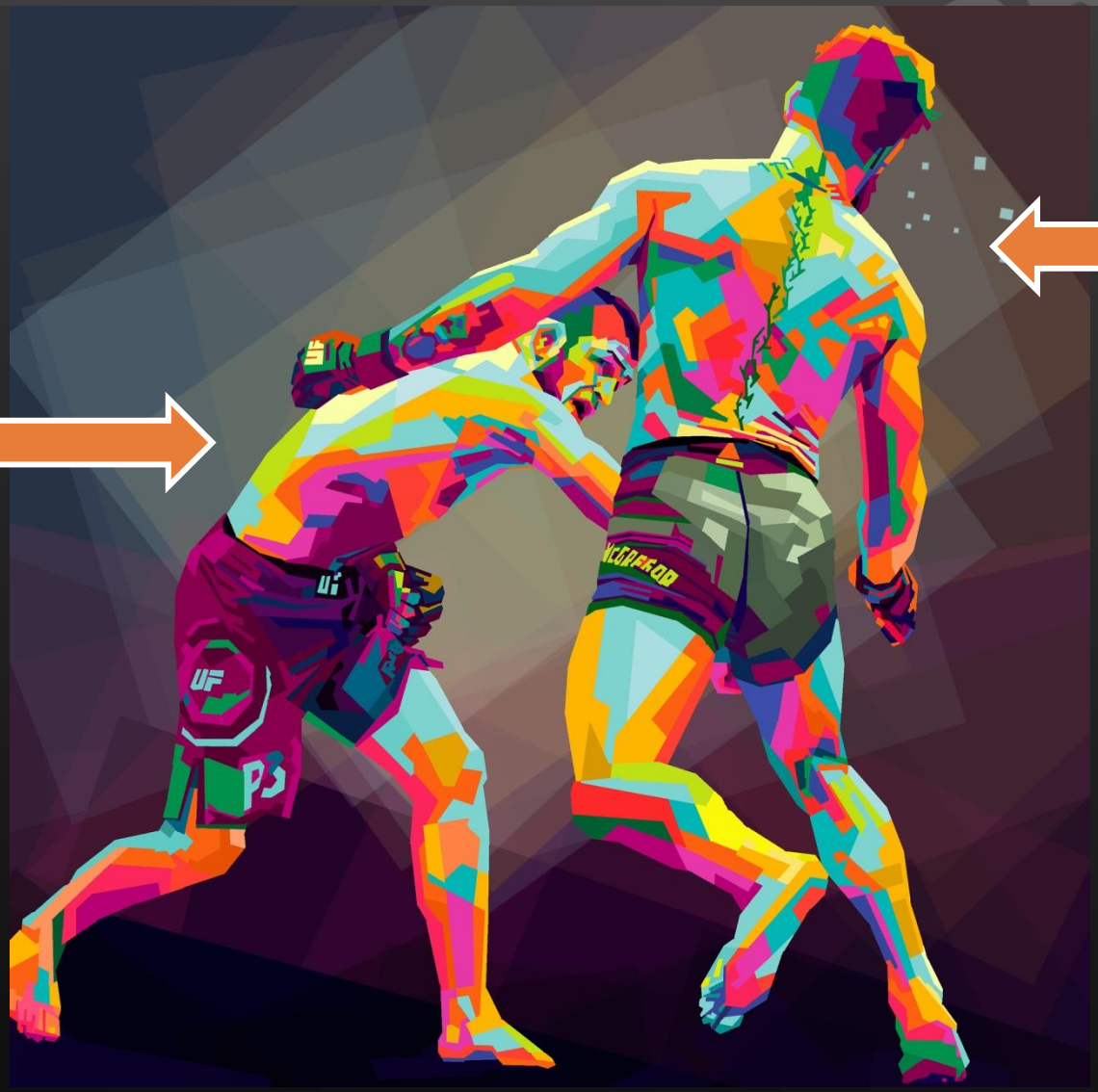
Advantages of Generators

1. Performance will be improved when compared with other data types
2. Memory utilization will be improved when compared with other data types
3. If you want to handle or read very large amount of data or handling crores of records from a database then generator is the best choice

Disadvantages of Generators

1. Generator objects will not store any data
2. Every time you want to reuse the elements in a collection it must be regenerated.
3. For the generator's work, you need to keep in memory the variables of the generator function.

Generators V/s Traditional Approach

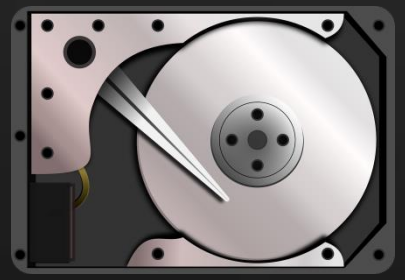


Generators

Traditional Approach

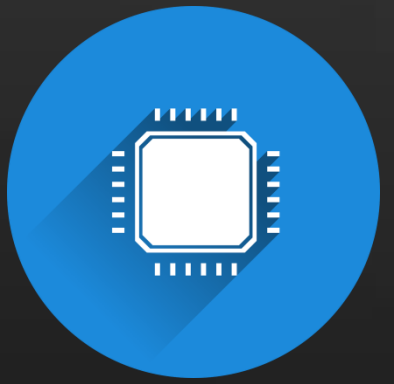
Generators V/s Traditional Approach

Memory Utilization



Demo Program

Performance



PREDICT THE OUTPUT


```
def gen():
    yield "a"
    yield "a"
    yield "a"

print(next(gen))
```

A a

B aa

C Type Error

D Syntax Error