**PYTHON**

**Who invented python?**

* Dutch Programmer Guido Van Rossum in the late 1980’s.
* First version 0.9.0 was released in February 1991

**What is python?**

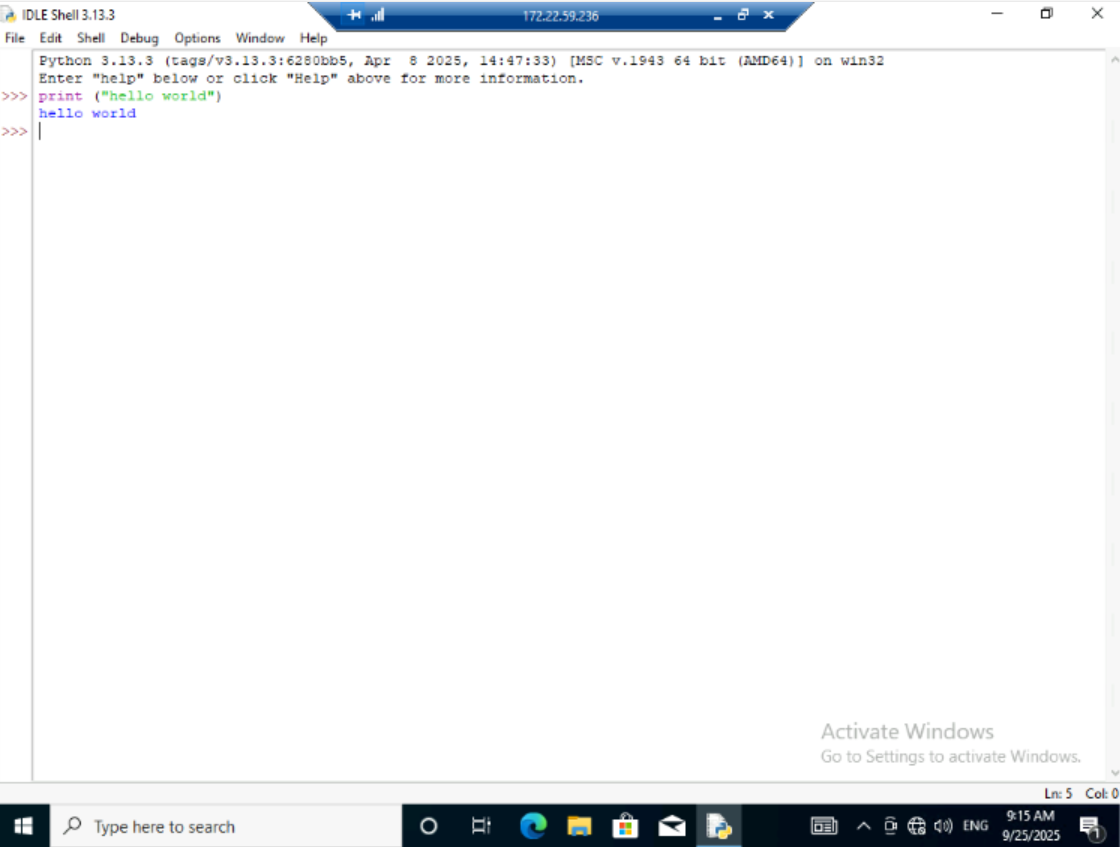
**Used for?**

* Software development
* System Scripting
* Used on a server to create web Applications.
* Used alongside software to create workflows
* Connects to database systems and also read and modigy files.
* To handle big data and perform comples mathematics.

**Why Python:**

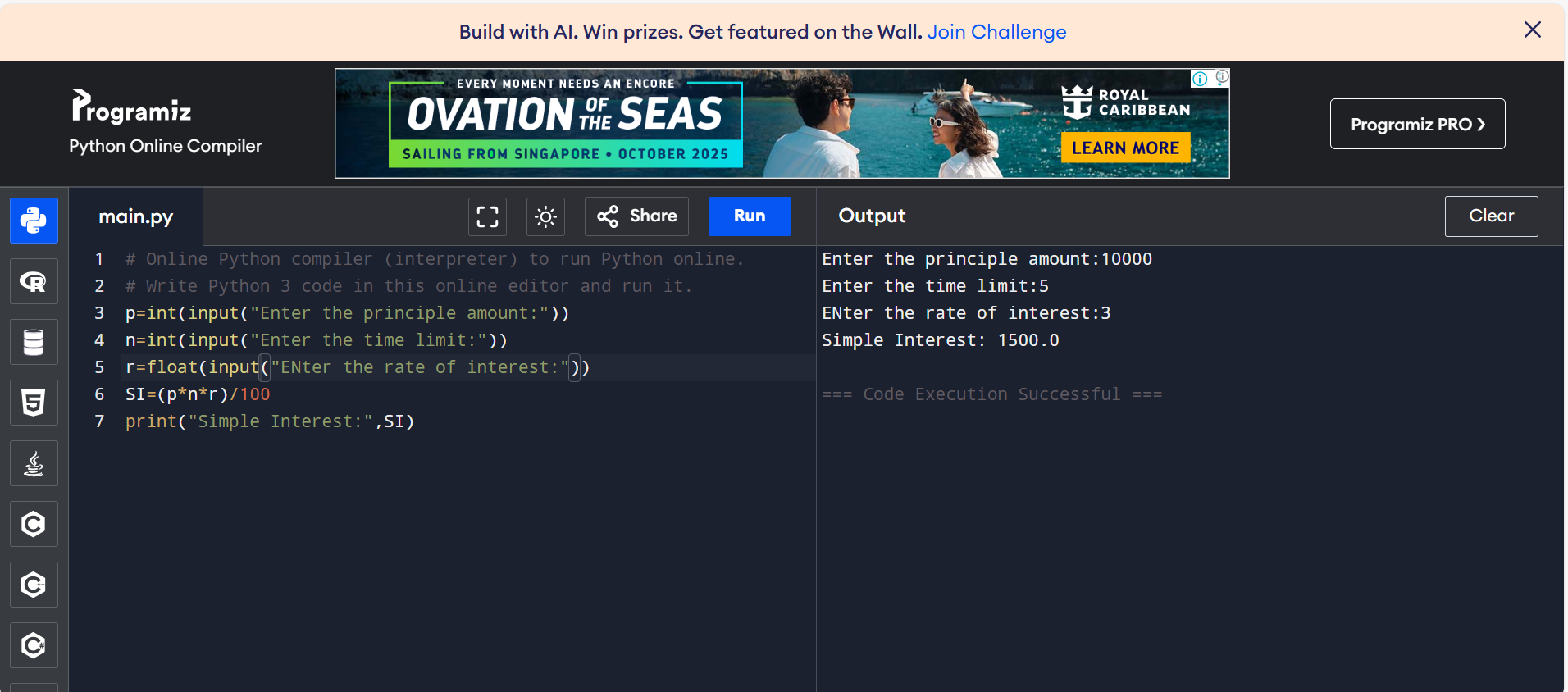
* Different Platform
* Simple syntax
* Fewer lines to develop a program than any other programming language.
* Runs on an interpreter system
* Procedural way or Object oriented way or functional way.

**First program:**



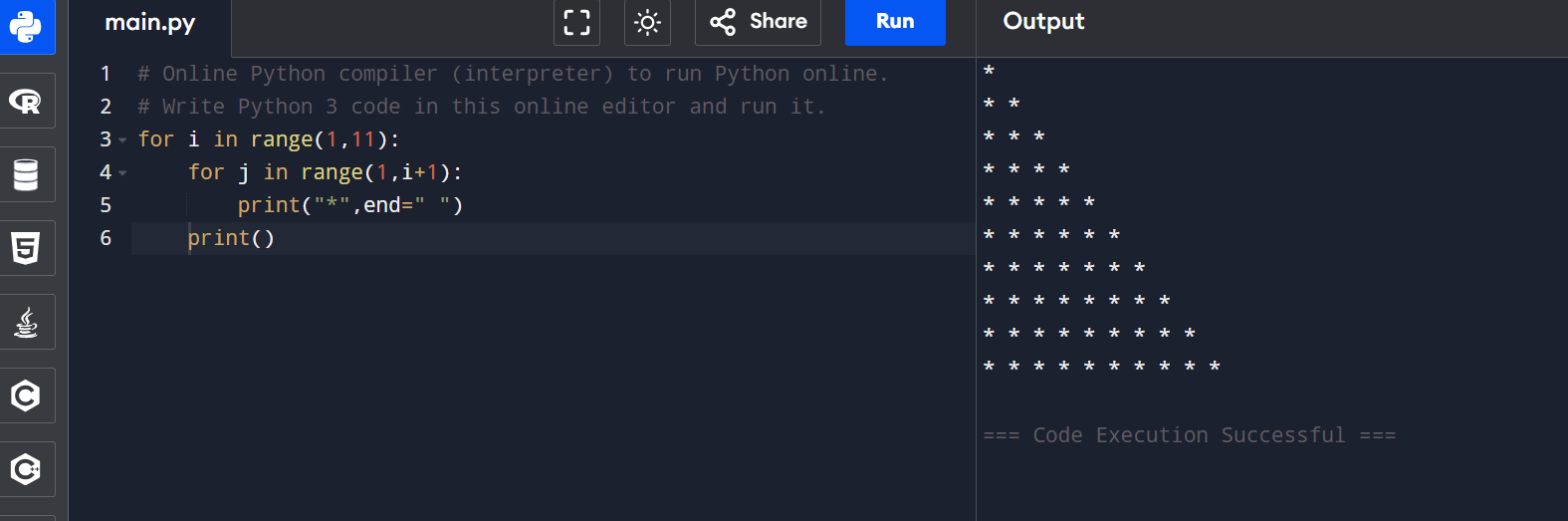
**Basics**

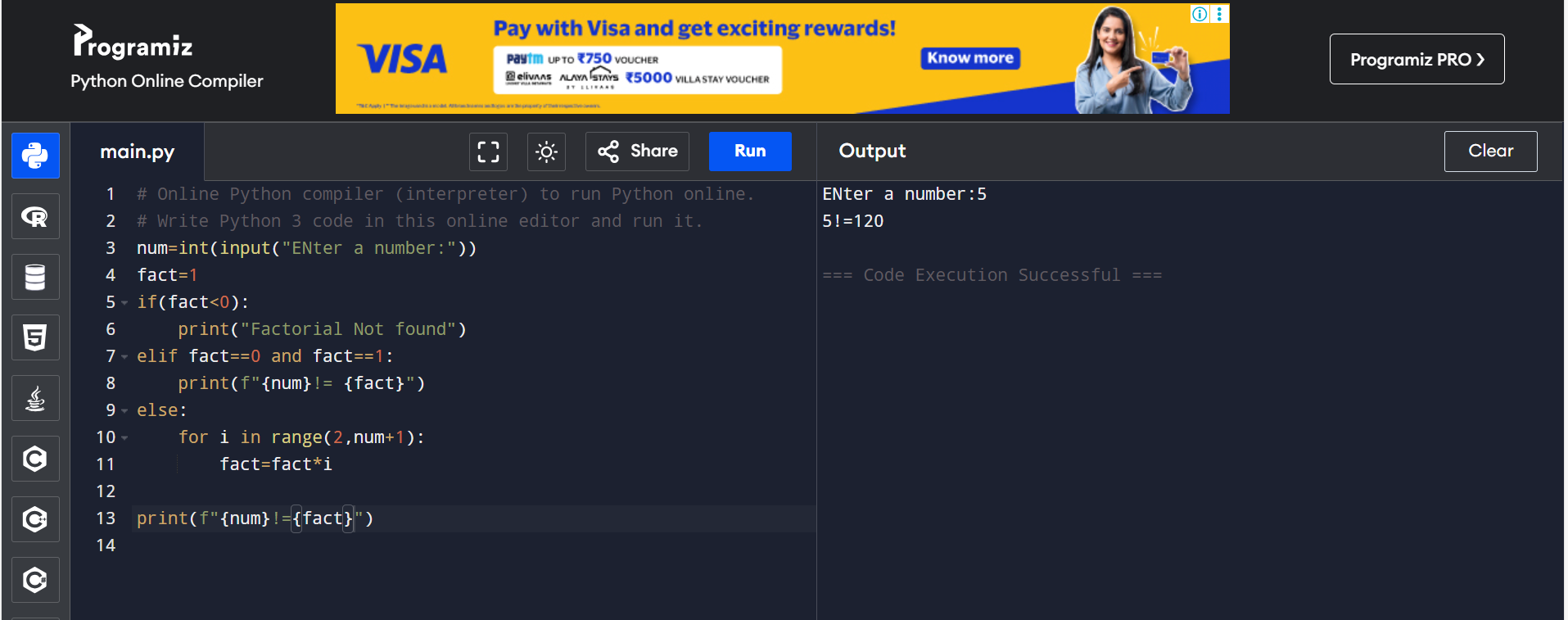
**Program for Simple Interest**



A screenshot of a computer

AI-generated content may be incorrect.





# Online Python compiler (interpreter) to run Python online.

# Write Python 3 code in this online editor and run it.

name=input("ENter the Name of the student:")

english=int(input("ENter the English Marks:"))

Maths=int(input("ENter the Maths Marks:"))

Science=int(input("ENter the Science Marks:"))

Social=int(input("ENter the Social Marks:"))

total=english+Maths+Science+Social

Average=total/4

percentage=(total/400)\*100

print(f"Name:{name}")

print(f"English Marks:{english}")

print(f"Maths Marks:{Maths}")

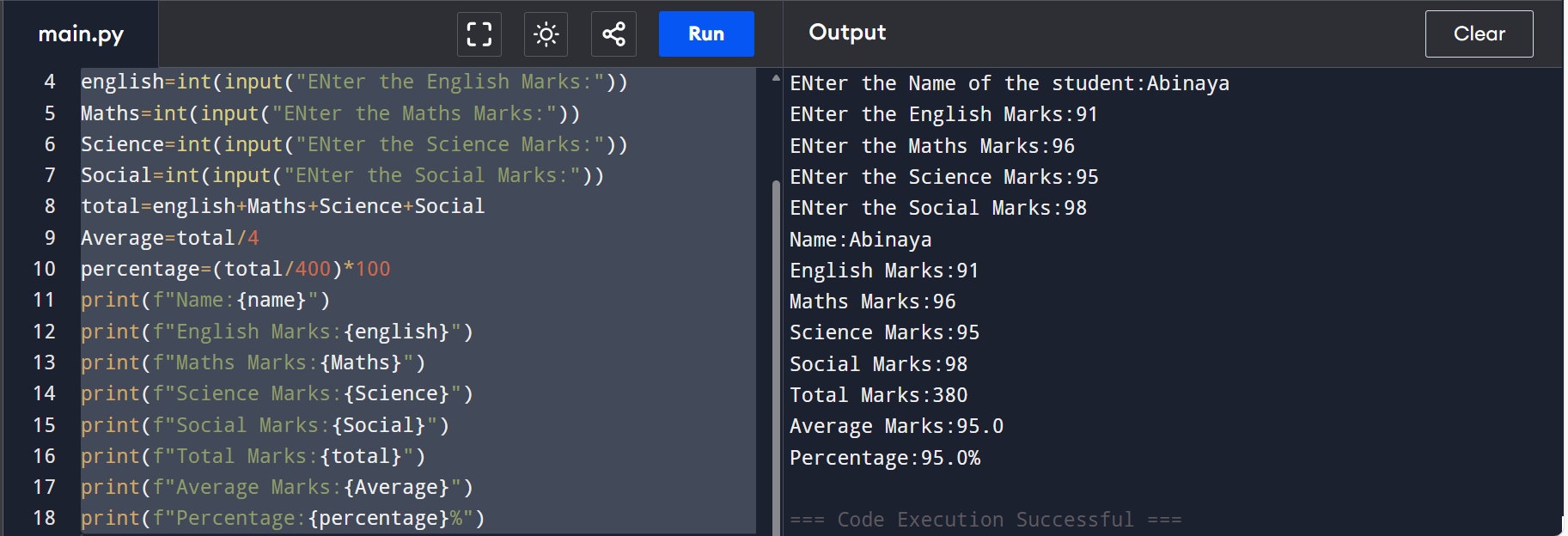
print(f"Science Marks:{Science}")

print(f"Social Marks:{Social}")

print(f"Total Marks:{total}")

print(f"Average Marks:{Average}")

print(f"Percentage:{percentage}%")



**List methods:**

|  |  |
| --- | --- |
| [append()](https://www.tpointtech.com/python-list-append-method) | This method is utilized to add an element x to the end of the list. |
| extend() | This method is utilized to add all elements of an iterable (list, tuple, etc.) to the list. |
| [insert()](https://www.tpointtech.com/python-list-insert-method) | This method is utilized to insert an element x at index i. |
| remove() | This method is utilized to remove the first occurrence of x. It raises ValueError if x is not found. |
| [pop()](https://www.tpointtech.com/python-list-pop-method) | This method is utilized to remove and returns the element at index i (default is the last element). |
| [clear()](https://www.tpointtech.com/python-list-clear-method) | This method is utilized to remove all elements, making the list empty. |
| index() | This method is utilized to return the first index of x between start and end. It raises ValueError if not found. |
| [count()](https://www.tpointtech.com/python-list-count-method) | This method is utilized to return the number of occurrences of x in the list. |
| sort() | This method is utilized to sort the list in place (default is ascending). |
| reverse() | This method is utilized to reverse the order of the list in place. |
| copy() | This method is utilized to return a shallow copy of the list. |

**Tuples:**

#Tuple

My\_list=("Naruto","Luffy","Ichigo","Tanjiro","Asta")

print(My\_list)

# My\_list[1]="Gon"

# print(My\_list) => Error

'''Accessing Tuple'''

print("Accessing 0th index:",My\_list[0])

print("Accessing 0th index:",My\_list[-1])

# Slicing the tuple

print("Positive slicing:",My\_list[1:3])

print("Negative slicing:",My\_list[1:-2])

sample\_tuple=("Apple","Mango","Orange","Guava")

print("Sample\_tuple",sample\_tuple)

# del sample\_tuple

# print(sample\_tuple)

# error=>

# Traceback (most recent call last):

# File "<main.py>", line 18, in <module>

# NameError: name 'sample\_tuple' is not defined

#appending an element to tuple

print("Before:",sample\_tuple)

sample\_tuple\_2=list(sample\_tuple)

sample\_tuple\_2.append("Papaya")

sample\_tuple=tuple(sample\_tuple\_2)

print("List:",sample\_tuple\_2)

print("After:",sample\_tuple)

temp\_list=("Gon",)

print(My\_list)

My\_list=My\_list+temp\_list

print(My\_list)

# for loop

i=1

print("The Heroes:")

for hero in My\_list:

print(i,"-",hero)

i+=1

#while loop

print("The Fruits:")

j=0

while j<len(sample\_tuple):

print(j,"-",sample\_tuple[j])

j+=1

#Count the specific

print(My\_list.count("Asta"))

#Index of the specific

print(My\_list.index("Asta"))

**Output:**

('Naruto', 'Luffy', 'Ichigo', 'Tanjiro', 'Asta')

Accessing 0th index: Naruto

Accessing 0th index: Asta

Positive slicing: ('Luffy', 'Ichigo')

Negative slicing: ('Luffy', 'Ichigo')

Sample\_tuple ('Apple', 'Mango', 'Orange', 'Guava')

Before: ('Apple', 'Mango', 'Orange', 'Guava')

List: ['Apple', 'Mango', 'Orange', 'Guava', 'Papaya']

After: ('Apple', 'Mango', 'Orange', 'Guava', 'Papaya')

('Naruto', 'Luffy', 'Ichigo', 'Tanjiro', 'Asta')

('Naruto', 'Luffy', 'Ichigo', 'Tanjiro', 'Asta', 'Gon')

The Heroes:

1 - Naruto

2 - Luffy

3 - Ichigo

4 - Tanjiro

5 - Asta

6 - Gon

The Fruits:

0 - Apple

1 - Mango

2 - Orange

3 - Guava

4 - Papaya

1

4

=== Code Execution Successful ===

**Set:**

#Creating a set

my\_set={101,102,103,104}

print(my\_set)

#No duplicates allowed in set

# Unordered: Sets do not maintain the order of how elements are stored in them.

# Unindexed: We cannot access the data elements of sets.

# No Duplicate Elements: Each data element in a set is unique.

# Mutable (Changeable): Sets in Python allow modification of their elements after creation.

#Types of creating a set

# 1- curly braces and 2-set()-function

#curly

#my\_set={101,102,103,104}

# Set method

list\_sample=[109,110,111]

set\_sample=set(list\_sample)

print(type(set\_sample))

# Since sets are unordered and unindexed, we cannot access the elements by position. However, we can iterate through a set with the help of loops.

print("for\_loop")

for num in set\_sample:

print(num)

# Adding Elements to the Set

# add(): This method is used to add a single element to the set.

# update(): This method is used to add multiple elements to the set.

set\_sample.add(112)

set\_sample.update([113,114])

print(set\_sample)

# Removing Elements from the Set

# remove(): This method allow us to remove a specific element from the set. It will raise a KeyError if the element is not found in the given set.

# discard(): This method is also used to remove a specified element from the set; however, it does not raise any error if the element is not found.

# pop(): This method is used to remove and returns a random element from the set.

# clear(): This method is used to remove all the elements from the given set.

new\_sample={90,91,92,93,94,95}

print(new\_sample)

new\_sample.remove(91)

print(new\_sample)

new\_sample.discard(96)

new\_sample.discard(97)

print(new\_sample)

new\_sample.pop()

print(new\_sample)

new\_sample.clear()

print(new\_sample)

# Set Operations

print("Set Operations")

a={1,2,3}

b={2,3,4,5}

print("Set A=",a)

print("Set B=",b)

print("Union")

#two methods

print("AUB=")

print(a|b)

print(a.union(b))

print("Intersection")

#two methods

print("AnB=")

print(a&b)

print(a.intersection(b))

print("Difference")

#two Ways

print(a-b)

print(a.difference(b))

print("Subset")

print(a.issubset(b))

print(b.issubset(a))

print("Disjoint")

print(a.isdisjoint(b))

print(b.isdisjoint(a))

Output:

{104, 101, 102, 103}

<class 'set'>

for\_loop

109

110

111

{109, 110, 111, 112, 113, 114}

{90, 91, 92, 93, 94, 95}

{90, 92, 93, 94, 95}

{90, 92, 93, 94, 95}

{92, 93, 94, 95}

set()

Set Operations

Set A= {1, 2, 3}

Set B= {2, 3, 4, 5}

Union

AUB=

{1, 2, 3, 4, 5}

{1, 2, 3, 4, 5}

Intersection

AnB=

{2, 3}

{2, 3}

Difference

{1}

{1}

Subset

False

False

Disjoint

False

False

=== Code Execution Successful ===

**Dictionaries:**

A **Python Dictionary** is one of the built-in data types used to store data in **'key: value'** pairs. The dictionary is an *unordered*, *mutable* and *indexed* collection where each key is unique and maps to a value. It is often used to store related data, like information associated with a particular entity or object, where we can easily get value on the basis of its key.

#Dictionaries

#Creating One

Dict={1:"Goku",2:"Luffy",3:"Naruto"}

Dict2=dict([(4,"Asta"),(5,"Tanjiro"),(6,"Gon")])

print(Dict)

print(Dict2)

# Mutable: Dictionaries can be modified after initialization allowing us to add, remove or update 'key: value' pairs.

# Unordered: Python dictionary does not follow a particular order to store items. However, starting from Python 3.7, the feature for the dictionary to maintain the insertion order of the items was added.

# Indexed: Unlike lists or tuples, which are indexed by position, dictionaries use keys to access values, offering faster and more readable data retrieval.

# Unique Keys: Each key in a dictionary must be unique. If we try to assign a value to an existing key, the old value will be replaced by the new one.

# Heterogeneous: Keys and values in a dictionary can be of any type.

#Accessing Dictionary Items

dict\_x={

"name":"Bharath",

"age":21,

"Job":"Cloud Developer",

"Gender":"Male"

}

print("personal details")

print(f"Name:{dict\_x["name"]}\nage:{dict\_x["age"]}\nJob:{dict\_x.get("Job")} \nGender:{dict\_x.get("Gender")}")

print("=============================")

dict\_x["country"]="India"

print("Updated personal details")

print(f"Name:{dict\_x["name"]}\nage:{dict\_x["age"]}\nJob:{dict\_x.get("Job")} \nGender:{dict\_x.get("Gender")} \nCountry:{dict\_x["country"]}")

# Removing Items from a Dictionary

# del: This keyword is used to remove an item by key.

# pop(): This method is used to remove an item by key. It also returns the value of the removed item.

# popitem(): This method removes and returns the last 'key: value' pair.

# clear(): This method is used to remove all items from the dictionary.

dict\_del={1:"Hi",2:"hello",3:"Hooo",4:"Yohoho",5:"hahahah"}

print(dict\_del)

del dict\_del[1]

print("After del:",dict\_del)

print(dict\_del.pop(2))

print("After pop:",dict\_del)

print(dict\_del.popitem())

print("After popitem:",dict\_del)

print(dict\_del.clear())

print("After clear:",dict\_del)

print("-=-=-=-=-=-=-=-=-=-=-=-=-=")

print("Iteration")

#Iterating

for key in Dict:

value=Dict.get(key)

print(f"{key}->{value}")

print("-=-=-=-=-=-=-=-=-=-=-=-=-=")

Output:

{1: 'Goku', 2: 'Luffy', 3: 'Naruto'}

{4: 'Asta', 5: 'Tanjiro', 6: 'Gon'}

personal details

Name:Bharath

age:21

Job:Cloud Developer

Gender:Male

=============================

Updated personal details

Name:Bharath

age:21

Job:Cloud Developer

Gender:Male

Country:India

{1: 'Hi', 2: 'hello', 3: 'Hooo', 4: 'Yohoho', 5: 'hahahah'}

After del: {2: 'hello', 3: 'Hooo', 4: 'Yohoho', 5: 'hahahah'}

hello

After pop: {3: 'Hooo', 4: 'Yohoho', 5: 'hahahah'}

(5, 'hahahah')

After popitem: {3: 'Hooo', 4: 'Yohoho'}

None

After clear: {}

-=-=-=-=-=-=-=-=-=-=-=-=-=

1->Goku

2->Luffy

3->Naruto

=== Code Execution Successful ===

**Function**

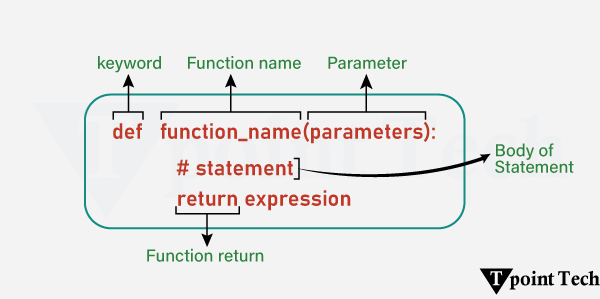
In Python, a **function** is a block of statements that performs a particular task. The main idea is to group tasks that we often do repeatedly into a single function. This way we can simply call the function whenever needed and reuse the code efficiently without writing the same code multiple times for different inputs.

Advantages of Using Functions in Python

The following are some of the key benefits of using Python Functions

* Functions help in the reusability of code.
* It also reduces the code length.
* It improves the readability of code.

Declaring a Function in python:



Types of Functions in Python

The following are the types of functions used in [Python](https://www.tpointtech.com/python-tutorial):

* [**Built-in Functions**](https://www.tpointtech.com/python-built-in-functions): Python standard functions that can be used anytime.
* **User-defined Functions:** Functions created by the users on the basis of their requirements.

Types of Python Function Arguments

There are several types of function arguments:

* [Positional Arguments](https://www.tpointtech.com/python-positional-only-arguments)
* [Keyword Arguments](https://www.tpointtech.com/keywords-and-positional-arguments-in-python)
* [Default Arguments](https://www.tpointtech.com/default-arguments-in-python)
* [Arbitrary Arguments (Variable-length arguments \*args and \*\*kwargs)](https://www.tpointtech.com/variable-length-arguments-in-python)

#Creating a function

def Hello():

print("Welcome to Kanniyakumari")

Hello()

#Passing Arguements

print("Passing arguements")

def hello(name):

print(f"Welcome to {name}")

hello("Kanyakumari")

#Return

#In Python, the return statement is used in order to return a value from the function.

print("Square of 13")

def squareofnum(num):

res=num\*\*2

return res

print(squareofnum(13))

print("=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=")

Output:

Welcome to Kanniyakumari

Passing arguements

Welcome to Kanyakumari

Square of 13

169

=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

Python Built-in Functions

#Python Buil-in functions

# abs

# The Python abs() function is used to return the absolute value of a number. It takes only one argument, a number whose absolute value is to be returned. The argument can be an integer or a floating-point number. If the argument is a complex number, then abs() returns its magnitude.

print("Absolute values\n",abs(-1),abs(2))

k = [1, 3, 4, 6]

print(all(k))

#The Python all() function accepts an iterable object (such as a list, dictionary, etc.). It returns true if all items in the passed iterable are true. Otherwise, it returns False. If the iterable object is empty, the all() function returns True.

# all values false

print("The All Functions:")

k = [0, False]

print(all(k))

# one false value

k = [1, 3, 7, 0]

print(all(k))

# one true value

k = [0, False, 5]

print(all(k))

# empty iterable

k = []

print(all(k))

#The Python bin() function is used to return the binary representation of a specified integer. A result always starts with the prefix 0b.

print("-=======------------==============")

print("Bin Function")

x=10

y=bin(x)

print("Binary value of 10 is:",y)

print("-=======------------==============")

print("Compile and exec")

code\_str = 'x=5\ny=10\nprint("sum =",x+y)'

code = compile(code\_str, 'sum.py', 'exec')

print(type(code))

exec(code)

#exec(x)

print("-=======------------==============")

# Python sum() Function

# As the name says, the Python sum() function is used to get the sum of numbers in an iterable, i.e., a list.

S=[1,2,3,4,5]

print("Sum of S:",sum(S))

print("-=======------------==============")

Output:

Absolute values

1 2

True

The All Functions:

False

False

False

True

-=======------------==============

Bin Function

Binary value of 10 is: 0b1010

-=======------------==============

Compile and exec

<class 'code'>

sum = 15

-=======------------==============

Sum of S: 15

=== Code Execution Successful ===

Python Lambda Functions:

In Python, Lambda Functions are concise, anonymous functions, which means they are functions without a name. They are used for streamlining and simplifying short tasks. It contributes in improving the readability of the program.

print("Addition")

add=lambda X:X+10

print(add(5))

print("Product")

prod=lambda X:X\*2

print(prod(4))

print("--------------------------------------------")

print("Odd or Even")

oddoreven=lambda X:"Zero" if X==0 else "Even" if X%2==0 else "Odd"

print(oddoreven(10))

print(oddoreven(0))

print(oddoreven(5))

print("--------------------------------------------")

print("List Comprehension")

list1=[lambda a=x:a\*10 for x in range(1,6)]

for i in list1:

print(i())

print("--------------------------------------------")

print("Lambda with Multiple Statements")

ans=lambda x,y:(x+y,x\*y,x-y,x/y,x%y)

print(ans(10,5))

print("--------------------------------------------")

print("Filtering")

num=[1,2,3,4,5,6]

even\_list=filter(lambda a:a%2==0,num)

print(list(even\_list))

print("--------------------------------------------")

print("Mapping")

num=[1,2,3,4,5]

mapping=map(lambda a:a\*\*2,num)

print(list(mapping))

print("--------------------------------------------")

print("Reducing")

from functools import reduce

year\_list=[2,0,2,5]

year=reduce(lambda x,y:x\*10+y,year\_list)

print(year)

print("--------------------------------------------")

Output:

Addition

15

Product

8

--------------------------------------------

Odd or Even

Even

Zero

Odd

--------------------------------------------

List Comprehension

10

20

30

40

50

--------------------------------------------

Lambda with Multiple Statements

(15, 50, 5, 2.0, 0)

--------------------------------------------

Filtering

[2, 4, 6]

--------------------------------------------

Mapping

[1, 4, 9, 16, 25]

--------------------------------------------

Reducing

2025

=== Code Execution Successful ===

Def function:

print("DEF Function")

def greet():

"""This function greets the user."""

print("Hello, welcome to Python programming!")

# Calling the function

greet()

print("--------------------------------------------")

print("Positional Arguements")

def greet(name):

"""This function greets the user by name."""

print(f"Hello, {name}! Welcome to Python programming.")

# Calling the function with an argument

greet("Alice")

print("--------------------------------------------")

print("Default Arguements")

def greet(name="Guest"):

"""This function greets the user by name, defaulting to 'Guest'."""

print(f"Hello, {name}! Welcome to Python programming.")

# Calling the function without an argument

greet()

print("--------------------------------------------")

print("Keyword Arguements")

def greet(name, message):

"""This function greets the user with a custom message."""

print(f"Hello, {name}! {message}")

# Calling the function with keyword arguments

greet(message="How are you today?", name="Alice")

print("--------------------------------------------")

print("Arbitrary Arguements")

def greet(\*names):

"""This function greets multiple users."""

for name in names:

print(f"Hello, {name}! Welcome to Python programming.")

# Calling the function with multiple arguments

greet("Alice", "Bob", "Charlie")

print("--------------------------------------------")

print("return values")

def add(a,b):

return a+b

print(add(10,5))

print("--------------------------------------------")

print("Returning Multiple values:")

def calculation(a,b):

return a+b,a-b

print(calculation(10,5))

print("--------------------------------------------")

print("Global and local Scope")

x=10

def local():

x=5

print("local:",x)

print("Global:",x)

local()

print("--------------------------------------------")

Output

DEF Function

Hello, welcome to Python programming!

--------------------------------------------

Positional Arguements

Hello, Alice! Welcome to Python programming.

--------------------------------------------

Default Arguements

Hello, Guest! Welcome to Python programming.

--------------------------------------------

Keyword Arguements

Hello, Alice! How are you today?

--------------------------------------------

Arbitrary Arguements

Hello, Alice! Welcome to Python programming.

Hello, Bob! Welcome to Python programming.

Hello, Charlie! Welcome to Python programming.

--------------------------------------------

return values

15

--------------------------------------------

Returning Multiple values:

(15, 5)

--------------------------------------------

Global and local Scope

Global: 10

local: 5

--------------------------------------------

=== Code Execution Successful ===

Class:

class Student:

def \_\_init\_\_(self,name,class\_,Roll\_no):

self.name=name

self.class\_=class\_

self.Roll\_no=Roll\_no

def greet(self):

print(f"Name:{self.name}\nclass\_:{self.class\_}\nRoll\_no:{self.Roll\_no}")

Student1=Student("Bharath","B.Tech",64)

Student1.greet()

Output:

Name:Bharath

class\_:B.Tech

Roll\_no:64

=== Code Execution Successful ===

Python Modules:

In Python, a module is referred to as a file comprising of functions, classes, or variables. It helps us organizing the code logically and reuse it across different programs. There are many Python modules, each with its specific work.

A Python module is a file consisting of Python script (generally, with a .py extension) having definitions of functions, classes, or variables. A module can also include runnable code.

Putting related code into a module makes it easier to read, use again in other programs, and keep everything organized.

There are three types of modules in Python:

1. User-defined Modules
2. Built-in Modules
3. Third-Party Modules

# module: calculator.py

*# function to add two numbers*

**def** add(a, b):

**return** a + b

*# function to subtract two numbers*

**def** subtract(a, b):

**return** a - b

*# function to multiply two numbers*

**def** multiply(a, b):

**return** a \* b

*# function to divide two numbers*

**def** divide(a, b):

**return** a / b

1. # importing the module
2. **import** calculator
4. *# initializing some variables*
5. num\_1 = 16
6. num\_2 = 7
8. *# calling the functions of the module*
9. total = calculator.add(num\_1, num\_2)
10. diff = calculator.subtract(num\_1, num\_2)
11. prod = calculator.multiply(num\_1, num\_2)
12. quot = calculator.divide(num\_1, num\_2)
14. *# printing results*
15. **print**(num\_1, '+', num\_2, '=', total)
16. **print**(num\_1, '-', num\_2, '=', diff)
17. **print**(num\_1, '\*', num\_2, '=', prod)
18. **print**(num\_1, '/', num\_2, '=', quot)

Python from…import Statement

Python allows us to import specific attributes without importing the module as a whole. To understand this, let us see the following example:

Example

1. *# importing certain attributes from the module*
2. **from** calculator **import** subtract, multiply
4. *# initializing some variables*
5. num\_1 = 12
6. num\_2 = 7
8. *# calling the imported functions*
9. diff = subtract(num\_1, num\_2)
10. prod = multiply(num\_1, num\_2)
12. *# printing results*
13. **print**(num\_1, '-', num\_2, '=', diff)
14. **print**(num\_1, '\*', num\_2, '=', prod)

Importing All Attributes

Similar to importing a certain attributes from the module, we can use the asterisk \* symbol to import everything from the module.

**Syntax:**

1. **from** module\_name **import** \*

Here is an example showing the same:

Example

1. *# importing everything from the module*
2. **from** calculator **import** \*
4. *# initializing some variables*
5. num\_1 = 19
6. num\_2 = 14
8. *# calling the imported functions*
9. prod = multiply(num\_1, num\_2)
10. quot = divide(num\_1, num\_2)
12. *# printing results*
13. **print**(num\_1, '\*', num\_2, '=', prod)
14. **print**(num\_1, '/', num\_2, '=', quot)

Exception Handling in Python

What is an Exception?

An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.

import tkinter as tk

from tkinter import messagebox

# Create the main window

window = tk.Tk()

# Set the title of the window

window.title("Tkinter GUI Demo")

# Set the default size of the window (width x height)

window.geometry("400x400")

# ------------------- LABELS -------------------

# Labels are used to display static text

label1 = tk.Label(window, text="Enter your name:")

label1.pack()

label2 = tk.Label(window, text="Enter your age:")

label2.pack()

label3 = tk.Label(window, text="Click the button below:")

label3.pack()

# ------------------- TEXTBOXES (Entry widgets) -------------------

# Entry widgets are used to take single-line input from the user

name\_entry = tk.Entry(window)

name\_entry.pack()

age\_entry = tk.Entry(window)

age\_entry.pack()

# ------------------- BUTTONS -------------------

# Buttons perform actions when clicked

def on\_submit():

    name = name\_entry.get()

    age = age\_entry.get()

    messagebox.showinfo("Info", f"Hello {name}, you are {age} years old!")

submit\_button = tk.Button(window, text="Submit", command=on\_submit)

submit\_button.pack()

# ------------------- EVENT HANDLING -------------------

# This function handles button click events

def on\_click\_event():

    messagebox.showinfo("Event", "Button was clicked!")

event\_button = tk.Button(window, text="Click Me", command=on\_click\_event)

event\_button.pack()

# ------------------- ADDITIONAL WIDGETS -------------------

# Checkbutton allows selection of options (like a checkbox)

subscribe\_var = tk.IntVar()

subscribe\_check = tk.Checkbutton(window, text="Subscribe to newsletter", variable=subscribe\_var)

subscribe\_check.pack()

# Radiobuttons allow selection of one option from a group

gender\_var = tk.StringVar(value="None")

tk.Label(window, text="Select Gender:").pack()

tk.Radiobutton(window, text="Male", variable=gender\_var, value="Male").pack()

tk.Radiobutton(window, text="Female", variable=gender\_var, value="Female").pack()

# Listbox allows selection from a list of items

tk.Label(window, text="Select your country:").pack()

country\_listbox = tk.Listbox(window)

countries = ["India", "USA", "UK", "Germany", "Australia"]

for country in countries:

    country\_listbox.insert(tk.END, country)

country\_listbox.pack()

# ------------------- MAIN LOOP -------------------

# Starts the GUI event loop

window.mainloop()

Python Decorator

Decorators are one of the most helpful and powerful tools of Python. These are used to modify the behavior of the function. Decorators provide the flexibility to wrap another function to expand the working of wrapped function, without permanently modifying it.

*In Decorators, functions are passed as an argument into another function and then called inside the wrapper function.*

It is also called **meta programming** where a part of the program attempts to change another part of program at compile time.

Before understanding the **Decorator**, we need to know some important concepts of Python.

What are the functions in Python?

Python has the most interesting feature that everything is treated as an object even classes or any variable we define in Python is also assumed as an object. Functions are **first-class** objects in the Python because they can reference to, passed to a variable and returned from other functions as well. The example is given below:

**Example:**

1. def func1(msg):    # here, we are creating a function and passing the parameter
2. print(msg)
3. func1("Hii, welcome to function ")   # Here, we are printing the data of function 1
4. func2 = func1      # Here, we are copying the function 1 data to function 2
5. func2("Hii, welcome to function ")   # Here, we are printing the data of function 2

**Output:**

*Hii, welcome to function*

*Hii, welcome to function*

In the above program, when we run the code it give the same output for both functions. The **func2**referred to function **func1** and act as function. We need to understand the following concept of the function:

* The function can be referenced and passed to a variable and returned from other functions as well.
* The functions can be declared inside another function and passed as an argument to another function.

Inner Function

Python provides the facility to define the function inside another function. These types of functions are called inner functions. Consider the following example:

**Example:**

1. def func():    # here, we are creating a function and passing the parameter
2. print("We are in first function")      # Here, we are printing the data of function
3. def func1():      # here, we are creating a function and passing the parameter
4. print("This is first child function")  # Here, we are printing the data of function 1
5. def func2():      # here, we are creating a function and passing the parameter
6. print("This is second child function")      # Here, we are printing the data of         # function 2
7. func1()
8. func2()
9. func()

**Output:**

*We are in first function*

*This is first child function*

*This is second child function*

In the above program, it doesn't matter how the child functions are declared. The execution of the child function makes effect on the output. These child functions are locally bounded with the **func()** so they cannot be called separately.

A function that accepts other function as an argument is also called **higher order function**. Consider the following example:

**Example:**

1. def add(x):          # here, we are creating a function add and passing the parameter
2. **return** x+1       # here, we are returning the passed value by adding 1
3. def sub(x):          # here, we are creating a function sub and passing the parameter
4. **return** x-1        # here, we are returning the passed value by subtracting 1
5. def operator(func, x):    # here, we are creating a function and passing the parameter
6. temp = func(x)
7. **return** temp
8. print(operator(sub,10))  # here, we are printing the operation subtraction with 10
9. print(operator(add,20))   # here, we are printing the operation addition with 20

**Output:**

*9*

*21*

In the above program, we have passed the **sub()** function and **add()** function as argument in **operator()** function.

A function can return another function. Consider the below example:

**Example:**

1. def hello():         # here, we are creating a function named hello
2. def hi():         # here, we are creating a function named hi
3. print("Hello")             # here, we are printing the output of the function
4. **return** hi         # here, we are returning the output of the function
5. **new** = hello()
6. **new**()

**Output:**

*Hello*

In the above program, the **hi()** function is nested inside the **hello()** function. It will return each time we call **hi()**.

Decorating functions with parameters

Let's have an example to understand the parameterized decorator function:

**Example:**

1. def divide(x,y):       # here, we are creating a function and passing the parameter
2. print(x/y)         # Here, we are printing the result of the expression
3. def outer\_div(func):      # here, we are creating a function and passing the parameter
4. def inner(x,y):      # here, we are creating a function and passing the parameter
5. **if**(x<y):
6. x,y = y,x
7. **return** func(x,y)
8. # here, we are returning a function with some passed parameters
9. **return** inner
10. divide1 = outer\_div(divide)
11. divide1(2,4)

**Output:**

*2.0*

Syntactic Decorator

In the above program, we have decorated **out\_div()** that is little bit bulky. Instead of using above method, Python allows to **use decorator in easy way with @symbol**. Sometimes it is called "pie" syntax.

1. def outer\_div(func):     # here, we are creating a function and passing the parameter
2. def inner(x,y):        # here, we are creating a function and passing the parameter
3. **if**(x<y):
4. x,y = y,x
5. **return** func(x,y)       # here, we are returning the function with the parameters
6. **return** inner
7. # Here, the below is the syntax of generator
8. @outer\_div
9. def divide(x,y):      # here, we are creating a function and passing the parameter
10. print(x/y)

**Output:**

*2.0*

Reusing Decorator

We can reuse the decorator as well by recalling that decorator function. Let's make the decorator to its own module that can be used in many other functions. Creating a file called **mod\_decorator.py** with the following code:

1. def do\_twice(func):      # here, we are creating a function and passing the parameter
2. def wrapper\_do\_twice():
3. # here, we are creating a function and passing the parameter
4. func()
5. func()
6. **return** wrapper\_do\_twice
7. We can **import** mod\_decorator.py in another file.
8. from decorator **import** do\_twice
9. @do\_twice
10. def say\_hello():
11. print("Hello There")
12. say\_hello()

We can import mod\_decorator.py in other file.

1. from decorator **import** do\_twice
2. @do\_twice
3. def say\_hello():
4. print("Hello There")
5. say\_hello()

**Output:**

*Hello There*

*Hello There*

Python Decorator with Argument

We want to pass some arguments in function. Let's do it in following code:

1. from decorator **import** do\_twice
2. @do\_twice
3. def display(name):
4. print(f"Hello {name}")
5. display()

**Output:**

*TypeError: display() missing 1 required positional argument: 'name'*

As we can see that, the function didn't accept the argument. Running this code raises an error. We can fix this error by using **\*args** and **\*\*kwargs**in the inner wrapper function. Modifying the **decorator.py**as follows:

1. def do\_twice(func):
2. def wrapper\_function(\*args,\*\*kwargs):
3. func(\*args,\*\*kwargs)
4. func(\*args,\*\*kwargs)
5. **return** wrapper\_function

Now **wrapper\_function()** can accept any number of argument and pass them on the function.

1. from decorator **import** do\_twice
2. @do\_twice
3. def display(name):
4. print(f"Hello {name}")
5. display("John")

**Output:**

*Hello John*

*Hello John*

Returning Values from Decorated Functions

We can control the return type of the decorated function. The example is given below:

1. from decorator **import** do\_twice
2. @do\_twice
3. def return\_greeting(name):
4. print("We are created greeting")
5. **return** f"Hi {name}"
6. hi\_adam = return\_greeting("Adam")

**Output:**

*We are created greeting*

*We are created greeting*

*Python Generators*

*28 Aug 2025 |  4 min read*

***Python generator****is a special type of function that returns an iterator object. Instead of returning a value, it uses yield to give a sequence of values over time, which pauses after each yield statement while keeping its state between iterations.*

*Generators are memory efficient, which means making items one at a time, with the help of****yield****keyword.*

*Let us see a simple example of Generator in*[*Python*](https://www.tpointtech.com/python-tutorial)*:*

*Example*

1. ***def****generate\_numbers(limit):*
2. ***for****num****in****range(1, limit + 1):*
3. ***yield****num*
4. *sequence = generate\_numbers(5)*
5. ***for****value****in****sequence:*
6. ***print****(value)*

***Output:***

*1*

*2*

*3*

*4*

*5*

***Explanation:***

*In the above code, a generate\_numbers function is defined, which accepts a limit as argument, that executes a for loop in the function range from 1 to limit + 1, and produces the number. The function is called with a limit value of 5 and prints the yielded value.*

*How to Define a Generator in Python?*

*It is easy to create a generator, simply define a function that has at least one*[***yield***](https://www.tpointtech.com/yield-keyword-in-python)*statement in the function. When the function is called, instead of returning a single value, you will get back a generator object, which is an iterator. Here is the standard syntax to define the generator in Python.*

***Syntax:***

1. ***def****function(parameters):*
2. ***yield****statement*

*Example*

1. ***def****number\_generator():*
2. *numbers = [1, 2, 3]*
3. ***for****num****in****numbers:*
4. ***yield****num*
5. *# Running the generator to display its output*
6. ***for****item****in****number\_generator():*
7. ***print****(item)*

***Output:***

*1*

*2*

*3*

***Explanation:***

*The above code in the example defines a generator function 'number\_generator' which generates the values from a list [1, 2, 3] one at a time with the help of a*[*for*](https://www.tpointtech.com/python-for-loop)*loop. Each call to the generator function provides the next value in sequence and can be more memory-efficient, which involves iterating over the data in a lazy manner.*