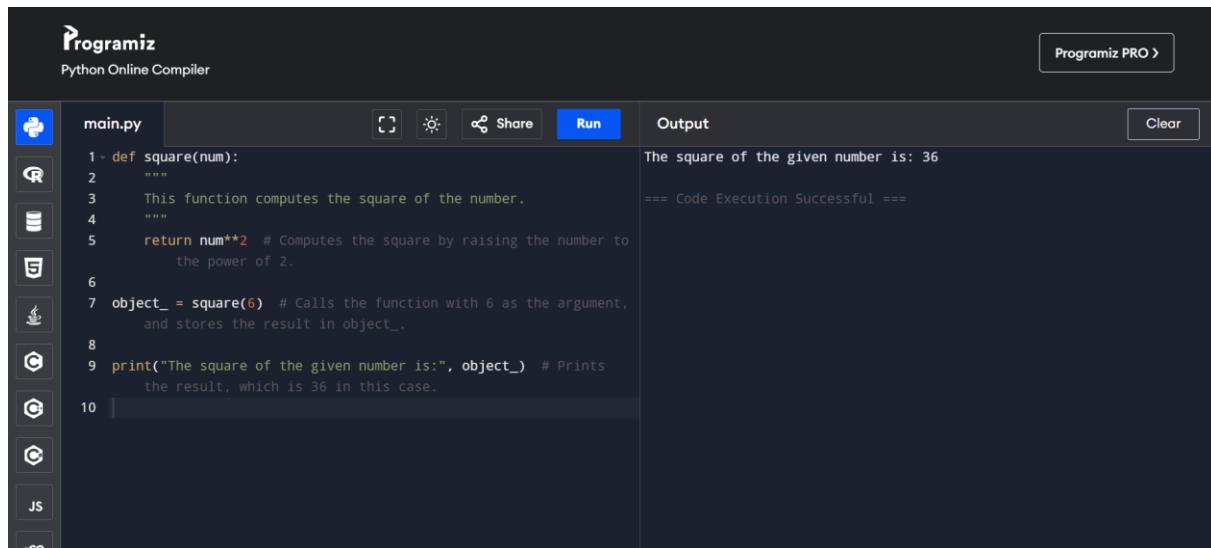


PYTHON PROJECT

Python Functions

A collection of related assertions that carry out a mathematical, analytical, or evaluative operation is known as a function. An assortment of proclamations called Python Capabilities returns the specific errand. Python functions are necessary for intermediate-level programming and are easy to define. Function names meet the same standards as variable names do. The objective is to define a function and group-specific frequently performed actions. Instead of repeatedly creating the same code block for various input variables, we can call the function and reuse the code it contains with different variables.

Illustration of a User-Defined Function



The screenshot shows the Programiz Python Online Compiler interface. On the left, there's a sidebar with icons for Python, C, C++, Java, JavaScript, and Go. The main area has tabs for 'main.py' (selected), 'Run', 'Share', and 'Output'. The code editor contains the following Python script:

```
1 def square(num):
2     """
3         This function computes the square of the number.
4     """
5     return num**2 # Computes the square by raising the number to
                     # the power of 2.
6
7 object_ = square(6) # Calls the function with 6 as the argument,
                     # and stores the result in object_.
8
9 print("The square of the given number is:", object_) # Prints
                     # the result, which is 36 in this case.
10
```

The output window shows the results of running the code:

```
The square of the given number is: 36
== Code Execution Successful ==
```

Calling a Function

The screenshot shows the Programiz Python Online Compiler interface. On the left, there's a sidebar with icons for Python, C, C++, Java, JavaScript, and Go. The main area has a file named 'main.py' containing the following code:

```

1 # Defining a function
2 def a_function(string):
3     "This prints the value of length of string" # This is a
        docstring, but it doesn't do anything in the code.
5
6 # Calling the function we defined
7 print("Length of the string 'Functions' is:", a_function
        ("Functions")) # Calls the function with "Functions"
8 print("Length of the string 'Python' is:", a_function("Python"))
        # Calls the function with "Python"
9

```

At the top, there are buttons for Run, Share, and a Run button. To the right, there's an advertisement for Lufthansa flights from San Francisco to India for ₹94,828*. The output window shows the results of running the code:

```

Length of the string 'Functions' is: 9
Length of the string 'Python' is: 6
== Code Execution Successful ==

```

Pass by Reference vs. Pass by Value

The screenshot shows the Programiz Python Online Compiler interface. On the left, there's a sidebar with icons for Python, C, C++, Java, JavaScript, and Go. The main area has a file named 'main.py' containing the following code:

```

1 # defining the function
2 def square(item_list):
3     '''This function will find the square of items in the list'''
4     squares = [] # Create an empty list to store the squares.
5     for l in item_list: # Loop through each element in the input
        list.
6         squares.append(l**2) # Square the element and append it
            to the 'squares' list.
7     return squares # Return the list of squared elements.
8
9 # calling the defined function
10 my_list = [17, 52, 8] # Initial list.
11 my_result = square(my_list) # Pass the list to the function and
        store the result in 'my_result'.
12
13 print("Squares of the list are:", my_result) # Print the
        resulting list of squares.
14

```

At the top, there are buttons for Run, Share, and a Run button. To the right, there's an advertisement for Skyscanner flights from India to Dubai for Rs14,263 and to Thailand for Rs11,153. The output window shows the results of running the code:

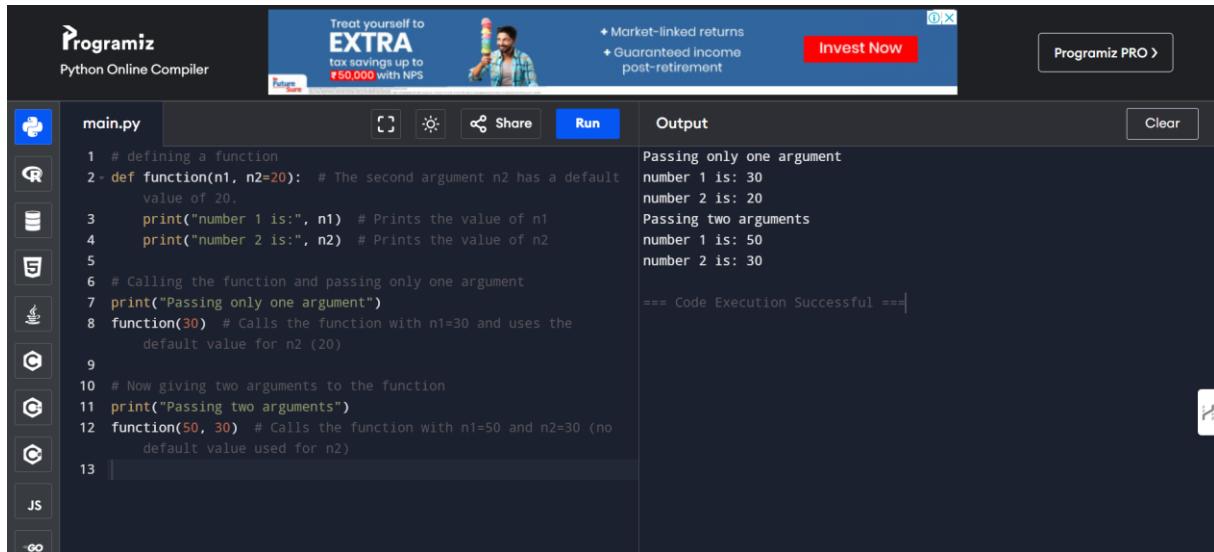
```

Squares of the list are: [289, 2704, 64]
== Code Execution Successful ==

```

Function Arguments

1) Default Arguments



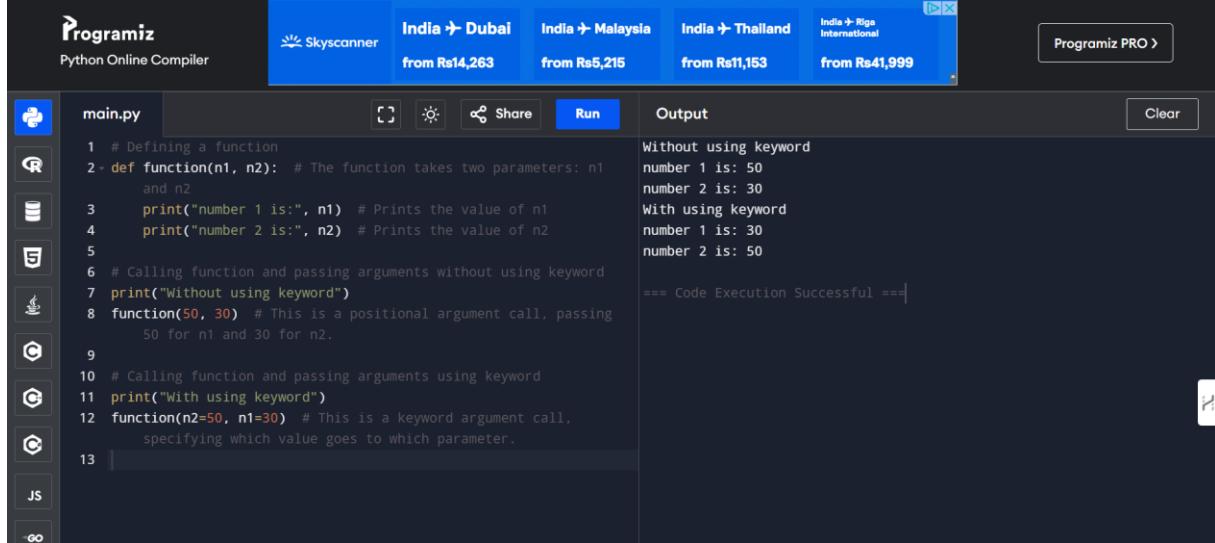
The screenshot shows the Programiz Python Online Compiler interface. On the left, there's a sidebar with icons for Python, C, C++, Java, JavaScript, and Go. The main area has tabs for 'main.py' and 'Output'. The code editor contains the following Python script:

```
1 # defining a function
2 def function(n1, n2=20): # The second argument n2 has a default
   value of 20.
3   print("number 1 is:", n1) # Prints the value of n1
4   print("number 2 is:", n2) # Prints the value of n2
5
6 # Calling the function and passing only one argument
7 print("Passing only one argument")
8 function(30) # Calls the function with n1=30 and uses the
   default value for n2 (20)
9
10 # Now giving two arguments to the function
11 print("Passing two arguments")
12 function(50, 30) # Calls the function with n1=50 and n2=30 (no
   default value used for n2)
13
```

The output window shows the results of running the code:

```
Passing only one argument
number 1 is: 30
number 2 is: 20
Passing two arguments
number 1 is: 50
number 2 is: 30
== Code Execution Successful ==
```

2) Keyword Arguments



The screenshot shows the Programiz Python Online Compiler interface. On the left, there's a sidebar with icons for Python, C, C++, Java, JavaScript, and Go. The main area has tabs for 'main.py' and 'Output'. The code editor contains the following Python script:

```
1 # Defining a function
2 def function(n1, n2): # The function takes two parameters: n1
   and n2
3   print("number 1 is:", n1) # Prints the value of n1
4   print("number 2 is:", n2) # Prints the value of n2
5
6 # Calling function and passing arguments without using keyword
7 print("Without using keyword")
8 function(50, 30) # This is a positional argument call, passing
   50 for n1 and 30 for n2.
9
10 # Calling function and passing arguments using keyword
11 print("With using keyword")
12 function(n2=50, n1=30) # This is a keyword argument call,
   specifying which value goes to which parameter.
13
```

The output window shows the results of running the code:

```
Without using keyword
number 1 is: 50
number 2 is: 30
With using keyword
number 1 is: 30
number 2 is: 50
== Code Execution Successful ==
```

3) Required Arguments

The screenshot shows a Python code editor interface. On the left, there's a sidebar with icons for various programming languages: Python, C, C++, Java, JavaScript, Go, PHP, and Swift. The main area has a file named 'main.py' with the following code:

```

1 # Defining a function
2 def function(n1, n2): # Function accepts two arguments: n1 and n2
3     print("number 1 is:", n1) # Prints the value of n1
4     print("number 2 is:", n2) # Prints the value of n2
5
6 # Calling function and passing two arguments out of order, we need num1
# to be 20 and num2 to be 30
7 print("Passing out of order arguments")
8 function(30, 20) # Positional arguments: n1 gets 30, n2 gets 20
9
10 # Calling function and passing only one argument
11 print("Passing only one argument")
12 try:
13     function(30) # Only passing one argument; this will raise an error
# because function needs two arguments
14 except:
15     print("Function needs two positional arguments") # Handle the error
# when the second argument is missing
16

```

On the right, there are buttons for 'Run' and 'Output'. The 'Output' section displays the results of running the code:

Passing out of order arguments
number 1 is: 30
number 2 is: 20
Passing only one argument
Function needs two positional arguments
== Code Execution Successful ==

4) Variable-Length Arguments

The screenshot shows a Python code editor interface. On the left, there's a sidebar with icons for various programming languages: Python, C, C++, Java, JavaScript, Go, PHP, and Swift. The main area has a file named 'main.py' with the following code:

```

1 # Defining a function using *args (variable-length positional arguments)
2 def function(*args_list):
3     ans = [] # List to store uppercase words
4     for l in args_list: # Loop through each element in args_list
5         ans.append(l.upper()) # Convert each element to uppercase and
# append to ans
6     return ans # Return the list of uppercase words
7
8 # Passing *args arguments
9 object = function('Python', 'Functions', 'Tutorial') # Pass three words
10 print(object) # Output will be ['PYTHON', 'FUNCTIONS', 'TUTORIAL']
11
12 # Defining a function using **kwargs (variable-length keyword arguments)
13 def function(**kargs_list):
14     ans = [] # List to store key-value pairs
15     for key, value in kargs_list.items(): # Loop through each key-value
# pair
16         ans.append([key, value]) # Append the key-value pair to ans as
# a list
17     return ans # Return the list of key-value pairs
18
19 # Passing **kwargs arguments
20 object = function(First="Python", Second="Functions", Third="Tutorial")
# Pass keyword arguments
21 print(object) # Output will be [['First', 'Python'], ['Second',
# 'Functions'], ['Third', 'Tutorial']]
22

```

On the right, there are buttons for 'Run' and 'Output'. The 'Output' section displays the results of running the code:

['PYTHON', 'FUNCTIONS', 'TUTORIAL']
[['First', 'Python'], ['Second', 'Functions'], ['Third', 'Tutorial']]
== Code Execution Successful ==

return Statement

The screenshot shows a Python code editor interface. On the left, there's a sidebar with icons for various programming languages: Python, C, C++, Java, JavaScript, Go, PHP, and Swift. The main area has a file named 'main.py' with the following code:

```

1 # Defining a function with return statement
2 def square(num):
3     return num**2 # Returns the square of the number
4
5 # Calling function and passing arguments.
6 print("With return statement")
7 print(square(52)) # This will print the square of 52
8
9 # Defining a function without return statement
10 def square(num):
11     num**2 # This will calculate the square but doesn't return it
12
13 # Calling function and passing arguments.
14 print("Without return statement")
15 print(square(52)) # This will print 'None' because the function doesn't
# return a value
16

```

On the right, there are buttons for 'Run' and 'Output'. The 'Output' section displays the results of running the code:

With return statement
2704
Without return statement
None
== Code Execution Successful ==

The Anonymous Functions

The screenshot shows a code editor interface with a dark theme. On the left, there is a sidebar with icons for various programming languages: Python (selected), C, C++, C#, JavaScript, Go, PHP, and Swift. The main area has tabs for "main.py" and "Output". The code in "main.py" is:

```
1 # Defining a lambda function
2 lambda_ = lambda argument1, argument2: argument1 + argument2
3
4 # Calling the lambda function and passing values
5 print("Value of the function is: ", lambda_(20, 30)) # The sum of 20 and
       30
6 print("Value of the function is: ", lambda_(40, 50)) # The sum of 40 and
       50
7
```

The "Output" tab shows the results of running the code:

```
Value of the function is: 50
Value of the function is: 90
== Code Execution Successful ==
```

Scope and Lifetime of Variables

The screenshot shows a code editor interface with a dark theme. On the left, there is a sidebar with icons for various programming languages: Python (selected), C, C++, C#, JavaScript, Go, PHP, and Swift. The main area has tabs for "main.py" and "Output". The code in "main.py" is:

```
1 # Defining a function to print a number
2 def number():
3     num = 50 # local variable
4     print("Value of num inside the function:", num)
5
6 # Global variable
7 num = 10
8
9 # Calling the function
10 number()
11
12 # Accessing the global variable outside the function
13 print("Value of num outside the function:", num)
14
```

The "Output" tab shows the results of running the code:

```
Value of num inside the function: 50
Value of num outside the function: 10
== Code Execution Successful ==
```

Python Capability inside Another Capability

The screenshot shows a code editor interface with a dark theme. On the left, there is a sidebar with icons for various languages: Python (selected), C, C++, C#, JavaScript, Go, PHP, and Swift. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Defining a nested function
2 def word():
3     string = 'Python functions tutorial' # Outer variable
4     x = 5 # Outer variable
5
6     # Defining an inner function
7     def number():
8         print(string) # Accessing the outer variable 'string'
9         print(x) # Accessing the outer variable 'x'
10
11    # Calling the inner function
12    number()
13
14 # Calling the outer function
15 word()
16
```

The 'Output' tab shows the results of running the code:

```
Python functions tutorial
5
== Code Execution Successful ==
```

Python Built-in Functions

Python abs() Function Example

The screenshot shows a code editor interface with a dark theme. On the left, there is a sidebar with icons for various languages: Python (selected), C, C++, C#, JavaScript, Go, PHP, and Swift. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Integer number
2 integer = -20
3 print('Absolute value of -20 is:', abs(integer))
4
5 # Floating number
6 floating = -20.83
7 print('Absolute value of -20.83 is:', abs(floating))
8
```

The 'Output' tab shows the results of running the code:

```
Absolute value of -20 is: 20
Absolute value of -20.83 is: 20.83
== Code Execution Successful ==
```

Python all() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there's a sidebar with icons for various languages: Python (selected), QR code, JSON, CSS, JS, Go, PHP, and Swift. The main area has a title bar with 'main.py' and tabs for Run, Share, and Output. The code in the editor is:

```
1 # All values true
2 k = [1, 3, 4, 6]
3 print(all(k)) # All values are truthy, so it returns True
4
5 # All values false
6 k = [0, False]
7 print(all(k)) # All values are falsy, so it returns False
8
9 # One false value
10 k = [1, 3, 7, 0]
11 print(all(k)) # Contains a falsy value (0), so it returns False
12
13 # One true value
14 k = [0, False, 5]
15 print(all(k)) # Contains a truthy value (5), so it returns False
16
17 # Empty iterable
18 k = []
19 print(all(k)) # Empty iterable, so it returns True
20
```

The output panel shows the results of each print statement and a success message:

```
True
False
False
False
True
===
Code Execution Successful ===
```

Python bin() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there's a sidebar with icons for various languages: Python (selected), QR code, JSON, CSS, JS, Go, PHP, and Swift. The main area has a title bar with 'main.py' and tabs for Run, Share, and Output. The code in the editor is:

```
1 x = 10
2 y = bin(x)
3 print(y)
4
```

The output panel shows the result of the print statement and a success message:

```
0b1010
===
Code Execution Successful ===
```

Python bool()

The screenshot shows a Python code editor interface with a dark theme. On the left, there's a sidebar with icons for various languages: Python (selected), QR code, JSON, CSS, JS, Go, PHP, and Swift. The main area has a title bar with 'main.py' and tabs for Run, Share, and Output. The code in the editor is:

```
1 test1 = []
2 print(test1, 'is', bool(test1)) # Empty list is falsy
3
4 test1 = [0]
5 print(test1, 'is', bool(test1)) # Non-empty list is truthy, even if it
       contains 0
6
7 test1 = 0.0
8 print(test1, 'is', bool(test1)) # 0.0 is falsy
9
10 test1 = None
11 print(test1, 'is', bool(test1)) # None is falsy
12
13 test1 = True
14 print(test1, 'is', bool(test1)) # True is truthy
15
16 test1 = 'Easy string'
17 print(test1, 'is', bool(test1)) # Non-empty string is truthy
18
```

The output panel shows the results of each print statement and a success message:

```
[] is False
[0] is True
0.0 is False
None is False
True is True
Easy string is True
===
Code Execution Successful ===
```

Python bytes()

main.py

```
1 string = "Hello World."
2 array = bytes(string, 'utf-8')
3 print(array)
4
```

Run Output Clear

```
b'Hello World.'
== Code Execution Successful ==
```

Python callable() Function

main.py

```
1 x = 8
2 print(callable(x))
3
```

Run Output Clear

```
False
== Code Execution Successful ==
```

Python compile() Function

main.py

```
1 # compile string source to code
2 code_str = 'x=5\ny=10\nprint("sum =",x+y)'
3
4 # compile the string into a code object
5 code = compile(code_str, 'sum.py', 'exec')
6
7 # Print the type of the code object
8 print(type(code)) # This will print <class 'code'>, as it's a code object
9
10 # Execute the compiled code
11 exec(code) # This will print: sum = 15
12
13 # Attempting to execute variable 'x' will raise an error because 'x' is an
# integer
14 # exec(x) # This line will result in a TypeError as 'x' is not callable
15
```

Run Output Clear

```
<class 'code'>
sum = 15
== Code Execution Successful ==
```

Python exec() Function

main.py

```
1 x = 8
2 exec('print(x==8)') # Checks if x is equal to 8 and prints the result
# (True)
3 exec('print(x+4)') # Adds 4 to x and prints the result (12)
4
```

Run Output Clear

```
True
12
== Code Execution Successful ==
```

Python sum() Function

main.py

```
1 s = sum([1, 2, 4])
2 print(s) # This prints the sum of the list: 7
3
4 s = sum([1, 2, 4], 10)
5 print(s) # This prints the sum of the list plus the initial value: 17
6
```

Run Output Clear

```
7
17
== Code Execution Successful ==
```

Python any() Function

main.py

```
1 l = [4, 3, 2, 0]
2 print(any(l)) # Checks if any element is truthy
3
4 l = [0, False]
5 print(any(l)) # Checks if any element is truthy
6
7 l = [0, False, 5]
8 print(any(l)) # Checks if any element is truthy
9
10 l = []
11 print(any(l)) # Checks if any element is truthy
12
```

Run Output Clear

```
True
False
True
False
== Code Execution Successful ==
```

Python ascii() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there are icons for file operations like Open, Save, and Run. The code editor window contains the following code:

```
main.py
1 normalText = 'Python is interesting'
2 print(ascii(normalText)) # Prints the string as is, since it only contains
                           ASCII characters.
3
4 otherText = 'Pythön is interesting'
5 print(ascii(otherText)) # Prints the string with non-ASCII characters
                           replaced by escape sequences.
6
7 print('Pyth\xf6n is interesting') # Directly prints the string with the
                                   non-ASCII character (ö).
```

The output window on the right shows the results of running the code:

```
'Python is interesting'
'Pythufffdn is interesting'
Pythön is interesting
== Code Execution Successful ==
```

Python bytearray()

The screenshot shows a Python code editor interface with a dark theme. On the left, there are icons for file operations like Open, Save, and Run. The code editor window contains the following code:

```
main.py
1 string = "Python is a programming language."
2
3 # string with encoding 'utf-8'
4 arr = bytearray(string, 'utf-8')
5 print(arr)
6
```

The output window on the right shows the results of running the code:

```
bytearray(b'Python is a programming language.')
== Code Execution Successful ==
```

Python eval() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there are icons for file operations like Open, Save, and Run. The code editor window contains the following code:

```
main.py
1 x = 8
2 print(eval('x + 1')) # Evaluates the expression 'x + 1' and prints the
                       result (9)
3
```

The output window on the right shows the results of running the code:

```
9
== Code Execution Successful ==
```

Python float() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there are icons for file operations like Open, Save, and Run. The code editor window contains the following code:

```
main.py
1 # for integers
2 print(float(9)) # Converts integer 9 to float 9.0
3
4 # for floats
5 print(float(8.19)) # Converts the float 8.19 to float 8.19
6
7 # for string floats
8 print(float("-24.27")) # Converts string "-24.27" to float -24.27
9
10 # for string floats with whitespaces
11 print(float(" -17.19\n")) # Converts string " -17.19\n" to float -17.19
                            (ignores whitespace and newline)
12
13 # string float error
14 print(float("xyz")) # Raises ValueError as "xyz" is not a valid number
```

The output window on the right shows the results of running the code, including some errors:

```
9.0
8.19
-24.27
-17.19
ERROR!
Traceback (most recent call last):
  File "<main.py>", line 14, in <module>
ValueError: could not convert string to float: 'xyz'
== Code Exited With Errors ==
```

Python format() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there are icons for file operations like Open, Save, and Run. The code editor window contains the following code:

```
main.py
1 # integer
2 print(format(123, "d")) # Formats the integer 123 as a decimal (default
                           format)
3
4 # float arguments
5 print(format(123.4567898, "f")) # Formats the float 123.4567898 as a fixed
                                   -point number
6
7 # binary format
8 print(format(12, "b")) # Converts the integer 12 to its binary
                           representation
9
```

The output window on the right shows the results of running the code:

```
123
123.456790
1100
== Code Execution Successful ==
```

Python frozenset() Function

```

main.py | Run | Output | Clear
1 # tuple of letters
2 letters = ('m', 'r', 'o', 't', 's')
3
4 # Create a frozenset from the tuple
5 fSet = frozenset(letters)
6
7 # Print the frozenset
8 print('Frozen set is:', fSet)
9
10 # Print an empty frozenset
11 print('Empty frozen set is:', frozenset())
12

```

Frozen set is: frozenset({'m', 'o', 'r', 's', 't'})
Empty frozen set is: frozenset()
== Code Execution Successful ==

Python getattr() Function

```

main.py | Run | Output | Clear
1 class Details:
2     age = 22
3     name = "Phill"
4
5 # Create an object of the Details class
6 details = Details()
7
8 # Accessing the 'age' attribute using getattr()
9 print('The age is:', getattr(details, "age"))
10
11 # Accessing the 'age' attribute using direct access
12 print('The age is:', details.age)
13

```

The age is: 22
The age is: 22
== Code Execution Successful ==

Python globals() Function

```

main.py | Run | Output | Clear
1 age = 22
2
3 # Setting age through globals() function
4 globals()['age'] = 22
5
6 # Printing the value of age
7 print('The age is:', age)
8

```

The age is: 22
== Code Execution Successful ==

Python hasattr() Function

```

main.py | Run | Output | Clear
1 l = [4, 3, 2, 0]
2 print(any(l)) # Checking if any value in the list is truthy
3
4 l = [0, False]
5 print(any(l)) # Checking if any value in the list is truthy
6
7 l = [0, False, 5]
8 print(any(l)) # Checking if any value in the list is truthy
9
10 l = []
11 print(any(l)) # Checking if any value in the empty list is truthy
12

```

True
False
True
False
== Code Execution Successful ==

Python iter() Function

```

main.py | Run | Output | Clear
1 # list of numbers
2 list = [1, 2, 3, 4, 5]
3
4 # create an iterator from the list
5 listIter = iter(list)
6
7 # prints '1'
8 print(next(listIter))
9
10 # prints '2'
11 print(next(listIter))
12
13 # prints '3'
14 print(next(listIter))
15
16 # prints '4'
17 print(next(listIter))
18
19 # prints '5'
20 print(next(listIter))

```

Output:

```

1
2
3
4
5
== Code Execution Successful ==

```

Python list()

```

main.py | Run | Output | Clear
1 # empty list
2 print(list()) # Output: []
3
4 # string
5 String = 'abcde'
6 print(list(String)) # Output: ['a', 'b', 'c', 'd', 'e']
7
8 # tuple
9 Tuple = (1, 2, 3, 4, 5)
10 print(list(Tuple)) # Output: [1, 2, 3, 4, 5]
11
12 # list
13 List = [1, 2, 3, 4, 5]
14 print(list(List)) # Output: [1, 2, 3, 4, 5]

```

Output:

```

[]
['a', 'b', 'c', 'd', 'e']
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
== Code Execution Successful ==

```

Python locals() Function

```

main.py | Run | Output | Clear
1 def localsAbsent():
2     return locals() # Returns a dictionary of local variables in the
                     # current scope (function)
3
4 def localsPresent():
5     present = True # Define a local variable 'present'
6     return locals() # Returns a dictionary of local variables, including
                     # 'present'
7
8 print('localsAbsent:', localsAbsent()) # Prints the dictionary of
                                         # local variables in 'localsAbsent'
9 print('localsPresent:', localsPresent()) # Prints the dictionary of
                                         # local variables in 'localsPresent'
10

```

Output:

```

localsAbsent: {}
localsPresent: {'present': True}
== Code Execution Successful ==

```

Python map() Function

The screenshot shows a Python code editor interface. On the left, there's a sidebar with icons for file operations like Open, Save, and Run, along with tabs for Python, QR, DB, and JS. The main area has a dark background with white text. A toolbar at the top includes icons for copy/paste, brightness, share, and a blue 'Run' button. To the right of the code area is an 'Output' panel with a 'Clear' button. The code itself is as follows:

```
main.py
1 def calculateAddition(n):
2     return n + n # This function doubles the input number
3
4 numbers = (1, 2, 3, 4) # A tuple of numbers
5
6 # Applying calculateAddition function to each element in the numbers tuple
#       using map
7 result = map(calculateAddition, numbers)
8
9 # Print the map object (which is an iterator)
10 print(result)
11
12 # Convert the map object to a set and print the result
13 numbersAddition = set(result)
14 print(numbersAddition)
15
```

The output panel shows the results of the execution:

```
<map object at 0x7a3795b31450>
{8, 2, 4, 6}
== Code Execution Successful ==
```

Python memoryview() Function

This screenshot shows another instance of a Python code editor. The layout is identical to the first one, with a sidebar, toolbar, and output panel. The code in 'main.py' demonstrates the use of the `memoryview` function:

```
main.py
1 # A random bytearray
2 randomByteArray = bytearray('ABC', 'utf-8')
3
4 # Create a memoryview of the bytearray
5 mv = memoryview(randomByteArray)
6
7 # Access the memory view's zeroth index
8 print(mv[0]) # Output: 65, the byte representation of 'A' (ASCII value)
9
10 # Create a byte object from a slice of the memory view (first two elements
#      )
11 print(bytes(mv[0:2])) # Output: b'AB' (byte representation of 'A' and 'B'
#      )
12
13 # Create a list from the memory view (first three elements)
14 print(list(mv[0:3])) # Output: [65, 66, 67] (ASCII values of 'A', 'B',
#      and 'C')
15
```

The output panel shows the results:

```
65
b'AB'
[65, 66, 67]
== Code Execution Successful ==
```

Python object() Function

The screenshot shows a Python code editor interface. On the left is the code editor window with the file name 'main.py'. The code defines a variable 'python' as an object and prints its type and dir. On the right is the 'Output' window which displays the results of running the code. The output shows the class 'object' and its methods.

```
1 python = object()
2
3 print(type(python))
4 print(dir(python))
5
```

```
<class 'object'>
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__',
 '__init__', '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__']

== Code Execution Successful ==
```

Python chr() Function

The screenshot shows a Python code editor interface. On the left is the code editor window with the file name 'main.py'. The code calls chr(102) and chr(112), prints the results, and checks if they are strings. On the right is the 'Output' window which displays the results of running the code. The output shows the characters 'f' and 'p' and their string type.

```
1 # Calling function
2 result = chr(102) # It returns string representation of a char
3 result2 = chr(112)
4
5 # Displaying result
6 print(result)
7 print(result2)
8
9 # Verify, is it string type?
10 print("is it string type:", type(result) is str)
11
```

```
f
p
is it string type: True

== Code Execution Successful ==
```

Python complex() Function

The screenshot shows a Python code editor interface. On the left is the code editor window with the file name 'main.py'. The code demonstrates the complex() function with single and double parameter passing. On the right is the 'Output' window which displays the results of running the code. The output shows complex numbers (1+0j) and (1+2j).

```
1 # Python complex() function example
2 # Calling function
3 a = complex(1) # Passing single parameter
4 b = complex(1, 2) # Passing both parameters
5
6 # Displaying result
7 print(a)
8 print(b)
9
```

```
(1+0j)
(1+2j)

== Code Execution Successful ==
```

Python delattr() Function

The screenshot shows a Python code editor interface. On the left is the code editor window with the file name 'main.py'. The code creates a Student class, creates an instance, and attempts to remove the 'course' attribute using delattr. On the right is the 'Output' window which displays the error message. The error indicates that the 'Student' type has no attribute 'course'.

```
1 class Student:
2     id = 101
3     name = "Pranshu"
4     email = "pranshu@abc.com"
5
6     # Declaring function
7     def getinfo(self):
8         print(self.id, self.name, self.email)
9
10 s = Student()
11 s.getinfo()
12
13 delattr(Student, 'course') # Removing attribute which is not available
14
15 s.getinfo() # error: throws an error
16
```

```
101 Pranshu pranshu@abc.com
ERROR!
Traceback (most recent call last):
  File "<main.py>", line 13, in <module>
AttributeError: type object 'Student' has no attribute 'course'

== Code Exited With Errors ==
```

Python dir() Function

The screenshot shows a Python code editor interface. On the left is the code editor window with the file name 'main.py'. The code calls dir() on the builtins module. On the right is the 'Output' window which displays the results of running the code. The output shows the list of attributes for the builtins module.

```
1 # Calling function
2 att = dir()
3
4 # Displaying result
5 print(att)
6
```

Python divmod() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Python divmod() function example
2 # Calling function
3 result = divmod(10, 2)
4
5 # Displaying result
6 print(result)
7
```

The 'Output' tab shows the results of running the code:

```
(5, 0)
== Code Execution Successful ==
```

Python enumerate() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Calling function
2 result = enumerate([1, 2, 3])
3
4 # Displaying result
5 print(result)
6 print(list(result))
7
```

The 'Output' tab shows the results of running the code:

```
<enumerate object at 0x7f09696736a0>
[(0, 1), (1, 2), (2, 3)]
== Code Execution Successful ==
```

Python dict()

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Calling function
2 result = dict() # returns an empty dictionary
3 result2 = dict(a=1, b=2)
4
5 # Displaying result
6 print(result)
7 print(result2)
8
```

The 'Output' tab shows the results of running the code:

```
{}
{'a': 1, 'b': 2}
== Code Execution Successful ==
```

Python filter() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Python filter() function example
2 def filterdata(x):
3     if x > 5:
4         return x
5
6 # Calling function
7 result = filter(filterdata, (1, 2, 6))
8
9 # Displaying result
10 print(list(result))
11
```

The 'Output' tab shows the results of running the code:

```
[6]
== Code Execution Successful ==
```

Python hash() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Python hash() Function Example
2 result = hash(21) # integer value
3 result2 = hash(22.2) # decimal value
4 print(result)
5 print(result2)
6
```

The 'Output' tab shows the results of running the code:

```
21
461168601842737174
== Code Execution Successful ==
```

Python help() Function

main.py

```
1 # Python help() Function Example
2 info = help() # No argument
3 print(info)
4
```

Output

```
Welcome to Python 3.12's help utility! If this is your first time using Python, you should definitely check out the tutorial at https://docs.python.org/3.12/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To get a list of available modules, keywords, symbols, or topics, enter "modules", "keywords", "symbols", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose name or summary contain a given string such as "spam", enter "modules spam". To quit this help utility and return to the interpreter, enter "q" or "quit".
```

help>

Python min() Function

main.py

```
1 # Python min() Function Example
2 small = min(2225, 325, 2025) # returns smallest element
3 small2 = min(1000.25, 2025.35, 5625.36, 10052.50)
4 print(small)
5 print(small2)
6
```

Output

```
325
1000.25
== Code Execution Successful ==
```

Python set() Function

main.py

```
1 # Python set() Function Example
2 result = set() # empty set
3 result2 = set('12')
4 result3 = set('javatpoint')
5 print(result)
6 print(result2)
7 print(result3)
8
```

Output

```
set()
{'2', '1'}
{'p', 'j', 'a', 'v', 'n', 't', 'i', 'o'}
```

== Code Execution Successful ==

Python hex() Function

main.py

```
1 # Python hex() Function Example
2 result = hex(1)
3 result2 = hex(342)
4 print(result)
5 print(result2)
6
```

Output

```
0x1
0x156
== Code Execution Successful ==
```

Python id() Function

main.py

```
1 # Python id() Function Example
2 val = id("Javatpoint") # string object
3 val2 = id(1200) # integer object
4 val3 = id([25, 336, 95, 236, 92, 3225]) # List object
5 print(val)
6 print(val2)
7 print(val3)
8
```

Output

```
135731578803312
135731580153360
135731581979008
== Code Execution Successful ==
```

Python setattr() Function

```

main.py | Run | Output
1 # Python setattr() Function Example
2 class Student:
3     id = 0
4     name = ""
5
6     def __init__(self, id, name):
7         self.id = id
8         self.name = name
9
10 student = Student(102, "Sohan")
11 print(student.id)
12 print(student.name)
13 # Adding new attribute
14 setattr(student, 'email', 'sohan@abc.com')
15 print(student.email)

```

Output:

```

102
Sohan
sohan@abc.com
== Code Execution Successful ==

```

Python slice() Function

```

main.py | Run | Output | Clear
1 # Python slice() Function Example
2 result = slice(5) # returns slice object
3 result2 = slice(0, 5, 3) # returns slice object
4 print(result)
5 print(result2)
6

```

Output:

```

slice(None, 5, None)
slice(0, 5, 3)
== Code Execution Successful ==

```

Python sorted() Function

```

main.py | Run | Output | Clear
1 # Python sorted() Function Example
2 str = "javatpoint" # declaring string
3 sorted1 = sorted(str) # sorting string
4 print(sorted1)
5

```

Output:

```

['a', 'a', 'i', 'j', 'm', 'o', 'p', 't', 't', 'v']
== Code Execution Successful ==

```

Python next() Function

```

main.py | Run | Output | Clear
1 # Python next() Function Example
2 number = iter([256, 32, 82]) # Creating iterator
3 item = next(number)
4 print(item)
5 item = next(number)
6 print(item)
7 item = next(number)
8 print(item)
9

```

Output:

```

256
32
82
== Code Execution Successful ==

```

Python input() Function

```

main.py | Run | Output | Clear
1 # Python input() Function Example
2 val = input("Enter a value: ")
3 print("You entered:", val)
4

```

Output:

```

Enter a value: 45
You entered: 45
== Code Execution Successful ==

```

Python int() Function

```

main.py | Run | Output | Clear
1 # Python int() Function Example
2 val = int(10) # integer value
3 val2 = int(10.52) # float value
4 val3 = int('10') # string value
5 print("integer values:", val, val2, val3)
6

```

Output:

```

integer values: 10 10.52 10
== Code Execution Successful ==

```

Python isinstance() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Python isinstance() Function Example
2 class Student:
3     id = 101
4     name = "John"
5
6     def __init__(self, id, name):
7         self.id = id
8         self.name = name
9
10 student = Student(1010, "John")
11 lst = [12, 34, 5, 6, 767]
12 print(isinstance(student, Student)) # isinstance of Student class
13 print(isinstance(lst, Student))
14
```

The 'Output' tab shows the results of running the code:

```
True
False
== Code Execution Successful ==
```

Python oct() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Python oct() Function Example
2 val = oct(10)
3 print("Octal value of 10: ", val)
4
```

The 'Output' tab shows the results of running the code:

```
Octal value of 10: 0o12
== Code Execution Successful ==
```

Python ord() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Python ord() Function Example
2 print(ord('A'))
3 print(ord('R'))
4 print(ord('&'))
5
```

The 'Output' tab shows the results of running the code:

```
56
82
38
== Code Execution Successful ==
```

Python pow() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Python pow() Function Example
2 print(pow(4, 2))
3 print(pow(-4, 2))
4 print(pow(4, -2))
5 print(pow(-4, -2))
6
```

The 'Output' tab shows the results of running the code:

```
16
16
0.0625
0.0625
== Code Execution Successful ==
```

Python print() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Python print() Function Example
2 print("Python is programming language.")
3 x = 7
4 print("x =", x)
5 y = x
6 print('x =', x, 'y =', y)
7
```

The 'Output' tab shows the results of running the code:

```
Python is programming language.
x = 7
x = 7 = y
== Code Execution Successful ==
```

Python range() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Python range() Function Example
2 print(list(range(0)))
3 print(list(range(4)))
4 print(list(range(1, 7)))
5
```

The 'Output' tab shows the results of running the code:

```
[]
[0, 1, 2, 3]
[1, 2, 3, 4, 5, 6]
== Code Execution Successful ==
```

Python reversed() Function

Python reversed() Function

```
main.py
1 # Python reversed() Function Example
2 String = 'Java'
3 print(list(reversed(String)))
4 Tuple = ('J', 'a', 'v', 'a')
5 print(list(reversed(Tuple)))
6 Range = range(8, 12)
7 print(list(reversed(Range)))
8 List = [1, 2, 7, 5]
9 print(list(reversed(List)))
```

Output

```
['a', 'v', 'a', 'J']
['a', 'v', 'a', 'J']
[11, 10, 9, 8]
[5, 7, 2, 1]
--- Code Execution Successful ---
```

Python round() Function

```
main.py
1 # Python round() Function Example
2 print(round(10))
3 print(round(10.8))
4 print(round(6.6))
5
```

Output

```
10
11
7
--- Code Execution Successful ---
```

Python issubclass() Function

```
main.py
1 # Python issubclass() Function Example
2 class Rectangle:
3     def __init__(rectangleType):
4         print('Rectangle is a ', rectangleType)
5
6 class Square(Rectangle):
7     def __init__(self):
8         Rectangle.__init__('square')
9
10 print(issubclass(Square, Rectangle))
11 print(issubclass(Square, list))
12 print(issubclass(Square, (list, Rectangle)))
13 print(issubclass(Rectangle, (list, Rectangle)))
```

Output

```
True
False
True
True
--- Code Execution Successful ---
```

Python str

```
main.py
1 # Python str() Function Example
2 print(str('4'))
3
```

Output

```
4
--- Code Execution Successful ---
```

Python tuple() Function

```
main.py
1 # Python tuple() Function Example
2 t1 = tuple()
3 print('t1=', t1)
4 t2 = tuple([1, 6, 9])
5 print('t2=', t2)
6 t1 = tuple('Java')
7 print('t1=', t1)
8 t1 = tuple({4: 'four', 5: 'five'})
9 print('t1=', t1)
10
```

Output

```
t1= ()
t2= (1, 6, 9)
t1= ('J', 'a', 'v', 'a')
t1= (4, 5)
--- Code Execution Successful ---
```

Python type()

```
main.py
1 # Python type() Function Example
2 List = [4, 5]
3 print(type(List))
4 Dict = {4: 'four', 5: 'five'}
5 print(type(Dict))
6 class Python:
7     a = 0
8 InstanceofPython = Python()
9 print(type(InstanceofPython))
```

Output

```
<class 'list'>
<class 'dict'>
<class '__main__.Python'>
--- Code Execution Successful ---
```

Python vars() function

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Python vars() Function Example
2 class Python:
3     def __init__(self, x = 7, y = 9):
4         self.x = x
5         self.y = y
6
7 InstanceOfPython = Python()
8 print(vars(InstanceOfPython))
9
```

The 'Output' tab shows the result of running the code: {'x': 7, 'y': 9} and a message === Code Execution Successful ===.

Python zip() Function

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Python zip() Function Example
2 numList = [4, 5, 6]
3 strList = ['four', 'five', 'six']
4 result = zip()
5 resultList = list(result)
6 print(resultList)
7 result = zip(numList, strList)
8 resultSet = set(result)
9 print(resultSet)
```

The 'Output' tab shows the result of running the code: [] and {(5, 'five'), (4, 'four'), (6, 'six')}, followed by a message === Code Execution Successful ===.

Python Lambda Functions

Code 1:

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Code to demonstrate how we can use a lambda function for adding 4 to the
2 # input number
3 add = lambda num: num + 4
4 print(add(6))
```

The 'Output' tab shows the result of running the code: 10 and a message === Code Execution Successful ===.

Program Code 2:

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Code to demonstrate a function using def to add 4 to the input number
2 def add(num):
3     return num + 4
4 print(add(6))
```

The 'Output' tab shows the result of running the code: 10 and a message === Code Execution Successful ===.

Program Code 3:

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Code to demonstrate a lambda function that multiplies two numbers
2 a = lambda x, y : (x * y)
3 print(a(4, 5))
```

The 'Output' tab shows the result of running the code: 20 and a message === Code Execution Successful ===.

Program Code 4:

The screenshot shows a Python code editor interface with a dark theme. On the left, there is a sidebar with various icons for file operations like Open, Save, and Run. The main area has tabs for 'main.py' and 'Output'. The code in 'main.py' is:

```
1 # Code to demonstrate a lambda function that adds three numbers
2 a = lambda x, y, z : (x + y + z)
3 print(a(4, 5, 5))
```

The 'Output' tab shows the result of running the code: 14 and a message === Code Execution Successful ===.

What's the Distinction Between Lambda and Def Functions?

The screenshot shows a Python code editor interface. On the left is the code file 'main.py' containing:

```
1 # Python code to show the reciprocal of the given number
2 def reciprocal(num):
3     return 1 / num
4
5 lambda_reciprocal = lambda num: 1 / num
6
7 print("Def keyword: ", reciprocal(6))
8 print("Lambda keyword: ", lambda_reciprocal(6))
9
```

On the right is the 'Output' pane showing the results of running the code:

```
Def keyword: 0.1666666666666666
Lambda keyword: 0.1666666666666666
== Code Execution Successful ==
```

Using Lambda Function with filter()

The screenshot shows a Python code editor interface. On the left is the code file 'main.py' containing:

```
1 # Code to filter the odd numbers from the given list using lambda function
# with filter()
2 list_ = [35, 12, 69, 55, 75, 14, 73]
3 odd_list = list(filter(lambda num: (num % 2 != 0), list_))
4 print('The list of odd number is:', odd_list)
5
```

On the right is the 'Output' pane showing the results of running the code:

```
The list of odd number is: [35, 69, 55, 75, 73]
== Code Execution Successful ==
```

Using Lambda Function with map()

The screenshot shows a Python code editor interface. On the left is the code file 'main.py' containing:

```
1 # Code to square all the entries in a list using the map() function
2 numbers_list = [2, 4, 5, 1, 3, 7, 8, 9, 10]
3 squared_list = list(map(lambda num: num ** 2, numbers_list))
4 print('Square of each number in the given list:', squared_list)
5
```

On the right is the 'Output' pane showing the results of running the code:

```
Square of each number in the given list: [4, 16, 25, 1, 9, 49, 64, 81, 100]
== Code Execution Successful ==
```

Using Lambda Function with List Comprehension

The screenshot shows a Python code editor interface. On the left is the code file 'main.py' containing:

```
1 # Code to calculate the square of each number using list comprehension with
# lambda
2 squares = [lambda num=num: num ** 2 for num in range(0, 11)]
3 for square in squares:
4     print('The square value of all numbers from 0 to 10:', square(), end="")
5
```

On the right is the 'Output' pane showing the results of running the code:

```
The square value of all numbers from 0 to 10: 0 The square value of all numbers
from 0 to 10: 1 The square value of all numbers from 0 to 10: 4 The square
value of all numbers from 0 to 10: 9 The square value of all numbers from 0
to 10: 16 The square value of all numbers from 0 to 10: 25 The square value
of all numbers from 0 to 10: 36 The square value of all numbers from 0 to 10:
49 The square value of all numbers from 0 to 10: 64 The square value of
all numbers from 0 to 10: 81 The square value of all numbers from 0 to 10:
100
== Code Execution Successful ==
```

Using Lambda Function with if-else

The screenshot shows a Python code editor interface. On the left is the code file 'main.py' containing:

```
1 # Code to use lambda function with if-else block
2 Minimum = lambda x, y: x if (x < y) else y
3 print('The greater number is:', Minimum(35, 74))
4
```

On the right is the 'Output' pane showing the results of running the code:

```
The greater number is: 35
== Code Execution Successful ==
```

Using Lambda with Multiple Statements

The screenshot shows a Python code editor interface. On the left is the code file 'main.py' containing:

```
1 # Code to find the third largest number from each sublist using lambda
# functions
2 my_List = [[3, 5, 8, 6], [23, 54, 12, 87], [1, 2, 4, 12, 5]]
3 sort_list = lambda num: (sorted(n) for n in num)
4 third_largest = lambda num, func: [l[func(num)] - 2] for l in func(num)]
5 result = third_largest(my_List, sort_list)
6 print('The third largest number from every sub list is:', result)
7
```

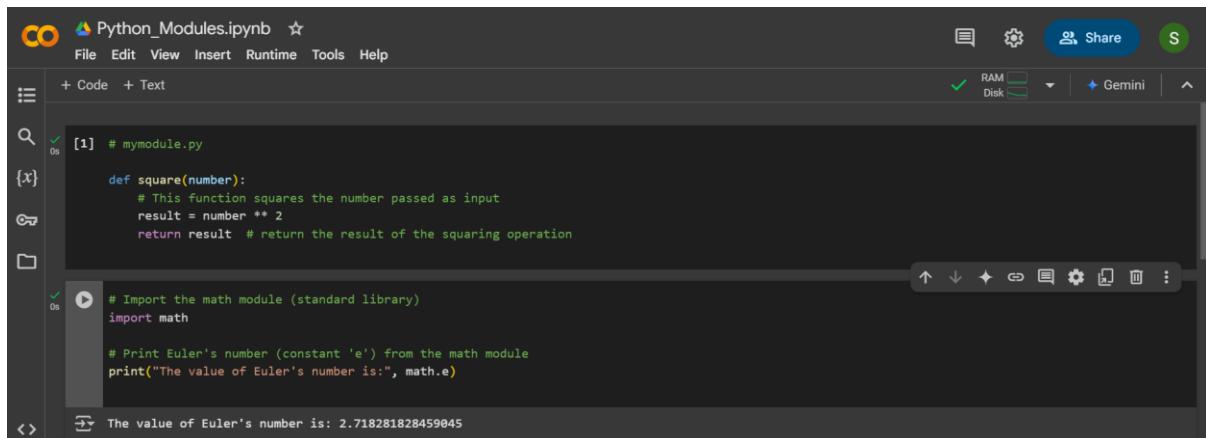
On the right is the 'Output' pane showing the results of running the code:

```
The third largest number from every sub list is: [6, 54, 5]
== Code Execution Successful ==
```

Python Modules

Modular programming is the practice of segmenting a single, complicated coding task into multiple, simpler, easier-to-manage sub-tasks. We call these subtasks modules. Therefore, we can build a bigger program by assembling different modules that act like building blocks.

Python import Statement



```
[1] # mymodule.py
def square(number):
    # This function squares the number passed as input
    result = number ** 2
    return result # return the result of the squaring operation

# Import the math module (standard library)
import math

# Print Euler's number (constant 'e') from the math module
print("The value of Euler's number is:", math.e)
```

The value of Euler's number is: 2.718281828459045

Importing and also Renaming

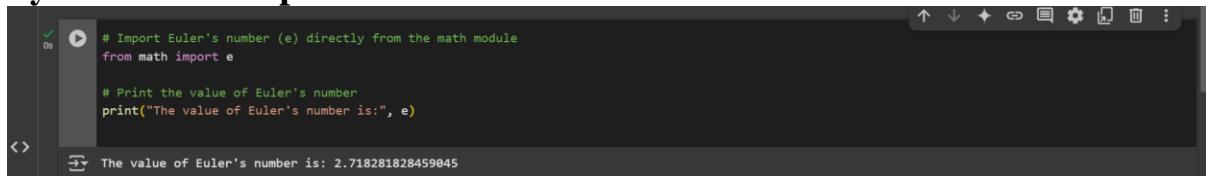


```
# Import the math module and give it an alias 'mt'
import math as mt

# Print Euler's number (constant 'e') from the math module using the alias
print("The value of Euler's number is:", mt.e)
```

The value of Euler's number is: 2.718281828459045

Python from...import Statement



```
# Import Euler's number (e) directly from the math module
from math import e

# Print the value of Euler's number
print("The value of Euler's number is:", e)
```

The value of Euler's number is: 2.718281828459045

Import all Names - From import * Statement



```
#from name_of_module import *
# Import everything from the math module using *
from math import *

# Access functions directly without using the dot operator

# Calculate the square root of 25
print("Calculating square root:", sqrt(25))

# Calculate the tangent of an angle ( $\pi/6$  radians)
print("Calculating tangent of an angle:", tan(pi/6))
```

Calculating square root: 5.0
Calculating tangent of an angle: 0.5773502691896257

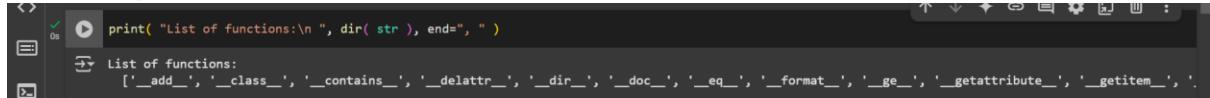
Locating Path of Modules



```
[13] # Here, we are importing the sys module
import sys
# Here, we are printing the path using sys.path
print("Path of the sys module in the system is:", sys.path)
```

Path of the sys module in the system is: ['/content', '/env/python', '/usr/lib/python311.zip', '/usr/lib/python3.11', '/usr/lib/python3.11/lib-dynload']

The dir() Built-in Function



```
print( "List of functions:\n ", dir( str ), end=" " )
```

List of functions:

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__init__', '__iter__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__sizeof__', '__str__', '__subclasshook__']
```

Namespaces and Scoping



```
# Global variable
Number = 204

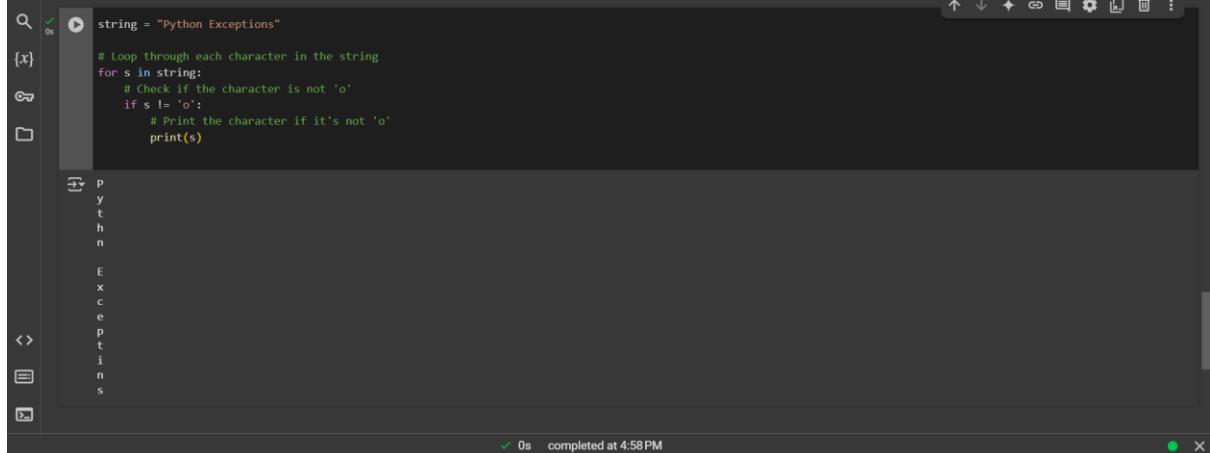
# Define the function AddNumber
def AddNumber():
    # Accessing the global variable 'Number'
    global Number
    # Modify the global 'Number' variable by adding 200
    Number = Number + 200
    # Print the number after addition within the function
    print("The number inside the function is:", Number)

# Call the AddNumber function
AddNumber()

# Print the global 'Number' after calling the function
print("The number outside the function is:", Number)
```

The number inside the function is: 404
The number outside the function is: 404

Python Exceptions



```
string = "Python Exceptions"

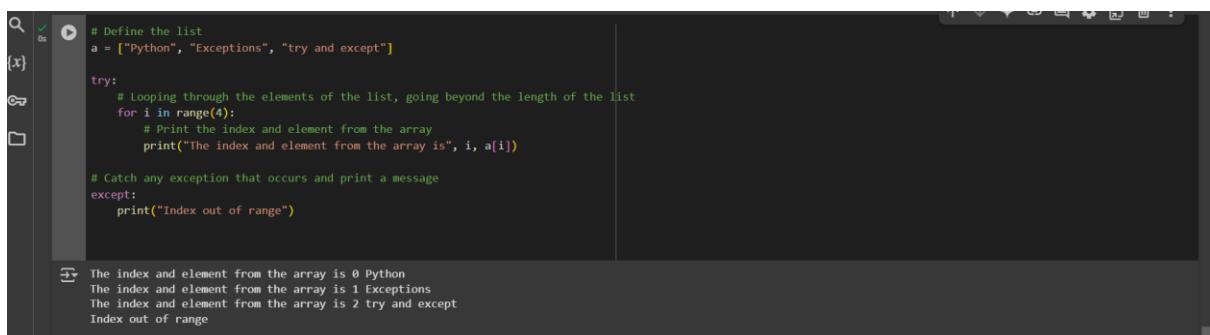
for s in string:
    # Check if the character is not 'o'
    if s != 'o':
        # Print the character if it's not 'o'
        print(s)
```

P
y
t
h
n

E
x
c
e
p
t
i
n
s

✓ 0s completed at 4:58PM

Try and Except Statement - Catching Exceptions



```
# Define the list
a = ["Python", "Exceptions", "try and except"]

try:
    # Looping through the elements of the list, going beyond the length of the list
    for i in range(4):
        # Print the index and element from the array
        print("The index and element from the array is", i, a[i])

    # Catch any exception that occurs and print a message
except:
    print("Index out of range")
```

The index and element from the array is 0 Python
The index and element from the array is 1 Exceptions
The index and element from the array is 2 try and except
Index out of range

How to Raise an Exception

```
try:
    num = [3, 4, 5, 7]
    if len(num) > 3:
        raise Exception(f"Length of the given list must be less than or equal to 3 but is {len(num)}")
except Exception as e:
    print(f"Caught an error: {e}")

Caught an error: Length of the given list must be less than or equal to 3 but is 4
```

Assertions in Python

```
def square_root(Number):
    assert (Number >= 0), "Give a non-negative integer" # Assert non-negative number
    return Number ** (1/2)

# Calling function with valid and invalid inputs
print(square_root(36)) # Valid input, should return 6.0
print(square_root(-36)) # Invalid input, should raise AssertionError

# Calling function and passing the values
print(square_root(36))
print(square_root(-36))
Input In [23], in square_root(Number)
3 def square_root( Number ):
----> 4     assert ( Number >= 0 ), "Give a positive integer"
5     return Number** (1/2)
AssertionError: Give a positive integer
```

Try with Else Clause

```
# Defining a function which returns reciprocal of a number
def reciprocal(num1):
    try:
        # Attempting to calculate the reciprocal
        reci = 1 / num1
    except ZeroDivisionError:
        # Catching division by zero error
        print("We cannot divide by zero")
    else:
        # Executed if no exception occurs
        print(reci)

# Calling the function with valid and invalid inputs
reciprocal(4) # Valid input, should print the reciprocal
reciprocal(0) # Invalid input, should print the error message

0.25
We cannot divide by zero
```

Finally Keyword in Python

```
# Raising an exception in try block
try:
    div = 4 // 0 # This will raise a ZeroDivisionError
    print(div)
except ZeroDivisionError:
    # This block will handle the exception raised
    print("Attempting to divide by zero")
finally:
    # This block will always be executed, no matter if an exception was raised or not
    print('This is code of finally clause')

Attempting to divide by zero
This is code of finally clause
```

User-Defined Exceptions

```
# Defining a custom exception class
class EmptyError(RuntimeError):
    def __init__(self, argument):
        self.arguments = argument

# Code that raises the exception
var = "" # Variable that will be checked
try:
    if not var.strip(): # Check if the variable is empty or contains only whitespace
        raise EmptyError("The variable is empty")
except EmptyError as e: # Catching the custom exception
    print(e.arguments) # Output the exception message

The variable is empty
```

try, except, else, and finally clauses

```
try:
    # Code block
    # These statements are those which can probably have some error
    num1 = 10
    num2 = 0
    result = num1 / num2 # This will raise a ZeroDivisionError
    print(result)
except ZeroDivisionError:
    # This block is optional.
    # If the try block encounters an exception, this block will handle it.
    print("You can't divide by zero!")
else:
    # If there is no exception, this code block will be executed by the Python interpreter
    print("Division was successful")
finally:
    # Python interpreter will always execute this code.
    print("This is the finally block. It always runs, regardless of exceptions.")

# You can't divide by zero!
# This is the finally block. It always runs, regardless of exceptions.
```

Python Arrays

Accessing array element

```
import array as arr

# Creating an array of integers
a = arr.array('i', [2, 4, 5, 6])

# Printing elements by positive index
print("First element is:", a[0])
print("Second element is:", a[1])
print("Third element is:", a[2])
print("Forth element is:", a[3])

# Printing elements by negative index
print("Last element is:", a[-1])
print("Second last element is:", a[-2])
print("Third last element is:", a[-3])
print("Forth last element is:", a[-4])

# Printing all elements using both positive and negative indices
print(a[0], a[1], a[2], a[3], a[-1], a[-2], a[-3], a[-4])

# First element is: 2
# Second element is: 4
# Third element is: 5
# Forth element is: 6
# Last element is: 6
# Second last element is: 5
# Third last element is: 4
# Forth last element is: 2
# 2 4 5 6 6 5 4 2
```

Arrays are mutable, and their elements can be changed similarly to lists.

```
import array as arr

# Creating an array of integers
numbers = arr.array('i', [1, 2, 3, 5, 7, 10])

# Changing the first element (index 0) from 1 to 0
numbers[0] = 0
print(numbers) # Expected Output: array('i', [0, 2, 3, 5, 7, 10])

# Changing the last element (index 5) from 10 to 8
numbers[5] = 8
print(numbers) # Expected Output: array('i', [0, 2, 3, 5, 7, 8])

# Replacing elements from index 2 to 4 with new values [4, 6, 8]
numbers[2:5] = arr.array('i', [4, 6, 8])
print(numbers) # Expected Output: array('i', [0, 2, 4, 6, 8, 8])

# array('i', [0, 2, 3, 5, 7, 10])
# array('i', [0, 2, 3, 5, 7, 8])
# array('i', [0, 2, 4, 6, 8, 8])
```

The elements can be deleted from an array using Python's del statement. If we want to delete any value from the Array, we can use the indices of a particular element.

```
On ① import array as arr # Importing the array module
      # Creating an array of integers
      number = arr.array('i', [1, 2, 3, 3, 4])
      # Using del to remove the third element (index 2)
      del number[2]
      # Printing the array after the element removal
      print(number) # Expected Output: array('i', [1, 2, 3, 4])
      
      ⏷ array('i', [1, 2, 3, 4])
```

Array Concatenation

We can easily concatenate any two arrays using the + symbol.

Example 1:

```
On ① import array as arr # Import the array module
      # Creating two arrays of type 'd' (floating point numbers)
      a = arr.array('d', [1.1, 2.1, 3.1, 2.6, 7.8]) # Array a
      b = arr.array('d', [3.7, 8.6]) # Array b
      # Creating an empty array c
      c = arr.array('d')
      # Concatenating arrays a and b
      c = a + b
      # Printing the resulting array c
      print("Array c = ", c)
      
      ⏷ Array c =  array('d', [1.1, 2.1, 3.1, 2.6, 7.8, 3.7, 8.6])
```

Example 2:

```
On ① import array as arr # Importing the array module
      # Initialize the array with integer values
      x = arr.array('i', [4, 7, 19, 22])
      # Accessing and printing the first element
      print("First element:", x[0]) # Output: 4
      # Accessing and printing the second element
      print("Second element:", x[1]) # Output: 7
      # Accessing and printing the second last element using negative indexing
      print("Second last element:", x[-2]) # Output: 19
      
      ⏷ First element: 4
          Second element: 7
          Second last element: 19
```

Python Decorator

Example

```
On ① def func1(msg): # Function definition with a parameter 'msg'
      print(msg) # Print the message passed as an argument
      func1("Hii, welcome to function ") # Call func1 and pass a string as the argument
      func2 = func1 # Assign Func1 to func2, making func2 another reference to func1
      func2("Hii, welcome to function ") # Call func2 (which references func1) and pass the same message
      ⏷ Hii, welcome to function
          Hii, welcome to function
```

Inner Function

```
✓ 0s  def func(): # Creating the outer function 'func'
    print("We are in first function") # Print message for func

    def func1(): # Creating the first inner function 'func1'
        print("This is first child function") # Print message for func1

    def func2(): # Creating the second inner function 'func2'
        print("This is second child function") # Print message for func2

    func1() # Call the first inner function 'func1'
    func2() # Call the second inner function 'func2'

func() # Call the outer function 'func'

→ We are in first function
This is first child function
This is second child function
```

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

```
✓ 0s  def add(x): # Define a function 'add' that adds 1 to the input 'x'
    return x + 1 # Return the value of 'x + 1'

def sub(x): # Define a function 'sub' that subtracts 1 from the input 'x'
    return x - 1 # Return the value of 'x - 1'

def operator(func, x): # Define a function 'operator' that takes a function and a value as parameters
    temp = func(x) # Call the passed function (add or sub) with 'x' as the argument
    return temp # Return the result of the function call

print(operator(sub, 10)) # Call 'operator' with the 'sub' function and 10, expected to return 9
print(operator(add, 20)) # Call 'operator' with the 'add' function and 20, expected to return 21

→ 9
21
```

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

```
✓ 0s  def hello(): # Define the outer function 'hello'
    def hi(): # Define the inner function 'hi'
        print("Hello") # The 'hi' function prints "Hello"

    return hi # Return the 'hi' function itself, not the result of calling it

new = hello() # Call 'hello', which returns the 'hi' function and store it in 'new'
new() # Call the function stored in 'new', which is actually 'hi', so it prints "Hello"

→ Hello
```

Decorating functions with parameters

```
✓ 0s  def divide(x, y): # Define the function 'divide' that takes two parameters
    print(x / y) # Print the result of dividing x by y

def outer_div(func): # Define a function 'outer_div' that takes a function as a parameter
    def inner(x, y): # Define the inner function that will modify the behavior of 'func'
        if x < y: # If the first number is less than the second number, swap them
            x, y = y, x # Swap x and y
        return func(x, y) # Call the original 'func' with the modified parameters
    return inner # Return the inner function, which is a closure that wraps 'func'

# Create a new function 'divide1' by applying the 'outer_div' decorator to 'divide'
divide1 = outer_div(divide)

# Call 'divide1', which will internally call 'inner', and 'inner' will call 'divide'
divide1(2, 4)

→ 2.0
```

Syntactic Decorator

```
✓ 0s  def outer_div(func): # Define a decorator 'outer_div' that takes a function 'func' as argument
    def inner(x, y): # Define the inner function that will modify the behavior of 'func'
        if x < y: # If the first number is smaller than the second, swap them
            x, y = y, x # Swap the values of x and y
        return func(x, y) # Call the original 'func' with the swapped values
    return inner # Return the 'inner' function which is a modified version of 'func'

@outer_div # Apply the 'outer_div' decorator to 'divide'
def divide(x, y): # Define the 'divide' function that takes two numbers as input
    print(x / y) # Print the result of dividing x by y
```

Reusing Decorator

```
✓ 0s  # 1. Define a decorator function
def do_twice(func): # Here, 'func' is the function that we will decorate
    # 2. Define a wrapper function to call 'func' twice
    def wrapper_do_twice():
        func() # Call the function once
        func() # Call the function again
    # 3. Return the wrapper function
    return wrapper_do_twice

# 4. Now, using the decorator:
# We can import 'do_twice' in another file, but here it's defined in the same script
# from decorator import do_twice # Assuming the decorator is in a file named 'decorator.py'

# 5. Apply the decorator to a function
@do_twice # This is the decorator syntax
def say_hello():
    print("Hello There")

# 6. Calling 'say_hello' will execute the wrapped version that calls the function twice
say_hello() # This will print "Hello There" twice
```

→ Hello There
Hello There

Fancy Decorators

Example: 1- @property decorator - By using it, we can use the class function as an attribute. Consider the following code:

```
✓ 0s  class Student: # here, we are creating a class with the name Student
    def __init__(self, name, grade): # Constructor to initialize the attributes
        self.name = name
        self.grade = grade

    @property # Using the property decorator for the display method
    def display(self): # Property method to get the student's name and grade
        return self.name + " got grade " + self.grade

    # Create an instance of the Student class
    stu = Student("John", "B")

    # Accessing attributes
    print("Name of the student: ", stu.name) # Prints the name of the student
    print("Grade of the student: ", stu.grade) # Prints the grade of the student

    # Using the property display to print the name and grade formatted
    print(stu.display) # This prints the formatted string from the @property method
```

→ Name of the student: John
Grade of the student: B
John got grade B

Example: 2-

```
✓ 0s  ► class Person: # here, we are creating a class with the name Person
      @staticmethod
      def hello(): # here, we are defining a static method hello
          print("Hello Peter")

      # Creating an instance of the Person class
      per = Person()

      # Calling the hello method on the instance
      per.hello()

      # Calling the hello method directly on the class
      Person.hello()

→ Hello Peter
Hello Peter
```

Decorator with Arguments

It is always useful to pass arguments in a decorator. The decorator can be executed several times according to the given value of the argument. Let us consider the following example:

```
✓ 0s  ► import functools # here, we are importing the functools into our program

def repeat(num): # here, we are defining a function repeat and passing parameter num
    # Here, we are creating and returning a wrapper function
    def decorator_repeat(func):
        @functools.wraps(func) # This preserves the original function's metadata (name, docstring)
        def wrapper(*args, **kwargs):
            for _ in range(num): # here, we are initializing a for loop and iterating till num
                value = func(*args, **kwargs) # Calling the original function
            return value # here, we are returning the value
        return wrapper # here, we are returning the wrapper function
    return decorator_repeat # Return the decorator function

# Here we are passing num as an argument, which repeats the print function 5 times
@repeat(num=5) # The decorator repeats the function call 5 times
def function1(name):
    print(f"{name}") # This function prints the name

# Calling the decorated function
function1("Hello")

→ Hello
Hello
Hello
Hello
Hello
```

Stateful Decorators

```
✓ 0s ⏪ import functools # here, we are importing the functools into our program

def count_function(func):
    # here, we are defining a function and passing the parameter func
    @functools.wraps(func)
    def wrapper_count_calls(*args, **kwargs):
        wrapper_count_calls.num_calls += 1
        print(f"Call {wrapper_count_calls.num_calls} of {func.__name__!r}")
        return func(*args, **kwargs)

    # Initialize the num_calls attribute
    wrapper_count_calls.num_calls = 0
    return wrapper_count_calls # here, we are returning the wrapper count calls

@count_function # Decorator is applied here
def say_hello():
    # here, we are defining a function that prints a message
    print("Say Hello")

# Calling the decorated function
say_hello()
say_hello()

➡ Call 1 of 'say_hello'
Say Hello
Call 2 of 'say_hello'
Say Hello
```

Classes as Decorators

```
✓ 0s ⏪ import functools # here, we are importing the functools into our program

class Count_Calls:
    # here, we are creating a class for getting the call count
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        self.num_calls = 0

    def __call__(self, *args, **kwargs):
        self.num_calls += 1
        print(f"Call {self.num_calls} of {self.func.__name__!r}")
        return self.func(*args, **kwargs)

@Count_Calls # Decorator applied here
def say_hello():
    # here, we are defining a function and passing the parameter
    print("Say Hello")

# Calling the decorated function
say_hello()
say_hello()
say_hello()

➡ Call 1 of 'say_hello'
Say Hello
Call 2 of 'say_hello'
Say Hello
Call 3 of 'say_hello'
Say Hello
```