1. What is a lambda function in Python, and how does it differ from a regular function?

Ans:-
A lambda function in Python is a type of anonymous function, also known as a lambda expression. It is a compact way of defining small, simple functions without the need to give them a formal name using the def keyword. Lambda functions are often used for short operations that can be expressed in a single line of code.
(1) Syntax: Lambda functions use a more concise syntax compared to regular functions defined using the def keyword. They are often limited to a single expression.

(2) Namelessness: Lambda functions are anonymous by nature. They don't have a formal name like regular functions. This means they're typically used for simple operations where a named function isn't necessary.

(3) Scope: Lambda functions are typically used in more limited scopes, like within another function or as arguments to higher-order functions. Regular functions can have a wider scope and are generally defined at the module level.

For Example:-

In [2]:

```
lambda arguments: expression
add = lambda x, y: x + y
result = add(3, 5)
print(result)
```

```
8
```

2. Can a lambda function in Python have multiple arguments? If yes, how can you define and use them?

Ans:-
Yes, a lambda function in Python can have multiple arguments. The syntax for defining a lambda function with multiple arguments is as follows:
For Example:-

In [3]:

```
# Define a lambda function that calculates the area of a rectangle
calculate_area = lambda length, width: length * width

# Use the lambda function
length = 5
width = 3
area = calculate_area(length, width)
print(f"The area of the rectangle is {area} square units.")
```

```
The area of the rectangle is 15 square units.
```

3. How are lambda functions typically used in Python? Provide an example use case.

Ans:-
Lambda functions in Python are anonymous, small, and inline functions that can be defined using the lambda keyword. They are also known as "lambda expressions." Unlike regular functions defined using the def keyword, lambda functions are used for simple operations and are typically written in a single line of code.

Lambda functions are often used in situations where you need a small function for a short-lived purpose, such as when passing a function as an argument to another function or when performing operations like sorting or filtering.

In [4]:

```
# List of tuples containing names and scores
scores = [("Alice", 95), ("Bob", 80), ("Charlie", 70), ("David", 88)]

# Sorting the list based on the second element (score)
sorted_scores = sorted(scores, key=lambda x: x[1], reverse=True)
print(sorted_scores)
```

```
[('Alice', 95), ('David', 88), ('Bob', 80), ('Charlie', 70)]
```

4. What are the advantages and limitations of lambda functions compared to regular functions in Python?

Ans:-
Lambda functions in Python, also known as anonymous functions, have some advantages and limitations compared to regular functions.
(a) Advantages of Lambda Functions:-

(1) Conciseness: Lambda functions are more concise than regular functions. They allow you to define simple functions in a single line of code, which can be especially useful for small tasks.

(2) Readability: For simple operations, lambda functions can make your code more readable by keeping the function definition close to where it's used.

(3) Functional Programming: Lambda functions are well-suited for functional programming concepts like mapping, filtering, and reducing. They can be easily used as arguments for functions like map(), filter(), and sorted().

(4) No Need for Named Functions: Lambda functions are particularly useful when you need a small function for a short period of time and don't want to define a separate named function.

(b) Limitations of Lambda Functions:-

(1) Limited Expressiveness: Lambda functions are restricted to a single expression. They can't contain multiple statements or complex logic. Regular functions, on the other hand, can have multiple lines of code and more intricate structures.

(2) Lack of Documentation: Lambda functions can make code less self-explanatory since they lack descriptive names. Regular functions with meaningful names can provide better documentation and understanding of the code's purpose.

(3) Reduced Reusability: Lambda functions are designed for small, specific tasks. If you find yourself needing the same functionality in multiple places, a regular named function would be more appropriate for better code reusability.

(4) Limited Use Cases: Due to their simplicity and lack of statements, lambda functions are not suitable for tasks that involve control flow structures like loops and conditionals. Regular functions are better suited for more complex operations.

(5) Debugging Challenges: Debugging lambda functions can be more challenging because they don't have named identifiers, making it harder to identify which specific lambda caused an issue.

5. Are lambda functions in Python able to access variables defined outside of their own scope?Explain with an example.

Ans:-
Yes, lambda functions in Python are able to access variables defined outside of their own scope. They have access to variables from the enclosing scope in which they are defined. This is because lambda functions are essentially just a shorthand way to define small anonymous functions, and they follow the same rules for variable scoping as regular named functions.

For Example:-

In [5]:

```
x = 10  # A variable defined in the outer/global scope

# Define a lambda function that uses the variable 'x' from its enclosing scope
lambda_function = lambda y: x + y

result = lambda_function(5)
print(result)
```

```
15
```

6. Write a lambda function to calculate the square of a given number.

In [6]:

```
square = lambda x: x ** 2

# Example usage
number = 5
result = square(number)
print("Square of", number, "is", result)
```

```
Square of 5 is 25
```

7. Create a lambda function to find the maximum value in a list of integers.

In [7]:

```
find_max = lambda lst: max(lst)
my_list = [12, 45, 67, 23, 89, 34]
maximum_value = find_max(my_list)
print("The maximum value is:", maximum_value)
```

```
The maximum value is: 89
```

8. Implement a lambda function to filter out all the even numbers from a list of integers.

Ans:-
Here's an example of how you can implement a lambda function to filter out all the even numbers from a list of integers in Python:

In [8]:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Using a lambda function to filter out even numbers
filtered_numbers = list(filter(lambda x: x % 2 != 0, numbers))

print(filtered_numbers)
```

```
[1, 3, 5, 7, 9]
```

9. Write a lambda function to sort a list of strings in ascending order based on the length of each string.

Ans:-

Here's a lambda function in Python that sorts a list of strings in ascending order based on the length of each string:

In [10]:

```
strings = ["apple", "banana", "cherry", "date", "mango"]

sorted_strings = sorted(strings, key=lambda x: len(x))
print(sorted_strings)
```

```
['date', 'apple', 'mango', 'banana', 'cherry']
```

10. Create a lambda function that takes two lists as input and returns a new list containing the common elements between the two lists.

Ans:-
Here's a Python lambda function that takes two lists as input and returns a new list containing the common elements between the two lists:

In [11]:

```
common_elements = lambda list1, list2: list(filter(lambda x: x in list2, list1))

# Example usage
list1 = [1, 2, 3, 4, 5]
list2 = [3, 4, 5, 6, 7]
result = common_elements(list1, list2)
print(result)
```

```
[3, 4, 5]
```

11. Write a recursive function to calculate the factorial of a given positive integer.

In [13]:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

# Test the function
num = int(input("Enter a positive integer: "))
if num < 0:
    print("Factorial is not defined for negative numbers.")
else:
    result = factorial(num)
    print(f"The factorial of {num} is {result}")
```

```
Enter a positive integer: 5
The factorial of 5 is 120
```

12. Implement a recursive function to compute the nth Fibonacci number.

In [17]:

```
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Test the function
n = int(input("Enter a positive integer: "))
result = fibonacci(n)
print(f"The {n}th Fibonacci number is: {result}")
```

```
Enter a positive integer: 6
The 6th Fibonacci number is: 8
```

13. Create a recursive function to find the sum of all the elements in a given list.

In [18]:

```
def recursive_sum(lst, index=0):
    if index == len(lst):
        return 0
    return lst[index] + recursive_sum(lst, index + 1)

# Example usage
my_list = [1, 2, 3, 4, 5]
result = recursive_sum(my_list)
print("Sum:", result)
```

```
Sum: 15
```

14. Write a recursive function to determine whether a given string is a palindrome.

In [19]:

```
def is_palindrome(s):
    # Base case: if the string has one or zero characters, it's a palindrome
    if len(s) <= 1:
        return True

    # Compare the first and last characters of the string
    if s[0] != s[-1]:
        return False

    # Recursively check the substring without the first and last characters
    return is_palindrome(s[1:-1])

# Test cases
print(is_palindrome("racecar"))
print(is_palindrome("hello"))
print(is_palindrome("madam"))
```

```
True
False
True
```

15. Implement a recursive function to find the greatest common divisor (GCD) of two positive integers.

In [20]:

```
def gcd_recursive(a, b):
    if b == 0:
        return a
    else:
        return gcd_recursive(b, a % b)

# Example usage
num1 = 48
num2 = 18
print("GCD of", num1, "and", num2, "is:", gcd_recursive(num1, num2))
```

```
GCD of 48 and 18 is: 6
```

In [ ]:

```

```