ASSIGNMENT 9

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

Lambda functions in Python are small, anonymous, and inline functions that are defined using the lambda keyword. They are also known as lambda expressions or anonymous functions because they do not have a name like regular functions defined with the def keyword. Lambda functions are typically used for simple operations and are often used in situations where you need a quick, throwaway function.

Here's the basic syntax of a lambda function:

lambda arguments: expression

Lambda functions can have one or more arguments but can only contain a single expression. The result of evaluating the expression is returned as the result of the lambda function. Lambda functions are often used as arguments to higher-order functions like map(), filter(), and reduce().

Here's an example of a lambda function that calculates the square of a number:

```
square = lambda x: x * x
result = square(5)
print(result) # Output: 25
```

In this example, lambda x: x * x defines a lambda function that takes one argument x and returns x * x. The square variable is assigned this lambda function, and we can then call square(5) to calculate the square of 5.

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

Yes, a lambda function in Python can have multiple arguments. Lambda functions can take any number of arguments, but they are typically used for simple operations. define multiple arguments in a lambda function by separating them with commas, just like in a regular function.

basic syntax:

```
lambda argument1, argument2, ...: expression
```

Here's an example of a lambda function with multiple arguments:

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

```
print(result)                    # Output: 8
```

In this example, the lambda function lambda x, y: x + y takes two arguments, x and y, and returns their sum.

Lambda functions with multiple arguments are often used in functions like map(), filter(), and sorted() when you need to apply a function to elements in a sequence based on multiple inputs. Here's an example using map() to add two lists element-wise:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
result = list(map(lambda x, y: x + y, list1, list2))
print(result)                    # Output: [5, 7, 9]
```

In this example, the lambda function takes two arguments, x and y, and adds them together. The map() function applies this lambda function element-wise to the corresponding elements of list1 and list2, resulting in a new list with the sums.

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

Lambda functions in Python, also known as anonymous functions or lambda expressions, are used for creating small, inline functions without explicitly defining a function using the def keyword. They are typically used in situations where a simple, one-line function is needed, especially when passing functions as arguments to higher-order functions like map, filter, and sorted, or in cases where you want a concise way to define a function.

Sorting a list of dictionaries by a specific key:Suppose you have a list of dictionaries, and you want to sort them based on a specific key within each dictionary. You can achieve this using the `sorted` function with a lambda function as the `key` parameter.

```
people = [
   {'name': 'Alice', 'age': 30},
   {'name': 'Bob', 'age': 25},
   {'name': 'Charlie', 'age': 35},
]
sorted_people = sorted(people, key=lambda x: x['age'])
print(sorted_people)
```

In this example, the lambda function lambda x: x[‘age’] is used as the key function for sorting. It specifies that the sorting should be based on the age key within each dictionary. The result will be a list of dictionaries sorted by age in ascending order:

```
[{'name': 'Bob', 'age': 25}, {'name': 'Alice', 'age': 30}, {'name': 'Charlie', 'age': 35}]
```

Lambda functions are particularly handy for such quick, one-off operations where you don't need to define a full-fledged named function. However, for more complex functions or functions that will be reused, it's better to define them using the def keyword to improve code readability and maintainability**.**

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

Advantages of Lambda Functions:

1.Conciseness: Lambda functions are concise and allow you to define simple, one-liner functions inline. This can make your code more compact and readable when you need a quick, short function.

2. No Need for a Name: Lambda functions are anonymous, meaning you don't need to assign them a name. This can be useful when you only need the function in one place and don't want to clutter your code with unnecessary function names.

3. Readability: Lambda functions can improve the readability of code when used appropriately, especially in cases where the function's purpose is clear from its one-line expression.

4. Functional Programming: Lambda functions are often used in functional programming paradigms, where functions are treated as first-class citizens. They make it easier to work with higher-order functions like map, filter, and sorted.

Limitations of Lambda Functions:

1. Limited Expressiveness: Lambda functions are limited to a single expression, which means they can't contain multiple statements or complex logic. This makes them unsuitable for more complex functions.

2.Lack of Documentation: Since lambda functions are anonymous, they don't have docstrings or meaningful names. This can make it challenging to understand their purpose without looking at the code context.

3.Reduced Reusability: Lambda functions are typically used for one-off, specific tasks. If you need to reuse a function in multiple places, it's better to define a regular function with a descriptive name.

4. Debugging: Debugging lambda functions can be more challenging than debugging regular functions because they lack a name for reference in error messages.

5. Limited Parameter Handling: Lambda functions can only take a single expression as input and return its result. They are less flexible in terms of handling multiple parameters and complex function signatures.

In summary, lambda functions are a powerful tool for creating small, concise functions, especially when working with functional programming constructs and higher-order functions. However, their limitations make them unsuitable for more complex tasks and functions that require documentation, reusability, or debugging ease. In such cases, it's recommended to use regular functions defined with the def keyword.

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

Yes, lambda functions in Python can access variables defined outside of their own scope. This concept is known as lexical scoping or closure. Lambda functions can

"capture" variables from their containing scope and use them within the lambda expression. This behavior is especially useful when you want to create functions with access to specific context variables, such as when defining custom sorting or filtering criteria.

```
# Variable defined in the outer scope
outer_variable = 10

# Lambda function that uses the outer variable
lambda_function = lambda x: x + outer_variable

result = lambda_function(5)
print(result)  # Output: 15
```

In this example, the lambda function lambda x: x + outer_variable captures the outer_variable from the outer scope and uses it within the lambda expression. When you call lambda_function(5), it adds the argument 5 to the captured outer_variable, resulting in 15.

This behavior allows lambda functions to be very flexible when used within a context where they can access and utilize variables from their enclosing scope. However, it's essential to understand that lambda functions are limited to capturing variables and not modifying them. If you want to modify an outer variable within a lambda function, you'll need to declare it as nonlocal within the function or use a regular function with the global keyword for global variables.

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

```
sqr=lambda x:x**2
print(sqr(5))
```

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

```
find_max = lambda lst: max(lst)

# Example usage:
numbers = [12, 45, 78, 32, 9, 67]
result = find_max(numbers)
print(result)
```

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

```
numbers = [12, 45, 78, 32, 9, 67]
find_even = list(filter(lambda x:x%2==0,numbers))
print(find_even)
```

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

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

# Use sorted with a lambda function to sort by string length
sorted_strings = sorted(strings, key=lambda x: len(x))
print(sorted_strings)
```

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

```
l1=list(range(10))
l2=list(range(5,15))

lst=list(filter(lambda x:x in l2,l1))
print(lst)
```

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

```
def fact(n):
    try:
        n=int(n)
        if (n==1) or (n==0):
            return 1
        else:
            return n* fact(n-1)
    except Exception as ex:
        return  f"{ex},please enter integer value"
num=input("enter the number:")
result=fact(num)
print(result)
```

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

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

n = 10
result = fibonacci(n)
print(f"The {n}th Fibonacci number is {result}")
```

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

```
def recursive_list_sum(lst):
    if not lst:
        return 0
    else:
        return lst[0] + recursive_list_sum(lst[1:])


my_list = [1, 2, 3, 4, 5]
result = recursive_list_sum(my_list)
print("Sum of elements in the list:", result)
```

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

```
def is_palindrome(s):
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    return is_palindrome(s[1:-1])

my_string = "racecar"
```

```
result = is_palindrome(my_string)

if result:
    print(f"'{my_string}' is a palindrome.")
else:
    print(f"'{my_string}' is not a palindrome.")
```

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

```
def gcd_recursive(a, b):
    if b == 0:
        return a
    else:
        return gcd_recursive(b, a % b)
num1 = 48
num2 = 18

result = gcd_recursive(num1, num2)
print(f"The GCD of {num1} and {num2} is {result}")
```