Assignment No-9

1. What is a lambda function in Python, and how does it differ from a regular function?
Solution: In Python, a lambda function is a small anonymous function that can take any number of arguments but can only have one expression. It is also known as an anonymous function because it does not have a name.
The syntax for a lambda function is:
lambda arguments: expression
Lambda functions are commonly used when a function is needed for a short period of time or as an argument to another function. They are usually written in a single line of code and are useful for simple computations that can be done in a single expression.
Here is an example of a lambda function that adds two numbers:

"add = lambda x, y: x + y
print(add(5, 3))"

The output of this code will be `8`.
On the other hand, regular functions are defined using the `def` keyword and have a name, a block of code, and a return statement. They can have multiple expressions and statements.
Here is an example of a regular function that performs the same addition:
"def add(x, y):
    return x + y

print(add(5, 3))"

The output of this code will also be `8`.
The main difference between a lambda function and a regular function is that a lambda function does not require a separate `def` statement or a return statement. Lambda functions are typically used for simple tasks whereas regular functions are more suitable for complex computations and code organization.

2. Can a lambda function in Python have multiple arguments? If yes, how can you define and use them?
Solution: Yes, a lambda function in Python can have multiple arguments. You can define and use them by separating the arguments with commas in the lambda expression.

Here's an example of a lambda function with multiple arguments:
"addition = lambda x, y: x + y
print(addition(3, 4)) # Output: 7"

In this example, the lambda function `addition` takes two arguments, `x` and `y', and returns their sum. Like any other function, the function can be called, passing the arguments in parentheses after the function name.

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

Solution: Lambda functions in Python are typically used for small, one-off functions that do not require a block of code. They are commonly used with higher-order functions or when a function is needed as an argument.

One use case for lambda functions is with the `map()` function. `map()` applies a function to each element of an iterable. Instead of defining a separate function, a lambda function can be used to provide a concise and inline definition.

Here's an example:
“numbers = [1, 2, 3, 4, 5]

# Using a lambda function to square each number using a map()
squared_numbers = list(map(lambda x: x**2, numbers))

print(squared_numbers) # Output: [1, 4, 9, 16, 25]”
In this example, we have a list of numbers. The lambda function `lambda x: x**2` is applied to each element of the list using the `map()` function. The lambda function squares each number, resulting in a new list of squared numbers.

4. What are the advantages and limitations of lambda functions compared to regular functions in Python?
Solution: Lambda functions, also known as anonymous functions, have several advantages and limitations compared to regular functions in Python:

Advantages of lambda functions:
1. Concise syntax: Lambda functions are defined using a single line of code, which makes them more concise and easier to read than regular functions.
2. No need for a separate function definition: Lambda functions don't need to be defined separately like regular functions. They can be created directly at the place where they are needed, which saves time and effort.
3. Simplicity: Lambda functions are best suited for small, one-line functions, where the simplicity of the syntax outweighs the need for a full function definition.

Limitations of lambda functions:
1. Limited functionality: Lambda functions are restricted to a single expression and cannot include statements or multiple lines of code. This limits their use in complex scenarios.
2. No function name: Lambda functions are anonymous, meaning they don't have a name. This can make debugging or understanding code difficult when lambda functions are used extensively.
3. Reduced readability: While lambda functions can be concise, they can also be harder to read and understand compared to regular functions, especially for more complex logic.

In summary, lambda functions provide a concise and convenient way to define small, single-line functions without the need for a separate function definition. However, they are limited in functionality, lack a function name, and can reduce code readability in certain cases.

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

Solution: Yes, lambda functions in Python can access variables defined outside of their own scope. This is because lambda functions have access to the variables in the enclosing scope, also known as the lexical scope.

Here is an example to demonstrate this:

```
"def outer_function():
    outer_variable = 10  # Variable defined in the outer scope

    inner_function = lambda x: x + outer_variable  # Lambda function accessing outer_variable

    return inner_function


closure = outer_function()
result = closure(5)
print(result)  # Output: 15"
```

In the example above, the outer_function returns a lambda function inner_function that takes an argument x. The lambda function `inner_function` uses the outer_variable from its enclosing scope to add it to the argument x. When the lambda function is called with 5, it adds 5 to outer_variable (10), resulting in 15. This demonstrates that the lambda function could access and use the outer variable defined outside of its scope.

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

Solution: Here is an example of a lambda function to calculate the square of a given number:

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

# Example usage
num = 5
result = square(num)
print(result)  # output: 25"
```

The lambda function is assigned to the variable 'square' in the above code. It takes a parameter "x" and returns the square of "x" using the exponentiation operator "**". You can then call the lambda function on any number to calculate its square.

In the example usage, we pass 'num' as the argument to the lambda function, which calculates the square and assigns it to the variable 'result'. Finally, we print the result, which is `25`.

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

Solution: Here's an example of a lambda function that finds the maximum value in a list of integers:

“

max_value = lambda lst: max(lst)

“

To use this lambda function, you can simply pass in your list of integers as an argument:

“

numbers = [5, 2, 9, 1, 7]

print(max_value(numbers)) # Output: 9

“

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

Solution: Here is an example of a lambda function that filters out all the even numbers from a list of integers:

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

even_numbers = list (filter (lambda x: x % 2 == 0, numbers))

print(even_numbers) # Output: [2, 4, 6, 8, 10]

“

In this example, we use the `filter` function along with a lambda function to check if each element in the ‘numbers’ list is divisible by 2 (i.e., even). The lambda function takes an input ‘x’ and returns True if ‘x % 2 == 0’, indicating that ‘x’ is an even number. The ‘filter’ function filters out all the elements in the `numbers` list for which the lambda function returns False, resulting in the list of even numbers being assigned to ‘even_numbers’.

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

Solution: lambda x: sorted (x, key=lambda y: len(y))

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

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

“

common_elements = lambda list1, list2: list(set(list1) & set(list2))

“

In this function, ‘set(list1)’ converts ‘list1’ to a set, which eliminates any duplicate elements. Similarly, ‘set(list2)’ converts ‘list2’ to a set. Then, ‘set(list1) & set(list2)’ returns the intersection of the two sets, which is a set containing only the common elements between the two lists. Finally, ‘list ()’ is used to convert the set back to a list.

You can use this lambda function as follows:

“

list1 = [1, 2, 3, 4, 5]

list2 = [4, 5, 6, 7, 8]

common = common_elements (list1, list2)

print(common) # Output: [4, 5]

“

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

Solution: Here is a recursive function in Python to calculate the factorial of a given positive integer:

“

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

“

In this function, we check if the input number ‘n’ is equal to 0. If it is, we return 1, as the factorial of 0 is defined to be 1. Otherwise, we calculate the factorial by multiplying `n` with the result of the factorial of ‘n-1’. This is done recursively until we reach the base case of ‘n = 0’.

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

Solution: An implementation of a recursive function to compute the nth Fibonacci number is:

“

```
def fibonacci(n):
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # Recursive case
    else:
        return fibonacci(n-1) + fibonacci(n-2)

# Test the function
print(fibonacci(0))  # Output: 0
print(fibonacci(1))  # Output: 1
print(fibonacci(5))  # Output: 5
print(fibonacci(10))  # Output: 55
```

“

In this implementation, we have three cases:

1. Base case 1: If n is 0, return 0 as the first Fibonacci number.
2. Base case 2: If n is 1, return 1 as the second Fibonacci number.
3. Recursive case: For any other value of n, compute the Fibonacci number by recursively calling fibonacci(n-1) and fibonacci(n-2) and adding them together.

13. Create a recursive function to find the sum of all the elements in a given list.
Solution: Here's an example of a recursive function in Python to find the sum of all the elements in a given list:
“
```
def recursive_sum(lst):
    if len(lst) == 0:   # base case: empty list
        return 0
    else:
        return lst[0] + recursive_sum(lst[1:])   # recursive case: add first element to sum of remaining elements
```
```

Explanation:

1. The function ‘recursive_sum’ takes a list ‘lst’ as input.
2. The base case of the recursive function is when the list is empty. In this case, the function returns 0 because there are no elements to sum.
3. In the recursive case, the function adds the first element of the list (‘lst[0]’) to the sum of the remaining elements (‘recursive_sum(lst[1:])’). This is done recursively until the base case is reached.

Example usage:
“
```
print(recursive_sum([1, 2, 3, 4, 5]))   # Output: 15
print(recursive_sum([-1, 0, 1]))   # Output: 0
print(recursive_sum([]))   # Output: 0 (empty list)
```
“

**14. Write a recursive function to determine whether a given string is a palindrome.**
**Solution:** A recursive function in Python is to determine whether a given string is a palindrome:
“
```
def is_palindrome(s):
    # base case: if the string is empty or has only one character, it is a palindrome
    if len(s) <= 1:
        return True
    # recursive case: check if the first and last characters are equal
    # if they are, recursively check the substring without these characters
    if s[0] == s[-1]:
        return is_palindrome(s[1:-1])
    # if the first and last characters are not equal, the string is not a palindrome
    return False

example usage of the function
print(is_palindrome("racecar")) # True
print(is_palindrome("hello")) # False
print(is_palindrome("abcba")) # True
```
“
```

This function uses a recursive approach by comparing the first and last characters of the string. If they are equal, the function recursively calls itself on the substring without these characters. If the first and last characters are not equal, the string is not a palindrome and the function returns `False`. The function also includes a base case for strings that are empty or have only one character, which are considered palindromes.

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

Solution: Sure! Here's an implementation of a recursive function to find the greatest common divisor (GCD) of two positive integers:

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

In this implementation, the function `gcd` takes two positive integers ‘a’ and ‘b’ as its parameters.

The base case is when ‘b’ is equal to 0. In this case, we have found the GCD and we simply return ‘a’.

If the base case is not met, we recursively call the `gcd` function with the arguments ‘b’ and ‘a % b’. The expression ‘a % b’ calculates the remainder of the division of ‘a’ by ‘b’.

The function continues this recursive call until it reaches the base case, at which point it returns the GCD.

You can test this function with any two positive integers to find their GCD. For example:

```
print(gcd(12, 8))  # Output: 4
print(gcd(18, 12))  # Output: 6
print(gcd(15, 25))  # Output: 5
```