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

Answer : A **lambda function** in Python is a small, anonymous function defined using the lambda keyword, rather than the traditional def keyword. Lambda functions are often used for short, throwaway operations, especially when a simple function is needed for a short period and defining a full function would be unnecessary.

### Syntax:

python

Copy code

lambda arguments: expression

* **arguments**: A comma-separated list of input parameters (similar to function parameters).
* **expression**: An expression evaluated and returned by the lambda function.

### Example:

python

Copy code

# Regular function

def add(x, y):

return x + y

# Lambda function equivalent

add\_lambda = lambda x, y: x + y

print(add(2, 3)) # Output: 5

print(add\_lambda(2, 3)) # Output: 5

### Key Differences Between Lambda and Regular Functions:

1. **Definition**:
   * Regular functions are defined using def and can have multiple lines of code, including statements, conditionals, and loops.
   * Lambda functions are defined using the lambda keyword and can only consist of a single expression.
2. **Return**:
   * A regular function explicitly uses a return statement to return a value.
   * A lambda function automatically returns the result of its expression.
3. **Readability**:
   * Regular functions are often easier to understand, especially for complex operations.
   * Lambda functions are more compact and are commonly used when you need a simple function for a short period.
4. **Naming**:
   * Regular functions are usually named (e.g., add(x, y)).
   * Lambda functions are anonymous, although they can be assigned to variables.

### Use Cases:

Lambda functions are typically used in situations where you need a simple function temporarily, such as with functions like map(), filter(), and sorted().

Example with sorted():

python

Copy code

# Sorting a list of tuples by the second element

pairs = [(1, 2), (3, 1), (5, 0)]

sorted\_pairs = sorted(pairs, key=lambda pair: pair[1])

print(sorted\_pairs) # Output: [(5, 0), (3, 1), (1, 2)]

In this case, the lambda function is used to define how the sorting should be done without the need for a full function definition.

# Question 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**. You can define and use lambda functions with multiple input parameters just like regular functions. The arguments are separated by commas, and the expression can use these arguments within its body.

### Syntax for Lambda with Multiple Arguments:

python

Copy code

lambda argument1, argument2, ... : expression

### Example with Multiple Arguments:

python

Copy code

# Lambda function with two arguments

multiply = lambda x, y: x \* y

print(multiply(3, 4)) # Output: 12

In this example, the lambda function takes two arguments (x and y) and returns their product.

### Example with More Than Two Arguments:

python

Copy code

# Lambda function with three arguments

sum\_of\_three = lambda a, b, c: a + b + c

print(sum\_of\_three(1, 2, 3)) # Output: 6

### Usage in Functions like map(), filter(), etc.:

You can also pass lambda functions with multiple arguments to higher-order functions like map(), filter(), or sorted().

#### Example with map():

python

Copy code

# List of tuples where each tuple has two elements

pairs = [(1, 2), (3, 4), (5, 6)]

# Using lambda with map to add the elements of each tuple

result = list(map(lambda x, y: x + y, [1, 3, 5], [2, 4, 6]))

print(result) # Output: [3, 7, 11]

Here, the lambda function adds the elements of two lists element-wise, taking two arguments per invocation (x and y).

### Key Points:

* A lambda function can accept any number of arguments, including zero or more.
* It’s often used in contexts where you need a simple, one-line function to be applied quickly without needing to define a full function using def.

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

Lambda functions in Python are typically used in situations where you need a simple, short, and anonymous function. They are particularly useful in higher-order functions like map(), filter(), and sorted(), where a small function is required for a specific task but defining a full function would be overkill.

### Common Use Cases for Lambda Functions:

1. **map()**: Apply a function to all items in an iterable (e.g., a list).
2. **filter()**: Filter elements from an iterable based on a condition.
3. **sorted()**: Sort an iterable using a custom sorting criterion.
4. **Custom operations**: Used in places where a function is needed temporarily, like inside another function.

### Example Use Case: Sorting a List of Tuples

Let's say you have a list of tuples representing students and their grades, and you want to sort the list based on the grades (the second element in each tuple).

#### Without Lambda Function:

python

Copy code

students = [("Alice", 85), ("Bob", 90), ("Charlie", 80)]

# Sorting by grade using a regular function

def get\_grade(student):

return student[1]

students\_sorted = sorted(students, key=get\_grade)

print(students\_sorted) # Output: [('Charlie', 80), ('Alice', 85), ('Bob', 90)]

#### With Lambda Function:

python

Copy code

students = [("Alice", 85), ("Bob", 90), ("Charlie", 80)]

# Sorting by grade using a lambda function

students\_sorted = sorted(students, key=lambda student: student[1])

print(students\_sorted) # Output: [('Charlie', 80), ('Alice', 85), ('Bob', 90)]

In this case, the lambda function is used to define the sorting criterion directly within the sorted() function, making the code more concise and readable.

### Example Use Case: Filtering Even Numbers

You can use a lambda function with filter() to extract elements that satisfy a condition.

python

Copy code

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

# Filtering even numbers using a lambda function

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even\_numbers) # Output: [2, 4, 6]

Here, the lambda function is used to define the condition for filtering: only numbers divisible by 2 (even numbers) are selected.

### Example Use Case: Applying a Transformation to Each Element

Using map() with a lambda function is a common pattern to transform elements in an iterable.

python

Copy code

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

# Squaring each number using a lambda function

squared\_numbers = list(map(lambda x: x\*\*2, numbers))

print(squared\_numbers) # Output: [1, 4, 9, 16, 25]

In this case, the lambda function squares each number in the list.

### Summary of Common Lambda Use Cases:

* **map()**: Apply a transformation to each element of an iterable.
* **filter()**: Select elements from an iterable based on a condition.
* **sorted()**: Sort elements in an iterable based on a custom criterion.
* **Ad-hoc operations**: Use lambda for short, one-off functions in places where defining a full function isn't necessary.

Lambda functions are ideal when you need quick, simple functionality in a concise form without the overhead of defining a full function.

# Question 4. What are the advantages and limitations of lambda functions compared to regular functions in

# Python?

### Advantages of Lambda Functions:

1. **Concise and Compact**:
   * Lambda functions allow you to define simple functions in a single line, making them more concise than regular functions.
   * This is particularly useful for short operations that don’t require a full function definition.

**Example**:

python

Copy code

add = lambda x, y: x + y

1. **Anonymous**:
   * Lambda functions do not require a name, making them useful when you need a function only for a short period or as an argument to another function.
2. **Simplifies Code**:
   * When using functions like map(), filter(), or sorted(), lambda functions allow you to inline the logic, improving readability for small tasks.

**Example**:

python

Copy code

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

squared = list(map(lambda x: x \*\* 2, numbers))

1. **Functional Programming Style**:
   * Lambda functions facilitate a functional programming style where functions are treated as first-class citizens and can be passed around as arguments, returned as values, or assigned to variables.
2. **Quick Temporary Use**:
   * They're ideal for situations where you need a small, throwaway function (e.g., for a one-time calculation or filtering task) without the need for a separate function definition.

### Limitations of Lambda Functions:

1. **Limited to a Single Expression**:
   * Lambda functions can only contain a single expression. This means they cannot include statements, multiple expressions, or complex logic like conditionals, loops, or error handling.

**Example**:

python

Copy code

# Invalid Lambda function because it uses a statement (if-else)

lambda x: if x > 10: print(x)

To do more complex tasks, you'd need to use a regular function.

1. **Less Readable for Complex Operations**:
   * While lambda functions are great for simple tasks, they can quickly become unreadable or overly cryptic if the logic becomes complex. In such cases, a regular function is often preferred for clarity.

**Example**:

python

Copy code

# A more readable alternative to a complex lambda function

def complex\_operation(x, y):

if x > y:

return x \* y

else:

return x + y

1. **No Documentation**:
   * Lambda functions are anonymous and do not have a name or docstring, making it difficult to document or provide detailed explanations for what the function does.
2. **Debugging and Testing**:
   * Since lambda functions don’t have a name, debugging them can be challenging, as stack traces will show the lambda function as an anonymous entity. Also, they can’t be tested in isolation without assigning them to a variable.
3. **Limited Reusability**:
   * Lambda functions are designed for one-off use cases. If you find yourself needing to reuse the same logic multiple times, it’s often better to define a regular function instead.

### When to Use Lambda Functions:

* **Simple, short operations**: For tasks like transformations, filtering, or sorting, where a full function definition is unnecessary.
* **In higher-order functions**: When using functions like map(), filter(), sorted(), etc., lambda provides a succinct way to apply a custom function without defining it explicitly.

### When to Use Regular Functions:

* **Complex logic**: When the function requires multiple expressions, conditionals, loops, or error handling.
* **Reusability**: When the function needs to be reused in multiple places.
* **Readability and clarity**: When clarity and explicit documentation are important for your code.

### Summary of Advantages vs. Limitations:

| **Advantages** | **Limitations** |
| --- | --- |
| Concise and compact | Limited to a single expression |
| Anonymous (no need for a name) | Less readable for complex operations |
| Great for functional programming style | No documentation (no docstring) |
| Simplifies code for small tasks | Harder to debug and test |
| Ideal for one-off use cases | Limited reusability |

In general, lambda functions are a powerful tool for concise, temporary, and simple operations, but for more complex, reusable, and readable logic, regular functions are preferred.

# Question 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**, just like regular functions. This behavior is a result of **lexical scoping** in Python, where the scope of a variable is determined by its location in the source code. A lambda function can "capture" and use variables from its surrounding environment (also known as free variables).

### Explanation:

* When a lambda function is defined, it has access to variables in the scope in which it is created (the enclosing scope), even if those variables are outside the lambda's immediate body.
* This is possible because the lambda function forms a closure, meaning it keeps references to the variables that were available when it was created.

### Example:

python

Copy code

x = 10 # Variable in the outer scope

# Lambda function accessing x from the outer scope

my\_lambda = lambda y: x + y

# Using the lambda function

result = my\_lambda(5)

print(result) # Output: 15

### Explanation of the Example:

* The variable x is defined outside the lambda function.
* The lambda function accesses x and adds it to y, which is the argument passed to the lambda.
* When my\_lambda(5) is called, it calculates 10 + 5 = 15 and returns the result.

### Important Notes:

* Lambda functions capture **the reference** to the variables from the outer scope, not their values at the time the lambda is created. This means that if the variable’s value changes later, the lambda will reflect that new value when called.

**Example:**

python

Copy code

def create\_lambda():

x = 10

return lambda y: x + y

# Create the lambda

my\_lambda = create\_lambda()

# Modify the value of x in the outer scope

x = 20

# Calling the lambda function still uses the latest value of x

result = my\_lambda(5)

print(result) # Output: 25 (because x is now 20)

* The lambda function "remembers" and uses the most recent value of x, even though x was modified after the lambda was created.

### Summary:

* **Yes**, lambda functions can access variables defined in their enclosing scope (outside their own body).
* This is due to lexical scoping and closures in Python.
* The lambda uses the value of those variables as they are at the time the lambda is called, not when it was defined.

Top of Form

Bottom of Form

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

To calculate the square of a given number using a lambda function, you can define it like this:

python

Copy code

square = lambda x: x \*\* 2

### Example Usage:

python

Copy code

# Calling the lambda function with an argument

result = square(5)

print(result) # Output: 25

### Explanation:

* lambda x: x \*\* 2 defines a lambda function that takes one argument x and returns the square of x (i.e., x\*\*2).
* When square(5) is called, it computes 5 \*\* 2, which equals 25, and returns that value.

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

You can use a lambda function in combination with the max() function to find the maximum value in a list of integers.

Here’s how you can define the lambda function:

python

Copy code

max\_value = lambda lst: max(lst)

### Example Usage:

python

Copy code

numbers = [3, 5, 1, 9, 2]

# Calling the lambda function to find the maximum value

result = max\_value(numbers)

print(result) # Output: 9

### Explanation:

* The lambda function lambda lst: max(lst) takes a list lst as an argument and returns the maximum value from the list using the built-in max() function.
* When max\_value(numbers) is called, it computes the maximum value from the list [3, 5, 1, 9, 2], which is 9, and returns that value.

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

You can use a lambda function with the filter() function to filter out all the even numbers from a list of integers. The filter() function allows you to apply a condition (in this case, checking if a number is even) to each element in the list.

### Lambda Function to Filter Even Numbers:

python

Copy code

even\_numbers = lambda lst: list(filter(lambda x: x % 2 == 0, lst))

### Example Usage:

python

Copy code

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

# Calling the lambda function to filter out even numbers

result = even\_numbers(numbers)

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

### Explanation:

* lambda x: x % 2 == 0 is the condition that checks if a number x is even (i.e., divisible by 2).
* filter(lambda x: x % 2 == 0, lst) filters the list lst by applying the condition, keeping only the elements that are even.
* list() converts the filtered result (which is an iterator) into a list so that it can be printed or used.

When even\_numbers(numbers) is called, it returns the list of even numbers from the original list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], which is [2, 4, 6, 8, 10].

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

# string.

You can use a lambda function with the sorted() function to sort a list of strings in ascending order based on the length of each string.

### Lambda Function to Sort Strings by Length:

python

Copy code

sorted\_by\_length = lambda lst: sorted(lst, key=lambda s: len(s))

### Example Usage:

python

Copy code

strings = ["apple", "banana", "kiwi", "cherry", "date"]

# Calling the lambda function to sort strings by length

result = sorted\_by\_length(strings)

print(result) # Output: ['kiwi', 'date', 'apple', 'banana', 'cherry']

### Explanation:

* The sorted() function takes a key argument, which specifies the function to be used for sorting.
* The lambda function lambda s: len(s) returns the length of each string in the list.
* sorted(lst, key=lambda s: len(s)) sorts the list of strings based on their lengths in ascending order.

# Question 10. Create a lambda function that takes two lists as input and returns a new list containing the

# common elements between the two lists.

You can use a lambda function to find the common elements between two lists by utilizing the filter() function along with a condition that checks if an element from the first list is also present in the second list.

### Lambda Function to Find Common Elements:

python

Copy code

common\_elements = lambda list1, list2: list(filter(lambda x: x in list2, list1))

### Example Usage:

python

Copy code

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

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

# Calling the lambda function to find common elements

result = common\_elements(list1, list2)

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

### Explanation:

* The lambda function lambda x: x in list2 checks if an element x from list1 is also present in list2.
* The filter() function applies this lambda function to each element in list1, keeping only the elements that are found in list2.
* list() converts the filtered result (which is an iterator) into a list.

In this case, the common elements between list1 and list2 are [4, 5].

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

To calculate the **factorial** of a given positive integer using recursion, you can define the function as follows:

### Recursive Function for Factorial:

python

Copy code

def factorial(n):

# Base case: factorial of 0 or 1 is 1

if n == 0 or n == 1:

return 1

# Recursive case: n \* factorial of (n-1)

else:

return n \* factorial(n - 1)

### Example Usage:

python

Copy code

number = 5

result = factorial(number)

print(result) # Output: 120

### Explanation:

* **Base case**: The factorial of 0 or 1 is 1, which stops the recursion.
* **Recursive case**: The function calls itself with n-1 and multiplies n by the result of factorial(n-1). This continues until the base case is reached.

For example, to calculate factorial(5):

* factorial(5) = 5 \* factorial(4)
* factorial(4) = 4 \* factorial(3)
* factorial(3) = 3 \* factorial(2)
* factorial(2) = 2 \* factorial(1)
* factorial(1) = 1 (base case)

The final result is 5 \* 4 \* 3 \* 2 \* 1 = 120.

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

Here's the implementation of a recursive function to compute the nth Fibonacci number:

python

Copy code

def fibonacci(n):

if n <= 1:

return n

else:

return fibonacci(n - 1) + fibonacci(n - 2)

# Example usage

n = 10

print(f"The {n}th Fibonacci number is: {fibonacci(n)}")

### Explanation:

* The Fibonacci sequence is defined as:
  + fibonacci(0) = 0
  + fibonacci(1) = 1
  + For n >= 2, fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
* The base case is when n is 0 or 1, which directly return n. For other values, it recursively calls itself to compute the result.

In the example usage, fibonacci(10) will return the 10th Fibonacci number.

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

Here's the implementation of a recursive function to find the sum of all the elements in a given list:

python

Copy code

def sum\_list(lst):

if len(lst) == 0:

return 0

else:

return lst[0] + sum\_list(lst[1:])

# Example usage

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

print(f"The sum of the list is: {sum\_list(lst)}")

### Explanation:

* The base case is when the list is empty (len(lst) == 0), in which case the sum is 0.
* The recursive case takes the first element (lst[0]) and adds it to the sum of the remaining elements (sum\_list(lst[1:])).

For example, for the list [1, 2, 3, 4, 5], the function will return the sum of the numbers, which is 15.

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

Here is the implementation of a recursive function to determine whether a given string is a palindrome:

python

Copy code

def is\_palindrome(s):

# Base case: if the string is empty or has only one character, it's a palindrome

if len(s) <= 1:

return True

# Check if the first and last characters are the same

if s[0] == s[-1]:

# Recursively check the substring excluding the first and last characters

return is\_palindrome(s[1:-1])

else:

return False

# Example usage

s = "racecar"

print(f"Is the string '{s}' a palindrome? {is\_palindrome(s)}")

### Explanation:

* The base case is when the string has a length of 0 or 1, in which case it is automatically a palindrome.
* The recursive case compares the first and last characters (s[0] and s[-1]):
  + If they are the same, it recursively checks the substring excluding those two characters (s[1:-1]).
  + If the characters are different, it returns False immediately.

For example, for the string "racecar", the function will return True because it reads the same forwards and backwards.

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

Here is the implementation of a recursive function to find the greatest common divisor (GCD) of two positive integers using the Euclidean algorithm:

python

Copy code

def gcd(a, b):

# Base case: if one number is zero, return the other number

if b == 0:

return a

else:

# Recursively call gcd with b and the remainder of a divided by b

return gcd(b, a % b)

# Example usage

a = 56

b = 98

print(f"The GCD of {a} and {b} is: {gcd(a, b)}")

### Explanation:

* The base case is when b is 0, in which case the GCD is a (since gcd(a, 0) = a).
* The recursive case applies the Euclidean algorithm: gcd(a, b) = gcd(b, a % b).
  + This continues until b becomes 0, and at that point, a is the GCD.

For example, for the integers 56 and 98, the function will return 14, which is their GCD.