**FSDS MAY BATCH 2022(Python Basics 24)**

**Submitted by: Shubham Tiwari**

Q1. What is the relationship between def statements and lambda expressions ?

Ans: In Python, both **def** statements and **lambda** expressions are used to define functions, but they have some key differences.

**def** statements are used to define named functions in Python. They have a block of statements, a name, and can accept arguments. For example:

def add(a, b):

return a + b

On the other hand, **lambda** expressions are a way to create anonymous functions. They can be defined in a single line, and they don't require a name. For example:

add = lambda a, b: a + b

In this example, the **lambda** expression defines a function that takes two arguments and returns their sum. This function is then assigned to the name **add**.

One key difference between **def** statements and **lambda** expressions is that **def** statements can contain multiple lines of code, whereas **lambda** expressions can only contain a single expression. Another difference is that **def** statements can have default argument values and optional arguments, while **lambda** expressions can only have a single expression that returns a value.

In general, **def** statements are more versatile and can be used to define complex functions, while **lambda** expressions are more concise and are often used for simple, one-off functions.

Q2. What is the benefit of lambda?

Ans: One of the main benefits of using lambda expressions in Python is that they allow for the creation of small, anonymous functions that can be defined on-the-fly without having to go through the process of defining a full named function using the **def** keyword. This can make your code more concise and easier to read, as it allows you to define simple functions in fewer lines of code.

Another benefit of using lambda expressions is that they can be used as arguments for higher-order functions that expect functions as input, such as **map()**, **filter()**, and **sort()**. This allows us to write more expressive and concise code when working with data collections and manipulating data.

Q3. Compare and contrast map, filter, and reduce.

Ans: **map()**, **filter()**, and **reduce()** are three built-in higher-order functions in Python that are used to process and transform data. They are often used together to perform complex data operations.

Here's how they compare and contrast:

1. **map()**: This function takes a function and a sequence as input and applies the function to each element in the sequence, returning a new sequence of the same length. The function is called with each element of the sequence as its argument, and the results are collected into a new sequence.

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

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

list(squared\_numbers)

[1, 4, 9, 16, 25]

In this example, the **map()** function applies the **lambda** function to each element in the **numbers** list, resulting in a new list of squared numbers.

1. **filter()**: This function takes a function and a sequence as input and returns a new sequence that contains only the elements of the original sequence for which the function returns **True**. The function is called with each element of the sequence as its argument.

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

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

list(even\_numbers)

[2, 4]

In this example, the **filter()** function applies the **lambda** function to each element in the **numbers** list and returns only the even numbers.

1. **reduce()**: This function takes a function and a sequence as input and returns a single value that is the result of applying the function to the elements of the sequence in a cumulative way. The function takes two arguments and returns a single value, which is then used as the first argument for the next call of the function, until the sequence is exhausted.

from functools import reduce

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

sum = reduce(lambda x, y: x + y, numbers)

sum

15

In this example, the **reduce()** function applies the **lambda** function cumulatively to the elements of the **numbers** list, resulting in the sum of all the elements.

In summary, **map()**, **filter()**, and **reduce()** are all higher-order functions in Python that operate on sequences, but they have different purposes and output. **map()** transforms each element of a sequence, **filter()** selects certain elements from a sequence based on a condition, and **reduce()** aggregates all elements of a sequence into a single value.

Q4. What are function annotations, and how are they used?

Ans: Function annotations in Python provide a way to add metadata to the parameters and return values of a function. They are optional and do not affect the runtime behavior of the code, but can be used by tools such as linters, IDEs, and documentation generators to provide additional information about the function.

Function annotations are specified by placing a colon **:** after the parameter name, followed by the annotation expression. For example:

def greet(name: str) -> str:

return f"Hello, {name}!"

In this example, the **greet** function takes a parameter **name** of type **str**, and returns a value of type **str**. The annotations **str** indicate that the parameter and return value should be of the string type. These annotations can be accessed using the **\_\_annotations\_\_** attribute of the function.

Q5. What are recursive functions, and how are they used?

Ans: A recursive function is a function that calls itself during its execution. Recursion is a powerful programming technique that can be used to solve problems that can be divided into smaller, similar subproblems. The function continues to call itself until a base case is reached, at which point the recursion stops and the function returns its result.

Recursive functions are often used to simplify the code for problems that can be broken down into smaller subproblems. For example, calculating the factorial of a number is a classic example of a problem that can be solved using recursion.

def factorial(n):

if n == 1:

return 1

else:

return n \* factorial(n-1)

In this example, the **factorial** function takes a positive integer **n** and returns the product of all positive integers less than or equal to **n**. If **n** is 1, then the function returns 1, which is the base case. Otherwise, it multiplies **n** by the factorial of **n-1**, which is calculated by calling the **factorial** function recursively.

Q6. What are some general design guidelines for coding functions?

Ans: Some general design guidelines are:

1. Keep it simple and focused: Functions should do one thing and do it well. Avoid writing functions that are too complex or try to do too many things at once.
2. Use descriptive and concise names: Use names that accurately describe what the function does. Avoid using names that are too general or too specific. Use variable names that are concise and easy to read.
3. Use default values for parameters when possible: Use default values for parameters that are not required, but can be useful in some cases. This can make the function more flexible and easier to use.
4. Avoid global variables: Avoid using global variables within a function. This can make the function less reusable and more difficult to test.
5. Minimize the use of side effects: Avoid functions that have side effects, such as modifying global state or printing output to the console. This can make the function harder to test and more difficult to reason about.
6. Use exception handling: Use exception handling to catch and handle errors that may occur during the execution of the function. This can make the function more robust and reliable.

7 Follow the PEP 8 style guide: Follow the PEP 8 style guide for naming conventions, code layout, and other best practices in Python. This can make the code more readable and easier to maintain.

8. Write docstrings: Write docstrings for functions that explain what the function does, what parameters it takes, and what it returns. This can help other developers understand and use the function.

9. Write tests: Write tests for functions to ensure that they work as expected and to catch regressions. This can make the code more reliable and easier to maintain.

Q7. Name three or more ways that functions can communicate results to a caller.

Ans: There can be ways such as :

1. Return values: Functions can return one or more values using the **return** statement. The caller can use these values in further computations or pass them to other functions.
2. Side effects: Functions can modify global variables or objects passed in as parameters, and the caller can inspect these changes. This approach should be used with caution, as it can make the code harder to test and reason about.
3. Output parameters: Functions can modify variables passed in as parameters, and the caller can inspect these changes. This approach is less common in Python, as it is generally preferred to use return values.
4. Exceptions: Functions can raise exceptions to signal errors or unusual conditions to the caller. The caller can catch these exceptions and handle them appropriately.
5. Callbacks: Functions can receive other functions as parameters and call them with the results. This approach is commonly used in asynchronous programming and event-driven architectures.