# LAMBDA FUNCTIONS IN PYTHON-

In Python, a lambda function is a small anonymous function that can be defined without a name. It is also known as an "anonymous function" or a "function literal." Lambda functions are typically used for simple, one-line operations where creating a full function definition would be unnecessary.

A lambda function can take any number of arguments, but can only have one expression.

```
lambda arguments: expression
```

```
# Defining a lambda function to calculate the square of a number
square = lambda x: x**2

# Using the lambda function
result = square(5)
print(result)  # Output: 25
```

In this example, we define a lambda function called **square** that takes a single argument **x** and returns the square of **x**. The lambda function is then invoked with the argument **5**, resulting in the value **25** being assigned to the variable **result**.

Lambda functions can also take multiple arguments:

```
# Defining a lambda function to calculate the product of two numbers
multiply = lambda x, y: x * y

# Using the lambda function
result = multiply(3, 4)
print(result)  # Output: 12
```

Lambda functions can be used in various ways, such as:

- As an argument to a higher-order function like **map()**, **filter()**, or **sort()**.
- In situations where you need a short-lived function and don't want to define a separate named function.
- When working with functional programming concepts like closures or currying.

However, it's important to note that lambda functions are limited in their functionality. They are designed for simple expressions and cannot contain multiple statements or complex logic. If you find that your function requires more complexity or additional statements, it's advisable to use a regular named function instead.

**EXAMPLES :**

Add 15 to argument `a`, and return the result:

```
x = lambda a : a + 15
print(x(5))
```

Multiply argument `a` with argument `b` and return the result:

```
x = lambda a, b : a * b
print(x(12, 9))
```

Summarize argument `a`, `b`, and `c` and return the result:

```
x = lambda a, b, c : a + b + c
print(x(1, 3, 7))
```

**WHY DO WE USE LAMBDA FUNCTIONS?**

1. Concise and Readable Code: Lambda functions allow you to write shorter and more concise code compared to using a regular named function. This can make your code more readable and expressive, especially when dealing with simple operations.
2. Anonymous Functions: Lambda functions are anonymous, meaning they don't require a formal name. This is useful when you have a small piece of functionality that doesn't need to be reused elsewhere in your code. You can define the function inline without cluttering your code with unnecessary function definitions.
3. Functional Programming: Lambda functions are commonly used in functional programming paradigms. They enable you to pass functions as arguments to other functions, which allows for higher-order functions like **`map()`**, **`filter()`**, and **`reduce()`**. This functional style can make your code more modular and flexible.
4. Callback Functions: Lambda functions can be used as callback functions, allowing you to define a function on the fly and pass it as an argument to another function. This is often seen in event-driven programming or asynchronous programming, where you want to define a function that will be executed when a certain event occurs.

5. Expressive One-Liners: Lambda functions are ideal for writing quick and concise one-liners, especially for simple transformations or operations. They can be handy for tasks like sorting, filtering, or transforming data, where you don't want to define a full function separately.
6. Encapsulation: Lambda functions encapsulate small pieces of functionality within a single expression. They help to keep the code focused and self-contained, as the logic is contained within the lambda function itself.

**EXAMPLE :**

```
def myfunc(n):
return lambda a : a * n
mydoubler = myfunc(2)
print(mydoubler(11))
```

The code you provided demonstrates the use of a higher-order lambda function. Here's a breakdown of what it does:

1. The **myfunc** function takes a parameter **n**.
2. Inside **myfunc**, it defines and returns a lambda function that takes a parameter **a**.
3. The lambda function multiplies **a** by **n** and returns the result.
4. The returned lambda function is assigned to the variable **mydoubler**.
5. Finally, **mydoubler** is invoked with the argument **11**, and the result is printed.

Let's go through the code step by step:

pythonCopy code

```
def myfunc(n):
    return lambda a: a * n

mydoubler = myfunc(2)
```

In this code, **myfunc** is defined as a function that takes a parameter **n**. It returns a lambda function that multiplies its argument **a** by **n**. When **myfunc(2)** is called, it returns the lambda function, effectively creating a new function that multiplies its argument by 2.

pythonCopy code

```
print(mydoubler(11))
```

In this line, **mydoubler** is invoked as a function with the argument **11**. Since **mydoubler** is the lambda function created by **myfunc(2)**, it multiplies **11** by **2**, resulting in the output **22**. This value is then printed.

So, the output of the code will be- 22

The **mydoubler** function created through **myfunc(2)** acts as a "doubler" function that can be reused to multiply any number by 2. Similarly, you could create other functions with different values of **n** using **myfunc**, allowing for code reusability and flexibility.

---

**EXAMPLE :**

```
def myfunc(n):
  return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```