Lambda Function - MAP, REDUCE, FILTER

💡 Use lambda functions when an anonymous function is required for a short period of time.

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.

  ```
  x = lambda a : a + 10
  print(x(5)) #15
  ```

- Lambda functions can take any number of arguments:

  ```
  x = lambda a, b : a * b
  print(x(5, 6)) # 30
  ```

  ```
  x = lambda a, b, c : a + b + c
  print(x(5, 6, 2))
  ```

Difference in normal and lambda function

1. Lambda has no return value, instead it return function

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

   mydoubler = myfunc(2)

   print(mydoubler(11)) # 22
   ```

   Here lambda is returning a function `a*2` then the value of a is given by `mydoubler`. Then your answer has came 22.

2. One line
3. Not used for code reusability
4. No name

Why use Lambda?

1. Along with Higher order functions.
   - **a higher-order function is a function that can take other functions as input or return functions as output.** They provide flexibility and reusability in your code.

```
def apply_operation(numbers, operation):

    results = []
    for num in numbers:
        results.append(operation(num))
    return results

# Example usage:
numbers = [1, 2, 3, 4, 5]

# Square each number
squared_numbers = apply_operation(numbers, lambda x: x**2)
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]

# Double each number
doubled_numbers = apply_operation(numbers, lambda x: x * 2)
print(doubled_numbers)  # Output: [2, 4, 6, 8, 10]

# Calculate the factorial of each number (recursive)
def factorial(x):
    if x == 0:
        return 1
    else:
        return x * factorial(x - 1)

factorial_numbers = apply_operation(numbers, factorial)
print(factorial_numbers)  # Output: [1, 2, 6, 24, 120]
```

Map

- It takes a function and an iterable as input.
- It applies the function to each element of the iterable.
- It returns a new iterable with the results.

- `map(func, *iterables)` → map object

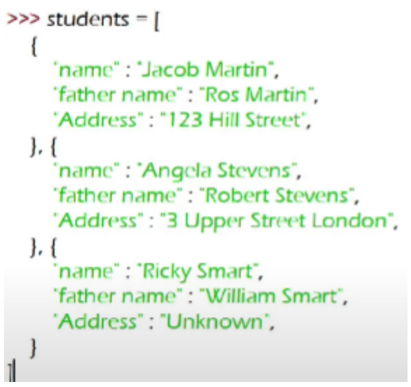
```
L = [1,2,3,4] # double every item in list

list(map(lambda x: x*2, L)) # [2,4,6,8]
```

why we use `list` before map

- `map` **returns an iterator:** When you apply the `map` function, it returns an iterator object
    - An iterator object is a special type of object in Python that provides a way to iterate over a sequence of values.
- To access and manipulate the individual elements of the result, we need to convert the iterator into a list

```
numbers = [1, 2, 3]
squared_numbers = map(lambda x: x**2, numbers)
print(squared_numbers)  # Output: <map object at 0x7f7d05730290>

# Converting to a list
squared_numbers_list = list(squared_numbers)
print(squared_numbers_list)  # Output: [1, 4, 9]
```

Eg.



```
>>> students = [
    {
      "name" : "Jacob Martin",
      "father name" : "Ros Martin",
      "Address" : "123 Hill Street",
    }, {
      "name" : "Angela Stevens",
      "father name" : "Robert Stevens",
      "Address" : "3 Upper Street London",
    }, {
      "name" : "Ricky Smart",
      "father name" : "William Smart",
      "Address" : "Unknown",
    }
]
```

```
# i want to fetch all the name from this list of dictionary.

list(map(lambda student: student['name'], students))
# ['Jacob Martin', 'Angela Stevens', 'Ricky Smart']
```

Filter Function

It is used to create a new iterable (like a list, tuple, or set) containing elements from an existing iterable that meet a specific condition. This condition is defined by a function that you provide as an argument to `filter()`.

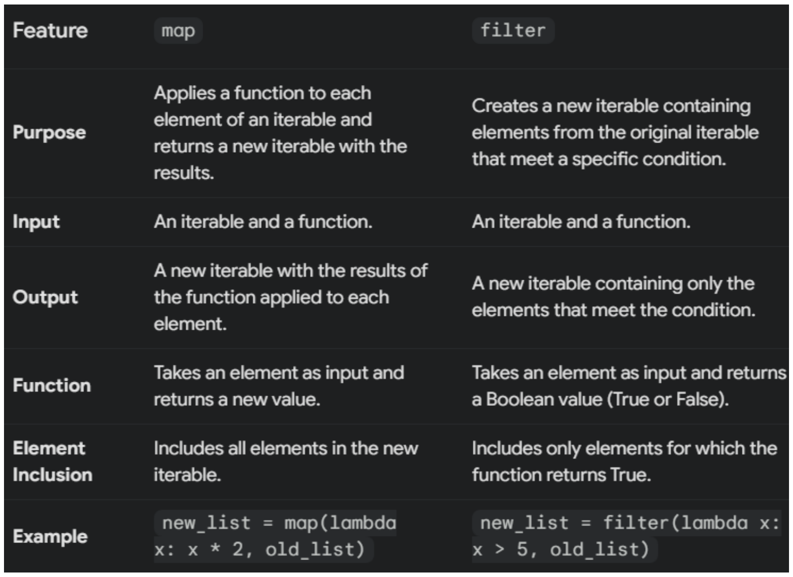
```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def is_even(x):
    return x % 2 == 0

even_numbers = filter(is_even, numbers)
print(list(even_numbers))  # Output: [2, 4, 6, 8, 10]

# if i use map function
even_numbers = map(is_even, numbers)
print(list(even_numbers)
# [False, True, False, True, False, True, False, True, False, True]
```

```


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

```
# # Output: [2, 4, 6, 8, 10]
```

```
fruits = ['apple', 'orange', 'mango', 'guava']

list(filter(lambda fruit: 'e' in fruit, fruits))
# ['Apple', 'Orange']
```

Map vs Filter



Feature	`map`	`filter`
Purpose	Applies a function to each element of an iterable and returns a new iterable with the results.	Creates a new iterable containing elements from the original iterable that meet a specific condition.
Input	An iterable and a function.	An iterable and a function.
Output	A new iterable with the results of the function applied to each element.	A new iterable containing only the elements that meet the condition.
Function	Takes an element as input and returns a new value.	Takes an element as input and returns a Boolean value (True or False).
Element Inclusion	Includes all elements in the new iterable.	Includes only elements for which the function returns True.
Example	`new_list = map(lambda x: x * 2, old_list)`	`new_list = filter(lambda x: x > 5, old_list)`

Reduce

- The `reduce()` function in Python is a powerful tool for combining elements of an iterable into a single value.
- It takes a function and an iterable as input and applies the function to the first two elements of the iterable. The result is then used as the first argument for the next application of the function with the third element of the iterable
- This process continues until all elements have been processed.

```
from functools import reduce

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

def add(x, y):
    return x + y

result = reduce(add, numbers)
print(result)
  # Output: 15
```