😊 Lambda Expressions, Map, and Filter

Lambda Expressions

A **lambda expression** (or **lambda function**) in Python is a small, anonymous function that can have any number of arguments but only one expression. Unlike regular functions, lambda functions are typically used for short, simple operations where defining a full function is unnecessary. They are mostly used in situations requiring a quick function definition for immediate use.

Map Function

The **map function** is used to apply a given function to all items in an iterable (such as a list or tuple) and returns a new iterable with the results. It helps in transforming data efficiently without using explicit loops. The key benefit of map is that it allows operations on large datasets in a concise way.

Filter Function

The **filter function** is used to filter elements from an iterable based on a given condition. It takes a function that returns either True or False and applies it to each element in the iterable. Only the elements that satisfy the condition (where the function returns True) are included in the final output.

These three concepts are commonly used for functional programming and data manipulation in Python.

To begin with the Lab

1. A function square(num) is defined to return the square of a given number. A list has been created and the **map()** function is used to apply the square function to each element of my_nums. However, calling the **map(square, my_nums)** alone does not display the results immediately; instead, it returns a map object.
2. To retrieve the values, the result of **map()** is converted into a list using **list(map(square, my_nums))**, which produces [1, 4, 9, 16, 25].
3. The **map()** function is useful for applying a function to every item in an iterable without using explicit loops.

```
[1]: def square(num):
         return num**2

[2]: my_nums = [1,2,3,4,5]

[5]: map(square,my_nums)

[5]: <map at 0x205baec21d0>

[7]: # To get the results, either iterate through map()
     # or just cast to a list
     list(map(square,my_nums))

[7]: [1, 4, 9, 16, 25]
```

4. The function can be more complex. The **splicer()** function is defined to check the length of a given string: If the length is even, it returns the string 'even'. If the length is odd, it returns the first character of the string.
5. This example shows how **map()** applies a function to each element in an iterable and processes the data efficiently without explicit loops.

```
[8]: def splicer(mystring):
         if len(mystring) % 2 == 0:
             return 'even'
         else:
             return mystring[0]

[9]: mynames = ['John','Cindy','Sarah','Kelly','Mike']

[10]: list(map(splicer,mynames))

[10]: ['even', 'C', 'S', 'K', 'even']
```

6. Now we are going to look at the **filter function**. The filter function returns an iterator yielding those items of the iterable for which the function(item) is true. Meaning you need to filter by a function that returns either True or False.
7. The **filter()** function is used to select elements from an iterable based on a condition.
8. Unlike **map()**, which applies a function to all elements, **filter()** only keeps elements that meet a condition.
9. The result of **filter()** is an iterator, so it needs to be converted to a list if you want to see the filtered values directly.

```
[12]: def check_even(num):
          return num % 2 == 0

[13]: nums = [0,1,2,3,4,5,6,7,8,9,10]

[15]: filter(check_even,nums)

[15]: <filter at 0x205baed4710>

[16]: list(filter(check_even,nums))

[16]: [0, 2, 4, 6, 8, 10]
```

10. One of Pythons most useful (and for beginners, confusing) tools is the Lambda expression. Lambda expressions allow us to create "anonymous" functions. This basically means we can quickly make ad-hoc functions without needing to properly define a function using **def**.
11. Function objects returned by running lambda expressions work exactly the same as those created and assigned by **defs**. There is a key difference that makes lambda useful in specialized roles.
12. The function computes the square of a given number. It first stores the result in a variable (result = num**2) and then returns it. The function is rewritten in a more concise way by directly returning num**2, removing the need for an extra variable.

[17]:
```
def square(num):
    result = num**2
    return result
```

[18]:
```
square(2)
```

[18]:
```
4
```

We could simplify it:

[19]:
```
def square(num):
    return num**2
```

[20]:
```
square(2)
```

[20]:
```
4
```

We could actually even write this all on one line.

[21]:
```
def square(num): return num**2
```

[22]:
```
square(2)
```

[22]:
```
4
```

13. This is the form a function that a lambda expression intends to replicate. A lambda expression can then be written as.

[23]:
```
lambda num: num ** 2
```

[23]:
```
<function __main__.<lambda>>
```

[25]:
```
# You wouldn't usually assign a name to a lambda expression, this is just for demonstration!
square = lambda num: num **2
```

•••

[26]:
```
square(2)
```

[26]:
```
4
```

14. This code demonstrates the use of **lambda functions** along with the **map()** and **filter()** functions in Python.

```
[29]: list(map(lambda num: num ** 2, my_nums))

[29]: [1, 4, 9, 16, 25]

[30]: list(filter(lambda n: n % 2 == 0,nums))

[30]: [0, 2, 4, 6, 8, 10]
```

15. This time we are demonstrating the different lambda expressions in Python. Lambda expressions are small, anonymous functions used for simple operations. They are often used in functional programming with functions like map(), filter(), and sorted().
16. The code lambda s: s[0] defines a function that takes a string s and returns its first character. lambda s: s[::-1] takes a string s and returns it reversed. [::-1] is Python slicing notation that reverses a string. lambda x, y: x + y defines a function that takes two numbers and returns their sum. This shows that lambda functions can accept multiple arguments.

```
[31]: lambda s: s[0]

[31]: <function __main__.<lambda>>
```

** Lambda expression for reversing a string: **

```
[32]: lambda s: s[::-1]

[32]: <function __main__.<lambda>>
```

You can even pass in multiple arguments into a lambda expression. Again, keep in mind that not every function can be translated into a lambda expression.

```
[34]: lambda x,y : x + y

[34]: <function __main__.<lambda>>
```