MODULE 02 - 028: Python - Dynamic Reducer using `operator` & `reduce`

This guided exercise introduces **functional programming** in Python using `operator` and `reduce()`

- `**operator**` allows us to dynamically execute mathematical functions.
- `**reduce()**` applies an operation to an entire list, making it an elegant way to implement a reducer.

It also introduces function expressions in Python using `lambda`functions:

The goal of this guide is to create a more flexible reduce() function.

Understanding the Dynamic Reducer

In this guide, we will build a **dynamic reducer** using Python's `operator` and `reduce()`. If you haven't worked with reducers before, this will be a great opportunity to learn about them while reinforcing your knowledge of functional programming.

A **reducer** takes a collection of values and a mathematical operation, then applies that operation across all elements in the collection.

Key Concepts Covered:

- Python's `operator` module.
- Using `reduce()` from `functools`.
- Lambda functions.
- Function lookups using dictionaries.

---

1 How Does the Dynamic Reducer Work?

The `dynamic_reducer()` function:

- Takes in **a list of numbers**.
- Accepts an **operator as a string** (e.g., `"+", "-", "*", "/"`).
- Performs the specified operation on the list using `reduce()`.

Example Usage:

```
from functools import reduce
import operator

def dynamic_reducer(collection, op):
    operators = {
        "+": operator.add,
        "-": operator.sub,
        "*": operator.mul,
        "/": operator.truediv
    }
    return reduce((lambda total, element: operators[op](total, element)), collection)

# Test cases
print(dynamic_reducer([1, 2, 3], "+"))  # Output: 6
print(dynamic_reducer([1, 2, 3], "-"))  # Output: -4
print(dynamic_reducer([1, 2, 3], "*"))  # Output: 6
print(dynamic_reducer([1, 2, 3], "/"))  # Output: 0.1666...
```

Best Practice: Use dictionaries to map string operators to Python functions, ensuring flexibility and clean code.

---

2 Breaking Down the Solution

Creating an Operator Lookup Table

We use a dictionary to map operator strings (`"+", "-", "*", "/"`) to their corresponding functions:

```
operators = {
    "+": operator.add,
    "-": operator.sub,
    "*": operator.mul,
    "/": operator.truediv
}
```

Using `reduce()` to Apply Operations

The `reduce()` function applies an operation to all elements in a list, accumulating results.

```
reduce(lambda total, element: operators[op](total, element), collection)
```

This line dynamically calls the correct operator function using:

```
operators[op](total, element)
```

Why Use `reduce()`?

- Simplifies looping through collections.
- Eliminates the need for manual accumulation.
- Works well with functional programming.

Note: If you prefer a manual approach, you can iterate using a loop:

```
total = collection[0]
for num in collection[1:]:
    total = operators[op](total, num)
print(total)
```

---

3 Understanding Lambda Functions in `reduce()`

The lambda function inside `reduce()`:

```
lambda total, element: operators[op](total, element)
```

- `total`: The accumulated result.
- `element`: The current list item.
- `operators[op](total, element)`: Dynamically applies the correct operation.

For example, given `[1, 2, 3]` and `op = "+"`, `reduce()` will perform:

1. `1 + 2 = 3`
2. `3 + 3 = 6`

Best Practice: Use lambda functions when passing operations dynamically.

---

4 Handling Edge Cases

1. **Empty List Handling**

   ```
   if not collection:
      return 0  # Default value for an empty list
   ```

2. **Invalid Operator Handling**

   ```
   if op not in operators:
      raise ValueError("Invalid operator")
   ```

3. **Division by Zero**

```
    if op == "/" and 0 in collection[1:]:
        raise ZeroDivisionError("Cannot divide by zero")
```

Best Practice: Always include error handling to prevent crashes.

Summary: Key Takeaways

Concept	Explanation
`reduce()`	Applies an operation across all elements in a list.
Operator Lookup	A dictionary maps symbols (`"+"`, `"-"`, etc.) to functions.
Lambda Functions	Used inside `reduce()` to dynamically apply operations.
Error Handling	Ensures safe execution (e.g., avoids division by zero).

Python Documentation Reference

functools.reduce()

Reduce a list to a single value using a function.

operator module

Provides standard operator functions like `add`, `sub`, `mul`, and `truediv`.

Lambda Functions

Lambda functions, also known as **anonymous functions**, are small, inline functions that can be defined without a name. They are typically used for short, simple operations.

```
lambda arguments: expression
```

- **No Name**: Lambda functions are anonymous, meaning they don't require a formal function definition.
- **Single Expression**: They can only contain a single expression, which is evaluated and returned.
- **Inline Use**: Lambdas are often used where a function is required for a short period, such as in `map()`, `filter()`, or `reduce()`.

Examples:

```
# Regular function
def add(x, y):
    return x + y

# Equivalent lambda function
add_lambda = lambda x, y: x + y

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

Use Case:

Lambda functions are ideal for short, throwaway functions. For example, sorting a list of tuples by the second element:

```
data = [(1, 3), (4, 1), (2, 2)]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)  # Output: [(4, 1), (2, 2), (1, 3)]
```

`reduce()` function

The `reduce()` function is part of the `functools` module and is used to **cumulatively apply a function to all elements in a list**, reducing the list to a single value.

Syntax:

```
from functools import reduce
reduce(function, iterable, initializer=None)
```

- **Cumulative Operation**: `reduce()` applies the function to the first two elements, then to the result and the next element, and so on.
- **Single Output**: It reduces the entire list to a single value.
- **Initializer**: An optional initial value can be provided to start the reduction.

Example:

```
from functools import reduce

# Sum of a list using reduce
numbers = [1, 2, 3, 4]
sum_result = reduce(lambda x, y: x + y, numbers)
print(sum_result)  # Output: 10
```

How It Works:

1. **First Iteration**: `1 + 2 = 3`
2. **Second Iteration**: `3 + 3 = 6`
3. **Third Iteration**: `6 + 4 = 10`

The final result is `10`.

Use Case:

`reduce()` is useful for operations that need to be applied across an entire list, such as summing, multiplying, or finding the maximum value.

```
# Product of a list using reduce
product_result = reduce(lambda x, y: x * y, numbers)
print(product_result)  # Output: 24
```

Combining `lambda` and `reduce()`

Lambda functions and `reduce()` are often used together to create concise, dynamic operations on lists.

```
from functools import reduce

# Dynamic reducer using lambda and reduce
def dynamic_reducer(collection, op):
    operators = {
        "+": lambda x, y: x + y,
        "-": lambda x, y: x - y,
        "*": lambda x, y: x * y,
        "/": lambda x, y: x / y
    }
    return reduce(operators[op], collection)

# Test cases
```

```
print(dynamic_reducer([1, 2, 3], '+'))  # Output: 6
print(dynamic_reducer([1, 2, 3], '*'))  # Output: 6
```

Why This Works:

- **Dynamic Lookup**: The `operators` dictionary maps strings (like `"+"`) to lambda functions.
- **Flexibility**: By using `reduce()`, we can apply the operation to the entire list without writing repetitive code.

Visual summary:

```
# Visual Summary

## Lambda Functions
- **Syntax**: `lambda x, y: x + y`
- **Use Case**: Short, throwaway functions.
- **Example**: Sorting, filtering, mapping.

## `reduce()`
- **Syntax**: `reduce(lambda x, y: x + y, [1, 2, 3])`
- **Use Case**: Cumulative operations on lists.
- **Example**: Summing, multiplying, finding max.

## Combined Example
```python
from functools import reduce

def dynamic_reducer(collection, op):
    operators = {
        "+": lambda x, y: x + y,
        "-": lambda x, y: x - y,
        "*": lambda x, y: x * y,
        "/": lambda x, y: x / y
    }
    return reduce(operators[op], collection)
```

Video lesson speech

I hope that you are in the mood for a challenging Python exercise because we are going to build a dynamic reducer today.

And if you have never worked with any of these types of skills that means that you are going to have a little bit of research ahead of you if you want to build out the solution to this exercise and really learn the concepts.

So if you've never even heard of reducers I'm going to walk through exactly what's expected in this exercise.

I'm going to start off with a comment and let's first just talk about the inputs and outputs because any time that I have something challenging that I'm going to work on.

I boil it down to the most simple kind of explanation.

I can think of and that is knowing what I'm going to put into a function and then what I'm going to get out of it.

Starting the def.

Let's say that we create a function here called dynamic_reducer.

What I want is the ability to pass in two arguments.
```

1. The first is going to be a list that has to be a list of numbers.
   So it might be something like `1`, `2` and `3`.

2. The second one argument is going to be an operator and it's going to be an operator passed in as a string.

So here it might be the plus sign and then what I would expect to be returned here is the sum of each one of those values.

So, in this case, it would be six.

Now if I passed in a different operator so I pass in all of those values and then I use the minus sign then I would expect something else.

I would expect whatever one minus two minus three would be.

So I'd expect some kind of negative number, and then I also want to have this for multiplication and I want to have this for division.

So this function needs to be pretty flexible it needs to be able to take in each one of the main mathematical operators and then it should run through and then keep track and tally up each one of the elements so it needs to perform computation and then it should return whatever the value is.

So the easiest base case example is how you can use this with a plus sign so it'd be a sum and in this case, with the values 1 2 3 even with multiplication the value here would be 6.

But as soon as you start adding other values then it's going to be a little different.

So I'm going to give you some hints on where to research because this is going to combine a number of different technologies and skill sets and libraries that you are going to use.

So these are all going to be libraries within python but you are going to have to import some things. And so we're going to need to import the operator library and then from functools we're going to need to import the Reduce library and so that is what I'm going to give you. So what we need to do once again is build out a function called Dynamic reducer. It takes in a list and then an operator as a string and then it should perform computation on that list.

If you are still a little bit unsure I'm purposely leaving this a little bit vague because I want you to perform some research on these two libraries. First the operator library in python and then in functools the Reduce function. So this is going to be something incredibly helpful it's a part of functional programming where you can pass in a function and then have that function run over an entire collection.

So, in this case, imagine a function like reduce at its most simple kind of implementation reduce is usually used for summing up values and so as you can see right here we have a list and reduce can iterate through this list and tally up each one of the elements. We need it to be a little bit more flexible that's a reason why we're calling this dynamic reduce because you need to be able to pass in each one of these four operators.

So I hope that you have some fun researching those libraries. Both of these are very powerful ways of being able to work with python and be able to implement functional programming and create a pretty dynamic program. So I recommend that you right now pause the video you go and you try to build this out yourself and then come back and watch my own personal solution.

Welcome back, if you built that out congratulations this is definitely something that is on the non-trivial side. And so I highly recommend you if you are able to successfully build this out and if you were not able to do not worry we are going to walk through the solution. And I'll also explain why I chose this specific solution when I personally built this out.

I'm going to start off by creating a function so I'm going to create a dynamic reducer function here. It's going to take in a collection and then an operator. Now I'm purposefully calling this op because if you did reference the documentation for the operator library then if I called this operator then we would have a naming collision. And so we do not want to do that and I don't really feel like aliasing that import either. So I'm going to call it operator and then the argument op.

Now inside of here what I'm going to do is I'm going to create a dictionary. So this is how I'm going to be able to perform the lookup and say that a plus sign or a minus sign or anything like that is passed in I can treat it like a traditional dictionary and then I can use the operator library to actually call the function and so I'm going to create a dictionary here called operators and I'm going to use curly braces inside of it. And now I'm going to have four key-value pairs, the first is going to be the plus sign and this is just a standard dictionary. But what I'm going to use is the value here is calling the operator library and then calling the add a function inside of it.

```
def dynamic_reducer(collection, op):
    operators = {
        "+": operator.add,
```

IMG

Figure 1: IMG

So if that looks a little weird to you than what I was able to do is go into the operator library in python. And if you look through their documentation or even if you look through the sourcecode what you would find is it has a number of methods inside of it. One of them is Add, and that is how we're able to add values. Now let me just multiply this out a few more times. So the next one's going to be Subtract and we're going to have multiplying and then we're going to have divide.

Now inside of here add is already in place that is the method name the next one is sub and the next ones multiply but just mul. And then last one if you're using this with Python 3 it is a slightly different method name than it used to be back in Python 2 6. You were able to just call div but now you have to call truediv. And that is our full lookup table so that is our dictionary. Let me just kind of move and rearrange this just looks a little bit better.

```
def dynamic_reducer(collection, op):
    operators = {
        "+": operator.add,
        "-": operator.sub,
        "*": operator.mul,
        "/": operator.truediv
}
```

Okay so we have a python dictionary now this is just a standard dictionary and like you would build with any kind of python program. And what we have is a set of key-value pairs and each one of the keys is a string and then the value is actually a function. So this is where it's a little bit different. So every time that we call a string and we look up the plus sign this is going to return the function of add, and when we pass in the minus sign it's going to return the function of subtract. So that is how I'm going to perform that Look-Up.

So now that we have the ability to perform a lookup with our operators now we can use the reduced library. So I'm going to say return and then let's call the reduce function. Now if you looked up the reduced function then you know that it takes in two arguments. The first argument it takes in is a function that takes in a lambda function specifically and so I'm going to pass in a lambda and this lambda is going to have a total and then an element. And then what I'm going to do here is I'm going to say operators. So this is if you look on line 12 this is our dictionary name and then with the operators I'm going to use just regular dictionary look up syntax and pass in the key which is op. Now, this op references the second argument that we passed into the function.

So I'm going to simply pass that in and then the way that this works because remember when if this is a little tricky for you. I recommend that you watch my solution a few times because what I'm doing here is if you remember when we perform this Look-Up remember what gets sent back to us, a function does, so because a function gets sent back to us that means that we can call it directly and then simply pass in what it expects.

If you went through the operator documentation then you know that add, subtract, multiply, and truediv all expect two arguments. And so they expect the first one and in our case, it's going to be total and then whatever the element is and that is it.

```
return reduce((lambda total, element: operators[op](total, element))
```

This is essentially the same as regular Python.

To illustrate, when we use `operators[op]` and `op` happens to be `"+"`, it's equivalent to saying `2 + 2`. If `total` is set to `2` and `element` is also `2`, that's exactly what we are doing. The syntax might look a bit strange, but that's because we are calling the actual operator functions directly, making the approach dynamic.

If you didn't want to build the function this way, you could have used conditionals. This is not necessarily the best practice, but you could have done something like:

```
if op == "+":
    return total + element
elif op == "-":
    return total - element
elif op == "*":
    return total * element
elif op == "/":
    return total / element
```

IMG

Figure 2: IMG

IMG

Figure 3: IMG

Let's see what the bug is here. No, it's actually it's a bug with line 19. It would appear I need to close off. Notice how I have that parenthesis I didn't close off?

So now let's run this again and there you go that worked perfectly.

We have 6 for addition, -4 for subtraction, 6 for multiplication, and then we have that 0.1666 number for division. So this is working perfectly and if we passed in any other kind of value. So let's make this one 250 and then for multiplication lets make this one 55 and then for our division here will make this one 100 just so we can have some different values to look at.

and if I run this one now you can see that it has dynamically adjusted our sums working, our subtraction is still working, multiplication is working perfectly, and then so is our division.

We did quite a bit here, so let's review to ensure everything is clear. As mentioned earlier, this is not a trivial exercise. You had to combine multiple Python libraries, work with lambda functions, implement a dictionary, and perform lookups for a function stored within a dictionary. If you've never done this before, it can be quite challenging.

- We used the **operator** library in Python, which allowed us to create a dictionary of operators.
- This made it possible to look up an operator dynamically and apply functions like `add`, `sub`, `mul`, and `truediv`.
- Additionally, we imported the **reduce** function from the `functools` module.

Why Use `reduce()`?

`reduce()` is a functional programming component in Python. It takes a function and applies it across all elements in a list. Without `reduce()`, we would have needed to manually iterate over the collection using a loop, accumulating values along the way. That would have resulted in a more verbose solution.

A manual approach might look like this:

```
total = 0
for element in collection:
    total += element
```

This approach works, but it is not as efficient or elegant as using `reduce()`.

Understanding Lambda Functions in `reduce()`

When working with `reduce()`, it expects a lambda function with two arguments:

```
lambda total, element: total + element
```

- `total`: The accumulated result.
- `element`: The current item in the list.

Each iteration updates `total` using the specified operation. If the list is `[1, 2, 3]` and the operation is addition, it proceeds as follows:

1. `1 + 2 = 3`
2. `3 + 3 = 6`

The final result is `6`.

IMG

Figure 4: IMG

IMG

Figure 5: IMG

Naming Variables in `reduce()`

There are no special requirements for variable names in a lambda function. We could use `x` and `y` instead of `total` and `element`, but descriptive names improve readability. This is why using `total` makes sense when revisiting the code.

Final Thoughts

When working with lambda functions, arguments are passed directly after `lambda`, followed by the operation:

```
lambda x, y: x * y
```

In our case, we use a dictionary lookup to dynamically select the operation:

```
operators[op](total, element)
```

This makes the function highly flexible.

If anything in this explanation is unclear, reach out for clarification. We covered several important Python concepts, including **functional programming, dictionary lookups, and lambda functions**. Take the time to review until everything makes sense, then move on to the next challenge.

Resources

- Exercise Source Code Solution