

Lab 3 Functional Programming (Section 1) - Report

Students:

1. Nguyen Huynh Thao My - ITCSIU21204
 2. Pham Duc Dat - ITITIU20184
-

1 Overview

Lab 03 focuses on the utilization of higher-order functions in Python, a crucial aspect of functional programming. The lab introduces the concepts of lambda functions, map, filter, and reduce, providing a foundational understanding necessary for applying functional programming techniques.

2 Results

In this section, you:

- Implemented both traditional and higher-order function methods to generate the square and cube of each element in an array, filter the squared results within a specified range, and identify even numbers in the array.
- Calculated the Euclidean distance between two points using a lambda function.
- Used higher-order functions to extract names of specific animals from a dictionary.
- Manipulated an array of characters with higher-order functions to sum elements from left to right and right to left and concatenate characters with additional elements.

2.1 Exercise 1

Description of Exercise 1: Given an array $A = [1, 2, 3, 4, 5, 6, 7, 8]$. Write code in two ways, traditional way and higher-order functions to:

- (a) Generate the square of each element in A.
- (b) Generate the power of 3 of each element in A.
- (c) Return the resultant square in the range of [20, 40].
- (d) Generate the elements in A which are even numbers.

2.1.1 Example for (a):

Input:

```
A = [1, 2, 3, 4, 5, 6, 7, 8]
```

Output:

(a) Generate the square of each element in A.

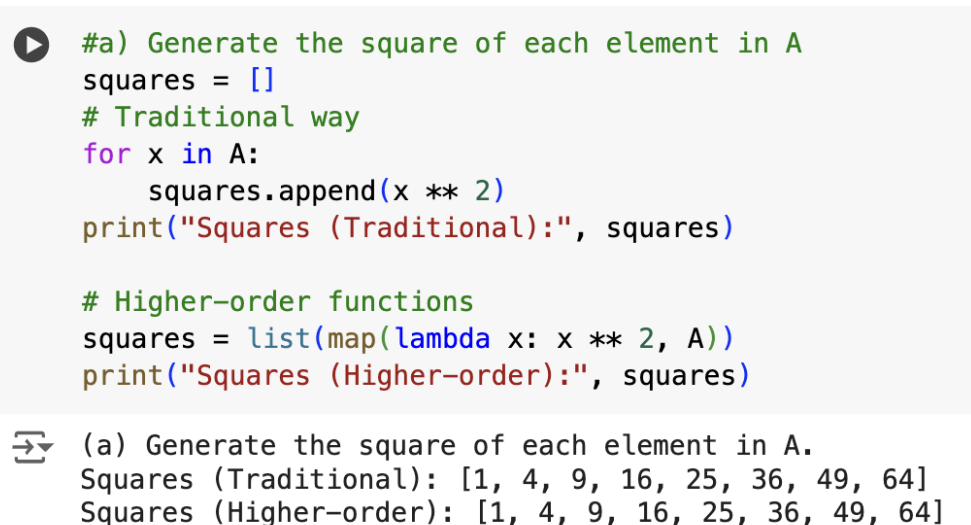
```
Squares (Traditional): [1, 4, 9, 16, 25, 36, 49, 64]
```

```
Squares (Higher-order): [1, 4, 9, 16, 25, 36, 49, 64]
```

Explanation: The code generates the square of each element in array A using both traditional and higher-order functions.

Traditional way: Uses a for-loop to iterate through each element in A, calculates its square, and appends the result to a list.

Higher-order functions: Uses the 'map' function combined with a 'lambda' function to achieve the same result in a more concise manner.



```
#a) Generate the square of each element in A
squares = []
# Traditional way
for x in A:
    squares.append(x ** 2)
print("Squares (Traditional):", squares)

# Higher-order functions
squares = list(map(lambda x: x ** 2, A))
print("Squares (Higher-order):", squares)
```

(a) Generate the square of each element in A.
 Squares (Traditional): [1, 4, 9, 16, 25, 36, 49, 64]
 Squares (Higher-order): [1, 4, 9, 16, 25, 36, 49, 64]

Figure 1: Example code and output for generating the square of each element in A

2.1.2 Example for (b):

Input:

```
A = [1, 2, 3, 4, 5, 6, 7, 8]
```

Output:

(b) Generate the power of 3 of each element in A.

Cubes (Traditional): [1, 8, 27, 64, 125, 216, 343, 512]

Cubes (Higher-order): [1, 8, 27, 64, 125, 216, 343, 512]

Explanation: The code generates the cube of each element in array A using both traditional and higher-order functions.

Traditional way: Uses a for-loop to iterate through each element in A, calculates its cube, and appends the result to a list.

Higher-order functions: Uses the 'map' function combined with a 'lambda' function to achieve the same result in a more concise manner.

```
# (b) Generate the power of 3 of each element in A.
# Traditional way
cubes = []
for x in A:
    cubes.append(x ** 3)
print("Cubes (Traditional):", cubes)

# Higher-order functions
cubes = list(map(lambda x: x ** 3, A))
print("Cubes (Higher-order):", cubes)
```

Cubes (Traditional): [1, 8, 27, 64, 125, 216, 343, 512]
Cubes (Higher-order): [1, 8, 27, 64, 125, 216, 343, 512]

Figure 2: Example code and output for generating the cube of each element in A

2.1.3 Example for (c):

Input:

A = [1, 2, 3, 4, 5, 6, 7, 8]

Output:

(c) Return the resultant square in the range of [20, 40].

Squares in range 20-40 (Traditional): [25, 36]

Squares in range 20-40 (Higher-order): [25, 36]

Explanation: The code returns the squares of elements in array A that fall within the range [20, 40] using both traditional and higher-order functions.

Traditional way: Uses a for-loop to iterate through each element in A, calculates its

square, and checks if the square is within the specified range before appending it to a list.

Higher-order functions: Uses the 'map' function to calculate the square of each element and the 'filter' function combined with a 'lambda' function to filter the results within the specified range.

```
# (c) Return the resultant square in the range of [20, 40].
# Traditional way
squares_in_range = []
for x in A:
    square = x ** 2
    if 20 <= square <= 40:
        squares_in_range.append(square)
print("Squares in range 20-40 (Traditional):", squares_in_range)

# Higher-order functions
squares_in_range = list(filter(lambda x: 20 <= x <= 40, map(lambda x: x ** 2, A)))
print("Squares in range 20-40 (Higher-order):", squares_in_range)

Squares in range 20-40 (Traditional): [25, 36]
Squares in range 20-40 (Higher-order): [25, 36]
```

Figure 3: Example code and output for returning squares in the range of [20, 40]

2.1.4 Example for (d):

Input:

A = [1, 2, 3, 4, 5, 6, 7, 8]

Output:

(d) Generate the elements in A which are even numbers.

Even numbers (Traditional): [2, 4, 6, 8]

Even numbers (Higher-order): [2, 4, 6, 8]

Explanation: The code generates the elements in array A that are even numbers using both traditional and higher-order functions.

Traditional way: Uses a for-loop to iterate through each element in A, checks if the element is even, and appends it to a list if it is. Higher-order functions: Uses the 'filter' function combined with a 'lambda' function to filter out the even numbers from the array.

2.1.5 Conclusion:

Exercise 1 demonstrates the power and simplicity of higher-order functions in Python. By comparing traditional for-loop implementations with higher-order functions, we can see the following:

```
[ ] # (d) Generate the elements in A which are even number
    # Traditional way
    evens = []
    for x in A:
        if x % 2 == 0:
            evens.append(x)
    print("Even numbers (Traditional):", evens)

    # Higher-order functions
    evens = list(filter(lambda x: x % 2 == 0, A))
    print("Even numbers (Higher-order):", evens)
```

```
⇒ Even numbers (Traditional): [2, 4, 6, 8]
   Even numbers (Higher-order): [2, 4, 6, 8]
```

Figure 4: Example code and output for generating the elements in A which are even numbers

Readability and Conciseness: Higher-order functions like ‘map’, ‘filter’, and ‘lambda’ provide more concise and readable code. They abstract away the details of iteration, allowing us to focus on the operations being performed.

Functionality: Both traditional and higher-order function approaches successfully perform the required tasks, such as generating squares and cubes, filtering results within a specific range, and identifying even numbers. This confirms that higher-order functions are as effective as traditional methods. Flexibility: The use of ‘lambda’ functions with ‘map’ and ‘filter’ showcases the flexibility and power of higher-order functions to perform complex operations with minimal code.

Overall, higher-order functions offer a more elegant and efficient way to process collections in Python, making the code easier to understand and maintain.

2.2 Exercise 2

Description of Exercise 2: Assuming two pixels at coordinates (x1, y1), (x2, y2). The distance (dst) of two pixels is defined as:

$$\text{dst} = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

Using lambda function to calculate the distance of two pixels at coordinates (4, 5), (3, 2).

Example:

Input:

Coordinates of the two pixels: (4, 5), (3, 2)

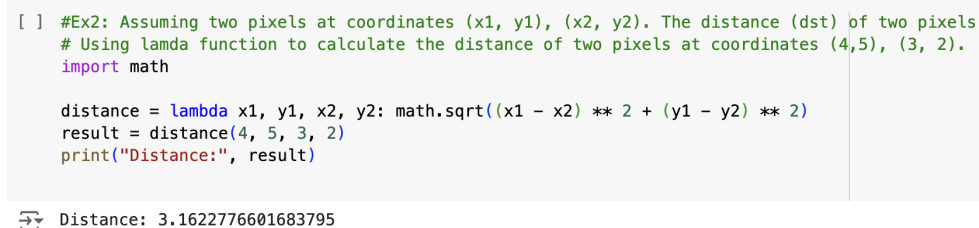
Output:

Distance: 3.1622776601683795

Explanation: The code calculates the Euclidean distance between two points using a lambda function.

Lambda Function: Defines an anonymous function to calculate the distance between two points.

Calculation: Uses the formula $\sqrt{(x1 - x2)^2 + (y1 - y2)^2}$ to compute the distance.



```
[ ] #Ex2: Assuming two pixels at coordinates (x1, y1), (x2, y2). The distance (dst) of two pixels
# Using lambda function to calculate the distance of two pixels at coordinates (4,5), (3, 2).
import math

distance = lambda x1, y1, x2, y2: math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
result = distance(4, 5, 3, 2)
print("Distance:", result)
```

Distance: 3.1622776601683795

Figure 5: Example code and output for calculating the distance between two pixels

Conclusion: Exercise 2 illustrates the use of lambda functions to perform mathematical calculations in a concise and efficient manner. By using a lambda function to calculate the Euclidean distance between two points, the following key points are highlighted:

Conciseness: The lambda function allows us to define an anonymous function for the distance calculation in a single line, making the code more concise and readable.

Flexibility: Lambda functions can be easily integrated into other functions or used as standalone expressions, providing flexibility in various programming scenarios.

Clarity: The mathematical formula for distance calculation is clearly expressed within the lambda function, making the logic easy to understand at a glance.

Overall, the use of lambda functions for mathematical operations demonstrates the power of functional programming concepts in writing clear, concise, and maintainable code.

2.3 Exercise 3

Description of Exercise 3: Using higher-order functions to generate the names of all dogs from the given list of animals.

Example:

Input:

```
animals = [
    {'type': 'Dog', 'name': 'Lucy'},
    {'type': 'Cat', 'name': 'Buddy'},
    {'type': 'Rabbit', 'name': 'Jack'},
    {'type': 'Cat', 'name': 'Duke'},
    {'type': 'Rabbit', 'name': 'Sadie'},
    {'type': 'Dog', 'name': 'Bella'}
]
```

Output:

```
Dog names: ['Lucy', 'Bella']
```

Explanation: The code generates the names of all dogs from the given list of animals using higher-order functions.

Filter Function: Filters out the dictionaries where the 'type' is 'Dog'.

Map Function: Maps the filtered dictionaries to extract the 'name' of each dog.



```
[ ] #Ex3: Using higher-order functions to generate the names of all dogs from the given list of animals
animals = [
    {'type': 'Dog', 'name': 'Lucy'},
    {'type': 'Cat', 'name': 'Buddy'},
    {'type': 'Rabbit', 'name': 'Jack'},
    {'type': 'Cat', 'name': 'Duke'},
    {'type': 'Rabbit', 'name': 'Sadie'},
    {'type': 'Dog', 'name': 'Bella'}
]

dog_names = list(map(lambda animal: animal['name'], filter(lambda animal: animal['type'] == 'Dog', animals)))
print("Dog names:", dog_names)
```

Dog names: ['Lucy', 'Bella']

Figure 6: Example code and output for generating the names of all dogs from the given list of animals

Conclusion: Exercise 3 showcases the effectiveness of higher-order functions in filtering and mapping data from a list of dictionaries. By using 'filter' and 'map' functions in

combination with lambda expressions, the exercise highlights the following advantages:

Simplicity and Readability: Higher-order functions provide a simple and readable way to filter and transform data. The code clearly expresses the intent to filter out dogs and extract their names.

Efficiency: The combined use of 'filter' and 'map' functions ensures that the filtering and mapping operations are performed efficiently in a single pass over the data.

Flexibility: Lambda functions offer a flexible and concise way to define the operations to be performed on each element, making the code adaptable to various scenarios.

Overall, the use of higher-order functions for data processing tasks demonstrates their power and utility in writing clean, efficient, and maintainable code.

2.4 Exercise 4

Description of Exercise 4: Given an array $B = ['a', 'b', 'c', 'd', 'e']$. Using higher-order functions to:

- (a) Generate sum of elements in B from left to right.
- (b) Generate sum of elements in B from right to left.
- (c) Concatenate 'Y' to the sum of elements in B from left to right.
- (d) Concatenate 'Y' to the sum of elements in B from right to left.

2.4.1 Example for (a):

Input :

```
B = ['a', 'b', 'c', 'd', 'e']
```

Output :

```
Sum from left to right: abcde
```

Explanation: The code generates the sum of elements in array B from left to right using the 'reduce' function.

Reduce Function: Combines the elements of the array from left to right using the

provided lambda function.

```
#(a) Generate sum of elements in B from left to right.
sum_left_to_right = reduce(lambda x, y: x + y, B)
print("Sum from left to right:", sum_left_to_right)
```

Sum from left to right: abcde

Figure 7: Example code and output for generating the sum of elements in B from left to right

2.4.2 Example for (b):

Input:

```
B = ['a', 'b', 'c', 'd', 'e']
```

Output:

Sum from right to left: edcba

Explanation: The code generates the sum of elements in array B from right to left using the 'reduce' function.

Reduce Function: Combines the elements of the array from right to left using the provided lambda function.

```
# (b) Generate sum of elements in B from right to left
sum_right_to_left = reduce(lambda x, y: y + x, B)
print("Sum from right to left:", sum_right_to_left)
```

Sum from right to left: edcba

Figure 8: Example code and output for generating the sum of elements in B from right to left

2.4.3 Example for (c):

Input:

```
B = ['a', 'b', 'c', 'd', 'e']
```

Output:

Concatenate 'Y' from left to right: abcdeY

Explanation: The code concatenates 'Y' to the sum of elements in array B from left to right.

Concatenation: The string 'Y' is concatenated to the result of part (a).

```
# (c) Concatenate 'Y' to the sum of elements in B from left to right
concat_left_to_right = sum_left_to_right + 'Y'
print("Concatenate 'Y' from left to right:", concat_left_to_right)
```

Concatenate 'Y' from left to right: abcdeY

Figure 9: Example code and output for concatenating 'Y' to the sum of elements in B from left to right

2.4.4 Example for (d):

Input:

B = ['a', 'b', 'c', 'd', 'e']

Output:

Concatenate 'Y' from right to left: edcbaY

Explanation: The code concatenates 'Y' to the sum of elements in array B from right to left.

Concatenation: The string 'Y' is concatenated to the result of part (b).

```
# (d) Concatenate 'Y' to the sum of elements in B from right to left
concat_right_to_left = sum_right_to_left + 'Y'
print("Concatenate 'Y' from right to left:", concat_right_to_left)
```

Concatenate 'Y' from right to left: edcbaY

Figure 10: Example code and output for concatenating 'Y' to the sum of elements in B from right to left

2.4.5 Conclusion:

Exercise 4 demonstrates the versatility and power of higher-order functions like 'reduce' in manipulating and processing collections of data. By comparing the results of left-to-right and right-to-left operations, we can observe how the order of operations affects the final result. The additional concatenation tasks further illustrate the flexibility of these

functions in combining and transforming data. Overall, higher-order functions provide a concise and expressive way to perform complex data manipulations, enhancing both code readability and maintainability.

2.5 Exercise 5

Colab File: <https://colab.research.google.com/drive/1TAndBiO5aGe8gmvUqQYALDLa9BVi1AGD?u>

2.5.1 Guidance to Use the Python Code

The provided Python script allows users to run specific functions from the exercises. Follow these steps to use the main function in the Python code:

1. Save the code in a file named `run.py`.
2. Open your terminal or command prompt.
3. Navigate to the directory where the `run.py` file is located.
4. Use the following commands to run the specific functions:
 - `python run.py 1`: This command runs all the functions for Exercise 1.
 - `python run.py 1a`: This command runs the function for generating squares of elements in array A using both traditional and higher-order functions.
 - `python run.py 1b`: This command runs the function for generating cubes of elements in array A using both traditional and higher-order functions.
 - `python run.py 1c`: This command runs the function for filtering squares in the range of [20, 40] from array A using both traditional and higher-order functions.
 - `python run.py 1d`: This command runs the function for identifying even numbers in array A using both traditional and higher-order functions.
 - `python run.py 2`: This command runs the function for calculating the distance between two points.
 - `python run.py 3`: This command runs the function for extracting names of dogs

from a list of animals.

- `python run.py 4`: This command runs all the functions for Exercise 4.
- `python run.py 4a`: This command runs the function for summing elements in array B from left to right.
- `python run.py 4b`: This command runs the function for summing elements in array B from right to left.
- `python run.py 4c`: This command runs the function for concatenating 'Y' to the sum of elements in array B from left to right.
- `python run.py 4d`: This command runs the function for concatenating 'Y' to the sum of elements in array B from right to left.

2.5.2 Example Usage

Here are some example commands and their expected outputs:

```
# To run all functions in Exercise 1
```

```
$ python run.py 1
```

Output:

```
Squares (Traditional): [1, 4, 9, 16, 25, 36, 49, 64]
```

```
Squares (Higher-order): [1, 4, 9, 16, 25, 36, 49, 64]
```

```
Cubes (Traditional): [1, 8, 27, 64, 125, 216, 343, 512]
```

```
Cubes (Higher-order): [1, 8, 27, 64, 125, 216, 343, 512]
```

```
Squares in range 20-40 (Traditional): [25, 36]
```

```
Squares in range 20-40 (Higher-order): [25, 36]
```

```
Even numbers (Traditional): [2, 4, 6, 8]
```

```
Even numbers (Higher-order): [2, 4, 6, 8]
```

```
# To generate squares of elements in array A
```

```
$ python run.py 1a
```

Output:

```
Squares (Traditional): [1, 4, 9, 16, 25, 36, 49, 64]
```

```
Squares (Higher-order): [1, 4, 9, 16, 25, 36, 49, 64]
```

```
# To generate cubes of elements in array A
```

```
$ python run.py 1b
```

```
Output:
```

```
Cubes (Traditional): [1, 8, 27, 64, 125, 216, 343, 512]
```

```
Cubes (Higher-order): [1, 8, 27, 64, 125, 216, 343, 512]
```

```
# To filter squares in the range of [20, 40] from array A
```

```
$ python run.py 1c
```

```
Output:
```

```
Squares in range 20-40 (Traditional): [25, 36]
```

```
Squares in range 20-40 (Higher-order): [25, 36]
```

```
# To identify even numbers in array A
```

```
$ python run.py 1d
```

```
Output:
```

```
Even numbers (Traditional): [2, 4, 6, 8]
```

```
Even numbers (Higher-order): [2, 4, 6, 8]
```

```
# To calculate the distance between two points (4, 5) and (3, 2)
```

```
$ python run.py 2
```

```
Input:
```

```
Enter coordinates of first point (x1 y1): 4 5
```

```
Enter coordinates of second point (x2 y2): 3 2
```

```
Output:
```

```
Distance between pixels (4, 5) and (3, 2): 3.1622776601683795
```

```
# To get names of all dogs from the list of animals
```

```
$ python run.py 3
```

```
Output:
```

```
Dog names: ['Lucy', 'Bella']
```

```
# To run all functions in Exercise 4
```

```
$ python run.py 4
```

Output:

```
Sum from left to right: abcde
```

```
Sum from right to left: edcba
```

```
Concatenate 'Y' from left to right: abcdeY
```

```
Concatenate 'Y' from right to left: edcbaY
```

```
# To sum elements in array B from left to right
```

```
$ python run.py 4a
```

Output:

```
Sum from left to right: abcde
```

```
# To sum elements in array B from right to left
```

```
$ python run.py 4b
```

Output:

```
Sum from right to left: edcba
```

```
# To concatenate 'Y' to the sum of elements in array B from left to right
```

```
$ python run.py 4c
```

Output:

```
Concatenate 'Y' from left to right: abcdeY
```

```
# To concatenate 'Y' to the sum of elements in array B from right to left
```

```
$ python run.py 4d
```

Output:

```
Concatenate 'Y' from right to left: edcbaY
```

By following these guidelines, users can easily run any of the functions defined in the Python script and see the results directly in their terminal or command prompt.