Lab 4 - Functional Programming (Section 1)

Course: Principles of Programming Languages (Code: IT092IU)
HCMIU-CSE, Summer 2024
Instructor: Le Thi Ngoc Hanh, PhD

Student Name:

Pham Thai Quoc ITITWE21118
Huynh Phuong Dai ITITWE21039

1. Higher-Order Functions

a) Lambda

Allows us to create anonymous functions.
Syntax: `lambda arguments: expression`

b) Map

Applies a function to all items in an input list; returns a map object.
Syntax: `map(function, arguments)`

c) Reduce

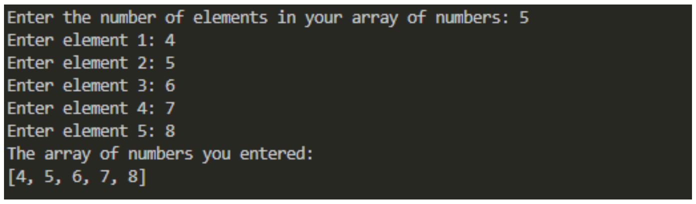
A function with 2 input parameters, function `f` and a list. Instead of iterating each element, `reduce` combines every 2 elements of the array with the input function `f`.
Syntax: `reduce(f, list)`
Required: `from functions import reduce`

d) Compose

Combines multiple functions into a single function. This combined function applies the given functions in sequence, where the output of one function becomes the input of the next.

2. Exercises

For this lab session, we will have a terminal interface with user inputs to control it as required by the lecturer, the input will be simple and the user will have the instructions given upon program launch.
USER INPUT EXAMPLE



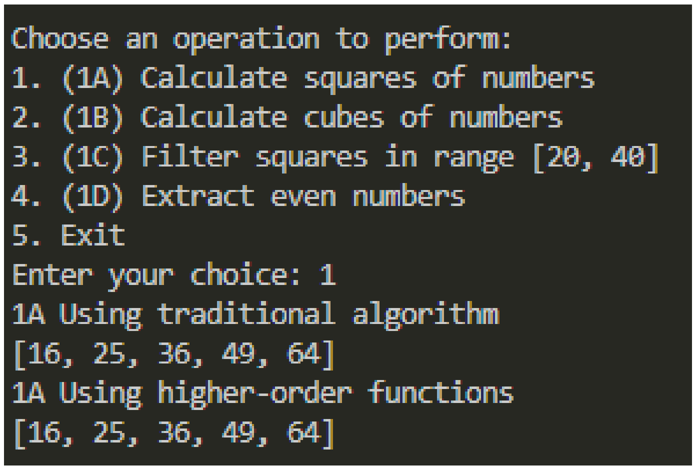
```
Enter the number of elements in your array of numbers: 5
Enter element 1: 4
Enter element 2: 5
Enter element 3: 6
Enter element 4: 7
Enter element 5: 8
The array of numbers you entered:
[4, 5, 6, 7, 8]
```

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:

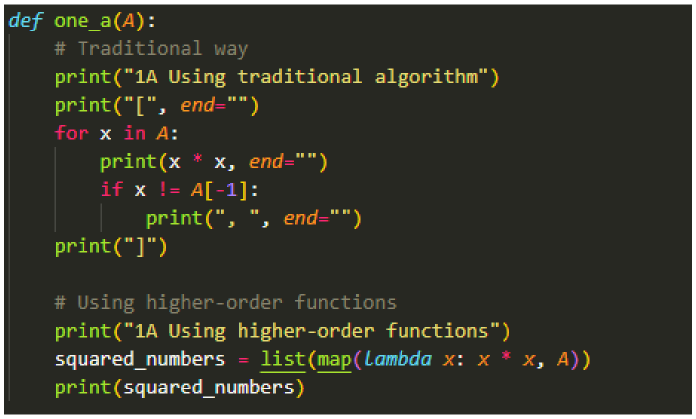
- (a) Generate the square of each element in `A`.

  ANSWER Output:



```
Choose an operation to perform:
1. (1A) Calculate squares of numbers
2. (1B) Calculate cubes of numbers
3. (1C) Filter squares in range [20, 40]
4. (1D) Extract even numbers
5. Exit
Enter your choice: 1
1A Using traditional algorithm
[16, 25, 36, 49, 64]
1A Using higher-order functions
[16, 25, 36, 49, 64]
```

  ANSWER Code:



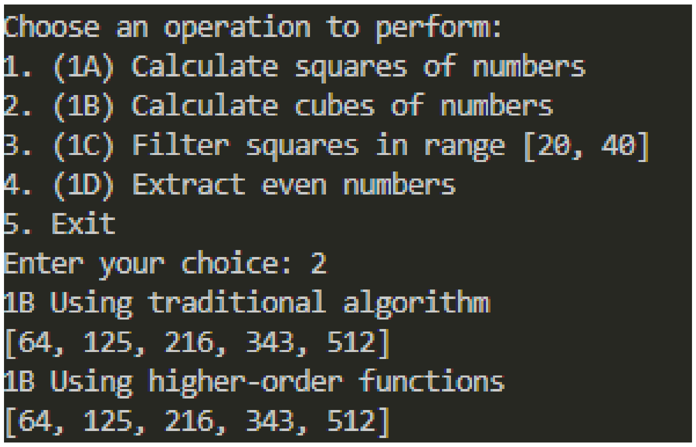
```python
def one_a(A):
    # Traditional way
    print("1A Using traditional algorithm")
    print("[", end="")
    for x in A:
        print(x * x, end="")
        if x != A[-1]:
            print(", ", end="")
    print("]")

    # Using higher-order functions
    print("1A Using higher-order functions")
    squared_numbers = list(map(lambda x: x * x, A))
    print(squared_numbers)
```

  Explanation: For this function we use a simple x time itself algorithm without the use of math library for the traditional way and use list to create a result array with map to go through all contents of the original array and multiply themselves once

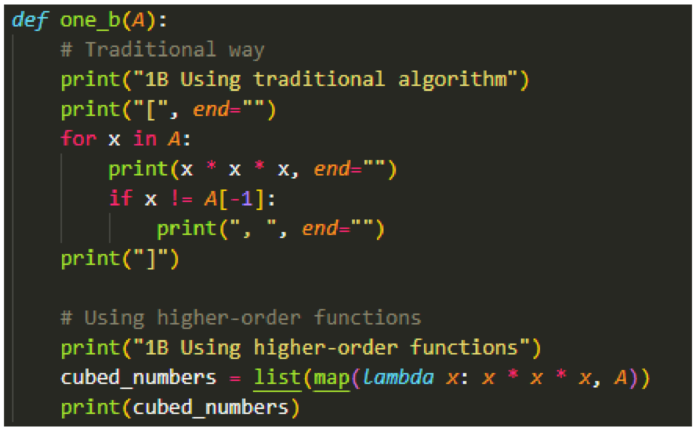
- (b) Generate the power of 3 of each element in `A`.

  ANSWER Output:



```
Choose an operation to perform:
1. (1A) Calculate squares of numbers
2. (1B) Calculate cubes of numbers
3. (1C) Filter squares in range [20, 40]
4. (1D) Extract even numbers
5. Exit
Enter your choice: 2
1B Using traditional algorithm
[64, 125, 216, 343, 512]
1B Using higher-order functions
[64, 125, 216, 343, 512]
```

  ANSWER Code:



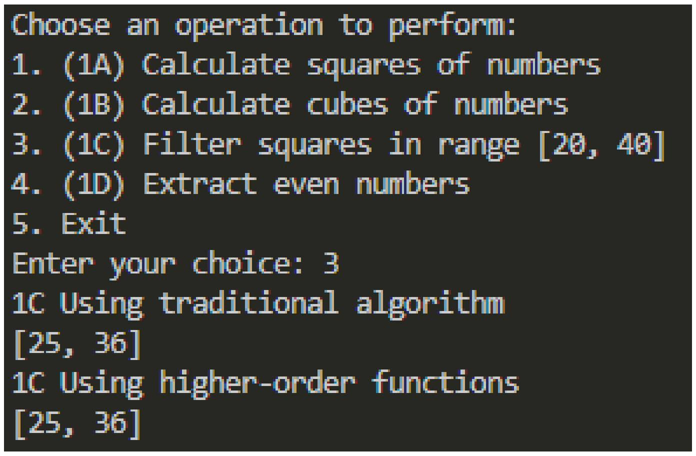
```python
def one_b(A):
    # Traditional way
    print("1B Using traditional algorithm")
    print("[", end="")
    for x in A:
        print(x * x * x, end="")
        if x != A[-1]:
            print(", ", end="")
    print("]")

    # Using higher-order functions
    print("1B Using higher-order functions")
    cubed_numbers = list(map(lambda x: x * x * x, A))
    print(cubed_numbers)
```

  Explanation: For this function, the use of the algorithm is similar with the addition of multiplying the array's element one extra time. Similarly with the higher-order function use

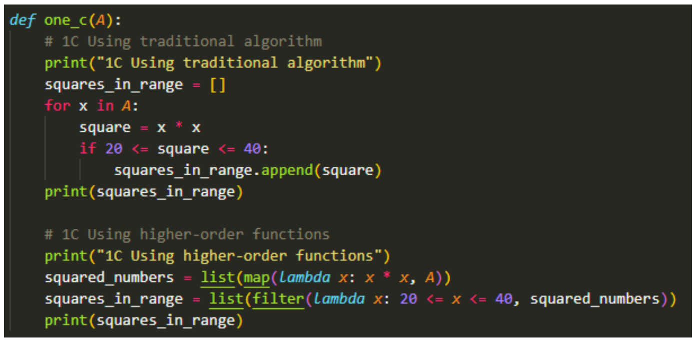
- (c) Return the resultant square in the range of [20, 40].

  ANSWER Output:



```
Choose an operation to perform:
1. (1A) Calculate squares of numbers
2. (1B) Calculate cubes of numbers
3. (1C) Filter squares in range [20, 40]
4. (1D) Extract even numbers
5. Exit
Enter your choice: 3
1C Using traditional algorithm
[25, 36]
1C Using higher-order functions
[25, 36]
```

  ANSWER Code:



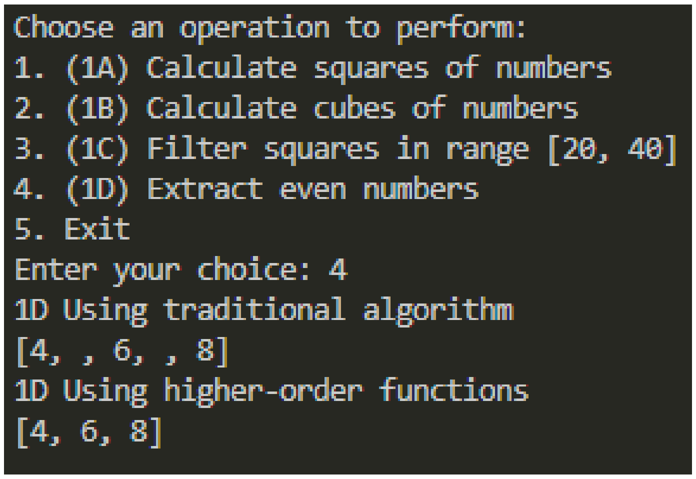
```python
def one_c(A):
    # 1C Using traditional algorithm
    print("1C Using traditional algorithm")
    squares_in_range = []
    for x in A:
        square = x * x
        if 20 <= square <= 40:
            squares_in_range.append(square)
    print(squares_in_range)

    # 1C Using higher-order functions
    print("1C Using higher-order functions")
    squared_numbers = list(map(lambda x: x * x, A))
    squares_in_range = list(filter(lambda x: 20 <= x <= 40, squared_numbers))
    print(squares_in_range)
```

Explanation: For the traditional algorithm, the algorithm still go through all of the elements one by one and multiplying themselves once with one notable exception that it only print out whatever answer within the limit of 20 to 40. For higher-order function we copied the same algorithm from question a and with the additional of the second higher-order function to compare the first resulting array to use **filter** function to look for only the ones within the specified range and save it in the resulting array

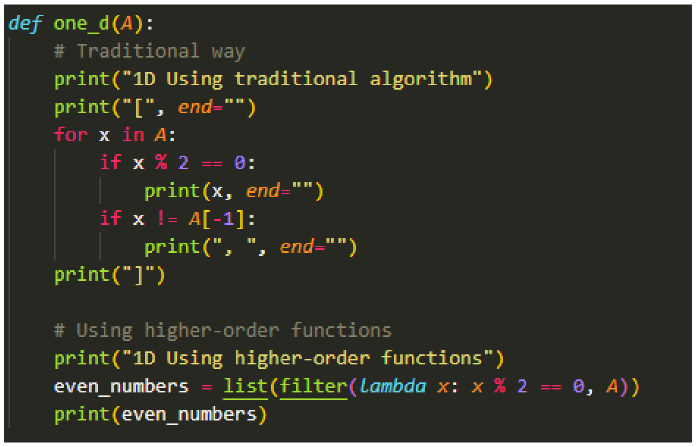
- (d) Generate the elements in `A` which are even numbers.

ANSWER Output:



```
Choose an operation to perform:
1. (1A) Calculate squares of numbers
2. (1B) Calculate cubes of numbers
3. (1C) Filter squares in range [20, 40]
4. (1D) Extract even numbers
5. Exit
Enter your choice: 4
1D Using traditional algorithm
[4, , 6, , 8]
1D Using higher-order functions
[4, 6, 8]
```

ANSWER Code:



```python
def one_d(A):
    # Traditional way
    print("1D Using traditional algorithm")
    print("[", end="")
    for x in A:
        if x % 2 == 0:
            print(x, end="")
        if x != A[-1]:
            print(", ", end="")
    print("]")

    # Using higher-order functions
    print("1D Using higher-order functions")
    even_numbers = list(filter(lambda x: x % 2 == 0, A))
    print(even_numbers)
```

Explanation: For traditional algorithm we use the usual mod argument to check for whether the element is divisible by two or not in order to print out the round numbers only. for the higher order we use **Filter** function to filter the element using the same modulo argument as the condition.

By the end you can exit the program by choosing 4. to exit

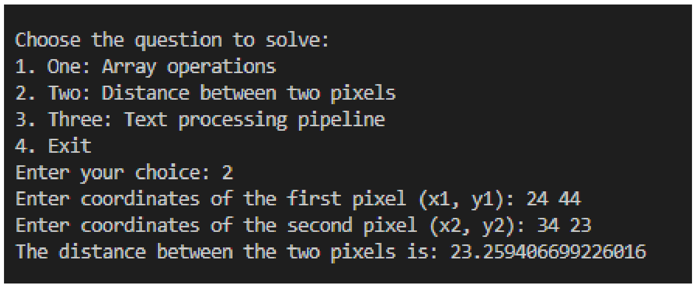
Exercise 2

Assuming two pixels at coordinates (x_1, y_1), (x_2, y_2). The distance (`dst`) of two pixels is defined as:

$$\text{dst} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

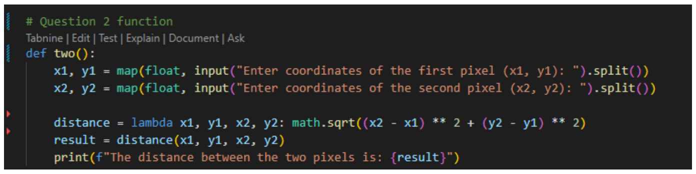
Use the lambda function to calculate the distance of two pixels.

ANSWER Output:



```
Choose the question to solve:
1. One: Array operations
2. Two: Distance between two pixels
3. Three: Text processing pipeline
4. Exit
Enter your choice: 2
Enter coordinates of the first pixel (x1, y1): 24 44
Enter coordinates of the second pixel (x2, y2): 34 23
The distance between the two pixels is: 23.259406699226016
```

ANSWER Code:



```python
# Question 2 function
Tabnine | Edit | Test | Explain | Document | Ask
def two():
    x1, y1 = map(float, input("Enter coordinates of the first pixel (x1, y1): ").split())
    x2, y2 = map(float, input("Enter coordinates of the second pixel (x2, y2): ").split())

    distance = lambda x1, y1, x2, y2: math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
    result = distance(x1, y1, x2, y2)
    print(f"The distance between the two pixels is: {result}")
```

Explanation: In **Exercise 2**, the code calculates the distance between two pixels based on their coordinates.

- The program asks the user to input the coordinates of two pixels: (x_1, y_1) and (x_2, y_2).
- A **lambda** function called `distance` is used to calculate the distance using the formula:

  $$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

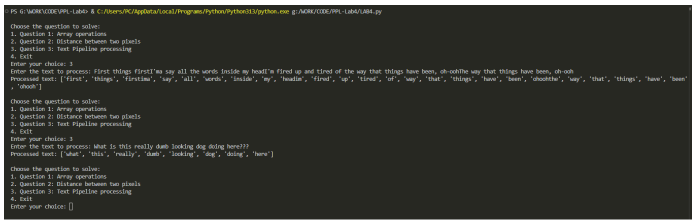
  This lambda function takes the four coordinates and performs the calculation.
- The result is then printed to show the distance between the two pixels.

Exercise 3

Create a series of text processing functions and use `compose()` to build a text processing pipeline. The pipeline should perform the following transformations on a given text:

- Remove punctuation.
- Convert the text to lowercase.
- Split the text into a list of words.
- Filter out common stop words (like "the", "is", "in", etc.).

Output



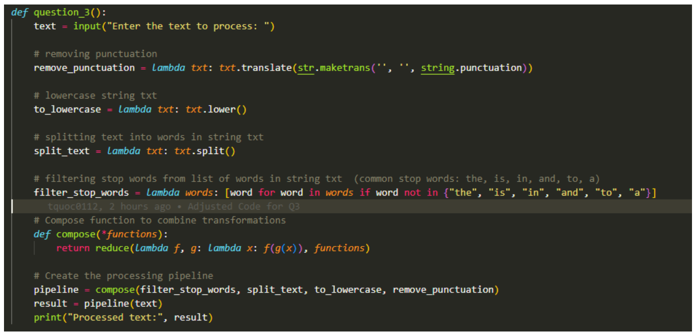
```
PS G:\WORK\CODE\PPL-Lab4> & C:/Users/PC/AppData/Local/Programs/Python/Python313/python.exe g:/WORK/CODE/PPL-Lab4/LAB4.py

Choose the question to solve:
1. Question 1: Array operations
2. Question 2: Distance between two pixels
3. Question 3: Text Pipeline processing
4. Exit
Enter your choice: 3
Enter the text to process: First things firstI'ma say all the words inside my headI'm fired up and tired of the way that things have been, oh-oohThe way that things have been, oh-ooh
Processed text: ['first', 'things', 'firstima', 'say', 'all', 'words', 'inside', 'my', 'headim', 'fired', 'up', 'tired', 'of', 'way', 'that', 'things', 'have', 'been', 'ohoohthe', 'way', 'that', 'things', 'have', 'been'
, 'ohooh']

Choose the question to solve:
1. Question 1: Array operations
2. Question 2: Distance between two pixels
3. Question 3: Text Pipeline processing
4. Exit
Enter your choice: 3
Enter the text to process: What is this really dumb looking dog doing here???
Processed text: ['what', 'this', 'really', 'dumb', 'looking', 'dog', 'doing', 'here']

Choose the question to solve:
1. Question 1: Array operations
2. Question 2: Distance between two pixels
3. Question 3: Text Pipeline processing
4. Exit
Enter your choice: 
```

Code



```python
def question_3():
    text = input("Enter the text to process: ")

    # removing punctuation
    remove_punctuation = lambda txt: txt.translate(str.maketrans('', '', string.punctuation))

    # lowercase string txt
    to_lowercase = lambda txt: txt.lower()

    # splitting text into words in string txt
    split_text = lambda txt: txt.split()

    # filtering stop words from list of words in string txt  (common stop words: the, is, in, and, to, a)
    filter_stop_words = lambda words: [word for word in words if word not in {"the", "is", "in", "and", "to", "a"}]
      tquoc0112, 2 hours ago • Adjusted Code for Q3
    # Compose function to combine transformations
    def compose(*functions):
        return reduce(lambda f, g: lambda x: f(g(x)), functions)

    # Create the processing pipeline
    pipeline = compose(filter_stop_words, split_text, to_lowercase, remove_punctuation)
    result = pipeline(text)
    print("Processed text:", result)
```

Explanation

Hint: Define individual text processing functions for each transformation. Use the `compose()` function to combine these transformations into a single processing pipeline. Apply the composed function to a sample text and output the segmented result.

Requirement

Organize all the above functions by writing a main function that guides users in passing variables to run any defined function.