Exercises

June 8, 2021

[1]:

```python
""" ------------------------- FUNCTIONAL OPERATORS ------------------------- """

import functools


# Functional operators/tools in Python
# filter(function,sequence) filters the given sequence with the help of a
# function that tests each element in the sequence to be true or not.
def fun2(x):
    return x > 0 and x % 2 != 0


filtered_L = filter(fun2, [-1, 3, 2, -3, 4, 5, 7, 8])
print(list(filtered_L))  # You need to explicitly put into list in order to do any further operation

# map(function,sequence) returns a list of the results after applying the given
# function to each item of a given iterable (list, tuple etc.)
def add1(x, y):
    return x + y


map_L = list(map(add1, [-3, 4, 7], [2, 5, 8]))
print(map_L)

# reduce(function,sequence) used to apply a particular function passed in its
# argument to all of the list elements mentioned in the sequence passed along
def fun3(x, y):
    return x + y


reduced_L = functools.reduce(fun3, [1, 2, 3])
print(reduced_L)
```

```
[3, 5, 7]
[-1, 9, 15]
6
```

[2]:

```
################################### DIRECT ASSIGNMENT␣
↪############################################################

print('DIRECT ASSIGNMENT')
i = [1, 2, 3]
j = i
print('Are the 2 object created by direct assingment same? -> {}'.format(id(i)␣
↪== id(j)))
i.append(4)
print('Appending 4 into original list : {}'.format(i))
print('Appending 4 into original list : {}'.format(j))
j.append(5)
print('Appending 5 into copy list : {}'.format(i))
print('Appending 5 into copy list : {}'.format(j))

print('DIRECT ASSIGNMENT NESTED')

i = [1, 2, 3, [4, 5]]
j = i[:]
print('Are the 2 object created by direct assingment same? -> {}'.format(id(i)␣
↪== id(j)))
print('Are the 2 object created by direct assingment same? -> {}'.
↪format(id(i[3]) == id(j[3])))
i[3].append(6)
print('Appending 6 into 3rd element of original list : {}'.format(i))
print('Appending 6 into 3rd element of original list : {}'.format(j))
j[3].append(7)
print('Appending 7 into 3rd element copy list : {}'.format(i))
print('Appending 7 into 3rd element of copy list : {}'.format(j))


################################### LIST SLICING ␣
↪############################################################
print('LIST SLICING')

i = [1, 2, 3]
j = i[:]
print('Are the 2 object created by list slicing same? -> {}'.format(id(i) ==␣
↪id(j)))
i.append(4)
print('Appending 4 into original list : {}'.format(i))
print('Appending 4 into original list : {}'.format(j))
j.append(5)
print('Appending 5 into copy list : {}'.format(i))
print('Appending 5 into copy list : {}'.format(j))

print('LIST SLICING NESTED')
```

```
i = [1, 2, 3, [4, 5]]
j = i[:]
print('Are the 2 object created by list slicing same? -> {}'.format(id(i) ==␣
 ↪id(j)))
print('Are the 2 object created by list slicing same? -> {}'.format(id(i[3]) ==␣
 ↪id(j[3])))
i[3].append(6)
print('Appending 6 into 3rd element of original list : {}'.format(i))
print('Appending 6 into 3rd element of original list : {}'.format(j))
j[3].append(7)
print('Appending 7 into 3rd element copy list : {}'.format(i))
print('Appending 7 into 3rd element of copy list : {}'.format(j))


################################ Shallow Copy␣
 ↪#######################################

import copy
print('SHALLOW COPY')

i = [1, 2, 3]
j = copy.copy(i)
print('Are the 2 object created by shallow copy same? -> {}'.format(id(i) ==␣
 ↪id(j)))
i.append(4)
print('Appending 4 into original list : {}'.format(i))
print('Appending 4 into original list : {}'.format(j))
j.append(5)
print('Appending 5 into copy list : {}'.format(i))
print('Appending 5 into copy list : {}'.format(j))

print('SHALLOW COPY NESTED')

i = [1, 2, 3, [4, 5]]
j = copy.copy(i)
print('Are the 2 object created by shallow copy same? -> {}'.format(id(i) ==␣
 ↪id(j)))
print('Are the 2 object created by shallow copy same? -> {}'.format(id(i[3]) ==␣
 ↪id(j[3])))
i[3].append(6)
print('Appending 6 into 3rd element of original list : {}'.format(i))
print('Appending 6 into 3rd element of original list : {}'.format(j))
j[3].append(7)
print('Appending 7 into 3rd element copy list : {}'.format(i))
print('Appending 7 into 3rd element of copy list : {}'.format(j))
```

```python
################################# Deep Copy ####################################
print('DEEPCOPY')
i = [1, 2, 3]
j = copy.deepcopy(i)
print('Are the 2 object created by deepcopy same? -> {}'.format(id(i) == id(j)))
i.append(4)
print('Appending 4 into original list : {}'.format(i))
print('Appending 4 into original list : {}'.format(j))
j.append(5)
print('Appending 5 into copy list : {}'.format(i))
print('Appending 5 into copy list : {}'.format(j))

print('DEEPCOPY NESTED')
i = [1, 2, 3, [4, 5]]
j = copy.deepcopy(i)
print('Are the 2 object created by deepcopy same? -> {}'.format(id(i) == id(j)))
print('Are the 2 object created by deepcopy same? -> {}'.format(id(i[3]) == id(j[3])))
i[3].append(6)
print('Appending 6 into 3rd element of original list : {}'.format(i))
print('Appending 6 into 3rd element of original list : {}'.format(j))
j[3].append(7)
print('Appending 7 into 3rd element copy list : {}'.format(i))
print('Appending 7 into 3rd element of copy list : {}'.format(j))
```

```
DIRECT ASSIGNMENT
Are the 2 object created by direct assingment same? -> True
Appending 4 into original list : [1, 2, 3, 4]
Appending 4 into original list : [1, 2, 3, 4]
Appending 5 into copy list : [1, 2, 3, 4, 5]
Appending 5 into copy list : [1, 2, 3, 4, 5]
DIRECT ASSIGNMENT NESTED
Are the 2 object created by direct assingment same? -> False
Are the 2 object created by direct assingment same? -> True
Appending 6 into 3rd element of original list : [1, 2, 3, [4, 5, 6]]
Appending 6 into 3rd element of original list : [1, 2, 3, [4, 5, 6]]
Appending 7 into 3rd element copy list : [1, 2, 3, [4, 5, 6, 7]]
Appending 7 into 3rd element of copy list : [1, 2, 3, [4, 5, 6, 7]]
LIST SLICING
Are the 2 object created by list slicing same? -> False
Appending 4 into original list : [1, 2, 3, 4]
Appending 4 into original list : [1, 2, 3]
Appending 5 into copy list : [1, 2, 3, 4]
Appending 5 into copy list : [1, 2, 3, 5]
LIST SLICING NESTED
Are the 2 object created by list slicing same? -> False
Are the 2 object created by list slicing same? -> True
Appending 6 into 3rd element of original list : [1, 2, 3, [4, 5, 6]]
Appending 6 into 3rd element of original list : [1, 2, 3, [4, 5, 6]]
Appending 7 into 3rd element copy list : [1, 2, 3, [4, 5, 6, 7]]
Appending 7 into 3rd element of copy list : [1, 2, 3, [4, 5, 6, 7]]
SHALLOW COPY
Are the 2 object created by shallow copy same? -> False
Appending 4 into original list : [1, 2, 3, 4]
Appending 4 into original list : [1, 2, 3]
Appending 5 into copy list : [1, 2, 3, 4]
Appending 5 into copy list : [1, 2, 3, 5]
SHALLOW COPY NESTED
Are the 2 object created by shallow copy same? -> False
Are the 2 object created by shallow copy same? -> True
Appending 6 into 3rd element of original list : [1, 2, 3, [4, 5, 6]]
Appending 6 into 3rd element of original list : [1, 2, 3, [4, 5, 6]]
Appending 7 into 3rd element copy list : [1, 2, 3, [4, 5, 6, 7]]
Appending 7 into 3rd element of copy list : [1, 2, 3, [4, 5, 6, 7]]
DEEPCOPY
Are the 2 object created by deepcopy same? -> False
Appending 4 into original list : [1, 2, 3, 4]
Appending 4 into original list : [1, 2, 3]
Appending 5 into copy list : [1, 2, 3, 4]
Appending 5 into copy list : [1, 2, 3, 5]
DEEPCOPY NESTED
Are the 2 object created by deepcopy same? -> False
Are the 2 object created by deepcopy same? -> False
Appending 6 into 3rd element of original list : [1, 2, 3, [4, 5, 6]]
Appending 6 into 3rd element of original list : [1, 2, 3, [4, 5]]
Appending 7 into 3rd element copy list : [1, 2, 3, [4, 5, 6]]
Appending 7 into 3rd element of copy list : [1, 2, 3, [4, 5, 7]]
```

[3]:

```python
class Person:
    def __init__(self, name):  # this method adds an object data attribute␣
 ↪‘name’
        print("-> Person")
        self.name = name
        print("<- Person")

    def modify_address(self, new_address):  # this method adds an object data␣
 ↪attribute ‘address’
        self.address = new_address

    def get_address(self):  # it returns the value of the data attribute␣
 ↪‘address’
        return self.address  # be careful! modify_address MUST be called before␣
 ↪get_address !!
```

```python
class Student(Person):
    def __init__(self, matricola=123, **kwargs):  # **kwargs is a special parameter
        print("-> Student")
        super().__init__(**kwargs)
        self.matricola = matricola  # it adds a new data attribute
        print("<- Student")

    def getAddress(self):
        return self.address

    def matr(self):
        return self.matricola


class Worker(Person):
    def __init__(self, salary=1000, **kwargs):  # **kwargs is a special parameter
        print("-> Worker")
        super().__init__(**kwargs)
        self.salary = salary  # it adds a new data attribute
        print("<- Worker")

    def get_salary(self):
        return self.salary

    def getAddress(self):
        return self.address


class WorkingStudent(Student, Worker):
    def __init__(self, n="name", m=100, s=1000):
        print("-> WorkingStudent")
        super().__init__(name=n, matricola=m, salary=s)
        print("<- WorkingStudent")

    def getAddressW():
        return Worker.getAddress()

    def getAddressS():
        return Student.getAddress()


if __name__ == "__main__":
    w = WorkingStudent(n="pippo", s=2500, m=123)
    print(w.get_salary())
    print(w.matr())
    print(w.name)
    print(w.modify_address('Via Roma 56'))
    print(w.address)
```

```
-> WorkingStudent
-> Student
-> Worker
-> Person
<- Person
<- Worker
<- Student
<- WorkingStudent
2500
123
pippo
None
Via Roma 56
```

1 EXERCISES 1

1) given two input lists L1 and L2, write a function which selects all elements which are present both in L1 and in L2, and returns them in a list L3

[4]:

```
def return_all_elements(l1: list, l2: list) -> list:
    """1) given two input lists L1 and L2, write a function which
    selects all elements which are present both in L1 and in L2,
    and returns them in a list L3"""
    _l3 = []
    for element1 in l1:
        if element1 in l2:
            _l3.append(element1)
    return _l3


print(return_all_elements([1, 2, 4], [1, 2]))
```

```
[1, 2]
```

2) compute the transposed matrix of a given input matrix of size NxN

[5]:

```
def transpose(matrix):
    """2) compute the transposed matrix of a given input matrix of size NxN"""
    _transposed = []
    for row in range(len(matrix[0])):
        _transposed.append([matrix[i][row] for i in range(len(matrix))])
    return _transposed


print(transpose([[1, 2], [1, 2]]))
```

```
[[1, 1], [2, 2]]
```

3) given a rectangular input matrix M, write a function which returns a boolean value True if and only if there exist two different rows in M, whose sum gives the null vector. Do the same for two different columns.

[6]:

```
def check_null_vector(matrix):
    """3) given a rectangular input matrix M, write a function which returns
    a boolean value True if and only if there exist two different rows in M,
    whose sum gives the null vector. Do the same for two different columns."""
    _shallow_matrix = matrix
    i = 0
    for row in matrix:
        j = i + 1
        while j < len(matrix):
            if sum(row) + sum(matrix[j]) == 0:
                return True
            j += 1
        i += 1
    return False


print(check_null_vector([[31, 0], [31, -3], [-31, 3]]))
```

```
True
```

4) Compute the height of a tree

5) Compute the height of a node in a binary search tree

6) verify if two binary are equal

7) Given a not balanced binary search tre construct a balanced search tree with the same information.

2 EXERCISES 2

2) define a class for a stack

[7]:

```
class Stack:
    """2) define a class for a stack"""

    def __init__(self):
        self.stack = []
    def push(self, data):
        self.stack.append(data)

    def pop(self):
        if not self.emptystack():
            return self.stack.pop()

    def top(self):
        if not self.emptystack():
            return self.stack[-1]

    def emptystack(self):
        return self.stack == []

    def printstack(self):
        print(self.stack)

    def __repr__(self):
        return '{}'.format([element for element in self.stack])

s = Stack()
s.push(2)
s.push(3)
s.printstack()
print(f"Popped : {s.pop()}")
s.push(4)
s.push(5)
print(f"Popped : {s.pop()}")
s.push(6)
s.printstack()
```

```
[2, 3]
Popped : 3
Popped : 5
[2, 4, 6]
```

1) define a class for a queue

[8]:

```
class Queue:
    """1) define a class for a queue"""
    def __init__(self):
        self.queue = []

    def push(self, data):
        self.queue.insert(0, data)

    def pop(self):
        if not self.emptystack():
            return self.queue.pop()

    def top(self):
        if not self.emptystack():
            return self.queue[-1]

    def emptystack(self):
        return self.queue == []

    def printqueue(self):
        print(self.queue)

    def __repr__(self):
        return '{}'.format([element for element in self.queue])

q = Queue()
q.push(2)
q.push(3)
q.printqueue()
print(f"Popped : {q.pop()}")
q.push(4)
q.push(5)
print(f"Popped : {q.pop()}")
q.push(6)
q.printqueue()
print(q)
```

```
[3, 2]
Popped : 2
Popped : 3
[6, 5, 4]
[6, 5, 4]
```

3) write a function which takes in input two queues of integers and returns an ordered queue containing the values in the input queues (same for two stacks)

[9]:

```
def retrieve_queue(queue):
    if len(queue.queue) == 0:
        return []
    else:
        return [queue.pop()] + retrieve_queue(queue)


def ordered_queue_mergers(queue1: Queue, queue2: Queue) -> Queue:
    """3) write a function which takes in input two queues of integers
    and returns an ordered queue containing the values in the
    input queues (same for two stacks)"""
    _list1 = retrieve_queue(queue1)
    _list1.sort()
    _list2 = retrieve_queue(queue2)
    _list2.sort()
    _merged_queue = Queue()
    idx_2 = 0
    for element in _list1:
        while idx_2 < len(_list2) and element > _list2[idx_2]:
            _merged_queue.push(_list2[idx_2])
            idx_2 += 1
        _merged_queue.push(element)
    while idx_2 < len(_list2):
        _merged_queue.push(_list2[idx_2])
        idx_2 += 1
    return _merged_queue


q1 = Queue()
q1.push(5)
q1.push(3)
q1.push(1)
q2 = Queue()
q2.push(2)
q2.push(4)
q2.push(4)
q2.push(4)
q2.push(1)
print(ordered_queue_mergers(q1, q2))
```

```
[5, 4, 4, 4, 3, 2, 1, 1]
```

4) define a function for computing the product of two matrices

[10]:

```
def matrix_product(matrix1, matrix2):
    """4)define a function for computing the product of two matrices"""
    idx_1 = 0
    _result_matrix = []
    _n_columns_matrix2 = len(matrix2[0])
    for row in matrix1:
        idx_2 = 0
        _temp = []
        while idx_2 < _n_columns_matrix2:
            _temp.append(sum([(row[idx] * matrix2[idx][idx_2]) for idx in␣
 ↪range(len(matrix1[0]))]))
            idx_2 += 1
        _result_matrix.append(_temp)
        idx_1 += 1
    return _result_matrix
```

```python

matrix11 = [[1, 2, 3], [1, 2, 3]]
matrix22 = [[1, 2], [1, 2], [1, 2]]
print(matrix_product(matrix11, matrix22))
```

```
[[6, 12], [6, 12]]
```

5) define a function which given a binary search tree T of integers and a rectangular matrix M of integers verify if there exists a row of M whose values belong to T. (Do the same exercise for ‘a column’).

6) define a function which given a binary search tree T of integers and a rectangular matrix M of integers verify if there exists an ordered row of M (e.g. values in ascending order) whose values are in T.

2.1 All tree exercises from 1 and 2

[11]:

```python
class Tree:
    """
    Class which represent a tree
    """

    def __init__(self, elem, left=None, right=None):
        """
        Constructor for a tree
        """
        self.left = left
        self.right = right
        self.elem = elem

def print_in_order(tree: Tree) -> list:
    """
    Perform the "in-order" traversal of a given tree
    """
    if tree is None:
        return []
    left = print_in_order(tree.left)
    right = print_in_order(tree.right)
    return left + [tree.elem] + right


def height_tree(tree: Tree) -> int:
    """
    Perform the height of a given tree
    Ex1.4) Compute the height of a tree
    """
    if tree is None:
        return 0
    return 1 + max(height_tree(tree.left), height_tree(tree.right))

def height_of_a_node(tree: Tree, node: Tree) -> int:
    """
    Compute the height of a node inside a tree
    Ex1.5) Compute the height of a node in a binary search tree
    """
    if tree is None:
        raise Exception("No Founded Value")
    if tree.elem == node.elem:
        return 0
    if node.elem > tree.elem:
        return 1 + height_of_a_node(tree.right, node)
    else:
        return 1 + height_of_a_node(tree.left, node)

def count_n_node(tree: Tree) -> int:
    """
    Count the number of total nodes in the tree
    """
    if tree is None:
        return 0
    return count_n_node(tree.right)+count_n_node(tree.left)+1

def equal_trees(tree1: Tree, tree2: Tree) -> bool:
    """
    Check whether or not two trees are equal (equal is defined in this case as node value, not simple the structure)
    Ex1.6) verify if two binary are equal
    """
    if not tree1 and not tree2:
        return True
    if tree1.elem != tree2.elem:
        return False
    return equal_trees(tree1.left, tree2.left) and equal_trees(tree1.right, tree2.right)

def balanced_tree(tree: Tree) -> bool:
    """
    Write a program to verify if a binary tree is balanced (all leaves at depth K or K+1)
    """
    if tree==None:
        return True
    if height_tree(tree.left)-height_tree(tree.right)<=1:
        #return balanced_tree(tree.left) and balanced_tree(tree.right)
        return True
    else:
        return False

def from_in_order_to_tree(list_tree_representation: list) -> Tree:
    """
    Return a representation of the inorder Balanced tree as an instance of Tree␣
 ↪class
    """
    if not list_tree_representation:
        return None
    if len(list_tree_representation) == 1:
        return Tree(list_tree_representation[0])
    return Tree(list_tree_representation[int((len(list_tree_representation) +␣
 ↪1) / 2)],
                from_in_order_to_tree(list_tree_representation[:
 ↪(int((len(list_tree_representation) + 1) / 2))]),
               ␣
 ↪from_in_order_to_tree(list_tree_representation[(int((len(list_tree_representation)␣
 ↪+ 1) / 2)) + 1:]))

def from_in_order_to_balanced_tree(list_tree_representation: list) -> Tree:
    """
    Return a representation of the inorder Balanced tree as an instance of Tree␣
 ↪class
    Ex1.7) Given a not balanced binary search tree construct a balanced
    search tree with the same information
    """
    if not list_tree_representation:
        return None
    if len(list_tree_representation) == 1:
        return Tree(list_tree_representation[0])
    mid = len(list_tree_representation) // 2
    return Tree(list_tree_representation[mid],
                from_in_order_to_tree(list_tree_representation[:mid]),
                from_in_order_to_tree(list_tree_representation[mid + 1:]))


def tree_in_row_matrix(tree: Tree, matrix) -> bool:
    """
    Given a bst of integers and a matrix of integers, verify if there exists␣
 ↪and ordered of matrix whose values are in tree
    Ex2.5) define a function which given a binary search tree T of integers and␣
 ↪a rectangular matrix M
    of integers verify if there exists a row of M whose values belong to T. (Do␣
 ↪the same exercise
    for 'a column').
    """
    _internal_tree_representation = print_in_order(tree)
    for row in matrix:
        if all(element in row for element in _internal_tree_representation):
            return True
    return False

def tree_in_ordered_row_matrix(tree: Tree, matrix) -> bool:
    """
    Given a bst of integers and a matrix of integers, verify if there exists␣
 ↪and ordered of matrix whose values are in tree
    Ex2.6) define a function which given a binary search tree T of integers and␣
 ↪a rectangular matrix M
    of integers verify if there exists an ordered row of M (e.g. values in␣
 ↪ascending order) whose
    values are in T.
    """
    _internal_tree_representation = print_in_order(tree)
    for row in matrix:
        sorted_row = sorted(row)
        if all(element in sorted_row for element in␣
 ↪_internal_tree_representation):
            return True
    return False

def transpose(matrix):
    """
    Compute the matrix transpose
    """
    _transposed = []
    for row in range(len(matrix)):
        _transposed.append([matrix[i][row] for i in range(len(matrix))])
    return _transposed

def tree_in_column_matrix(tree: Tree, matrix: list) -> bool:
    """
    Given a bst of integers and a matrix of integers, verify if there exists␣
 ↪and ordered of matrix whose values are in tree
    Ex2.5) define a function which given a binary search tree T of integers and␣
 ↪a rectangular matrix M
    of integers verify if there exists a row of M whose values belong to T. (Do␣
 ↪the same exercise
    for 'a column')
    """
    _internal_tree_representation = print_in_order(tree)
    _transposed_matrix = transpose(matrix)
    for row in _transposed_matrix:
        if all(element in row for element in _internal_tree_representation):
            return True
    return False

def tree_in_ordered_column_matrix(tree: Tree, matrix: list) -> bool:
    """
    Given a bst of integers and a matrix of integers, verify if there exists␣
 ↪and ordered of matrix whose values are in tree
    Ex2.6) define a function which given a binary search tree T of integers and␣
 ↪a rectangular matrix M
    of integers verify if there exists an ordered row of M (e.g. values in␣
 ↪ascending order) whose
    values are in T.
    """
    _internal_tree_representation = print_in_order(tree)
    _transposed_matrix = transpose(matrix)
    for row in _transposed_matrix:
        sorted_row = sorted(row)
        if all(element in row for element in _internal_tree_representation):
            return True
    return False

def find_element_bst(tree: Tree, element: int) -> bool:
    """
    Return if an element is present into a bst
    """
    if tree is None:
        return False
    if tree.elem == element:
        return True
    if element > tree.elem:
        return find_element_bst(tree.right, element)
    else:
        return find_element_bst(tree.left, element)

def add_node_bst(tree: Tree, node: Tree) -> Tree:
    """
    Add a node in a bst, as leaf.
    """
    if tree is None:
        return node
    if tree.elem > node.elem:
        tree.left = add_node_bst(tree.left, node)
    if tree.elem < node.elem:
        tree.right = add_node_bst(tree.right, node)
    return tree
```

[12]:

```
if __name__ == "__main__":
    tree = Tree(4, Tree(2), Tree(5, Tree(4), Tree(7, Tree(6))))
    tree2 = Tree(4, Tree(2), Tree(6, Tree(5)))
    tree3 = Tree(4, Tree(2), Tree(10, None, Tree(12)))

    # Height of a tree
    print("----- Height of a tree")
    print(f"Case 1: {height_tree(tree)}")
    print(f"Case 2: {height_tree(tree2)}")
    print(f"Case 3: {height_tree(tree3)}")
    print('\n')

    # Print in-order
    print("----- Print in-order")
    print(f"Case 1: {print_in_order(tree)}")
    print(f"Case 2: {print_in_order(tree2)}")
    print(f"Case 3: {print_in_order(tree3)}")
    print('\n')

    # Height of a Node
    print("----- Height of a Node")
    print(f"Case 1: {height_of_a_node(tree,Tree(5))}")
    try:
        print(f"Case 2: {height_of_a_node(tree, Tree(11))}")
    except Exception as ex:
        print(f"Case 2: {ex}")
    try:
        print(f"Case 3: {height_of_a_node(tree, Tree(-1))}")
    except Exception as ex:
        print(f"Case 3: {ex}")
    print('\n')

    # Equal Trees
    print("----- Equal Trees")
    print(f"Case 1: {equal_trees(tree3, tree2)}")
    print(f"Case 2: {equal_trees(tree3, tree3)}")
    print(f"Case 3: {equal_trees(tree, tree2)}")
    print(f"Case 4: {equal_trees(tree3, tree)}")
    print('\n')

    # From in-order to tree
    print("----- From in-order to tree")
    tree = from_in_order_to_tree(print_in_order(tree))
    print(f"Case 1: {print_in_order(tree)}")
    print('\n')

    # From in-order to balanced tree
    print("----- From in-order to balanced tree")
    tree4 = Tree(20)
    tree4.left = Tree(15)
    tree4.left.left = Tree(10)
    tree4.left.left.left = Tree(5)
    tree4.left.left.left.left = Tree(2)
    tree4.left.left.left.right = Tree(8)
    print(f"Case 4: {print_in_order(tree4)}")
    print(f"Is Case 4 a balanced tree: {balanced_tree(tree4)}")
    tree4 = from_in_order_to_balanced_tree(print_in_order(tree4))
    print(f"Case 4: {print_in_order(tree4)}")
    print(f"Is Case 4 a balanced tree: {balanced_tree(tree4)}")
    print('\n')

    # Tree in a Matrix Row
    print("----- Tree in a Matrix Row")
    print(f"Case 1:␣
↪{tree_in_row_matrix(tree3,[[0,0,0,0],[2,4,10,12],[1,2,3,4]])}")
    print(f"Case 2: {tree_in_row_matrix(tree3, [[2]])}")
    print('\n')

    # Tree in an Ordered Matrix Row
    print("----- Tree in an Ordered Matrix Row")
    print(f"Case 1:␣
↪{tree_in_ordered_row_matrix(tree3,[[0,0,0,0],[2,4,10,12],[1,2,3,4]])}")
    print(f"Case 2: {tree_in_ordered_row_matrix(tree3, [[2]])}")
    print('\n')

    # Tree in a Matrix Column
    print("----- Tree in a Matrix Column")
    print(f"Case 1:␣
↪{tree_in_column_matrix(tree3,[[2,4,3,10],[4,2,43,50],[10,10,10,10],[12,0,0,0]])}")
    print('\n')

    # Tree in an Ordered Matrix Column
    print("----- Tree in an Ordered Matrix Column")
    print(f"Case 1:␣
↪{tree_in_ordered_column_matrix(tree3,[[2,4,3,10],[4,2,43,50],[10,10,10,10],[12,0,0,0]])}")
    print('\n')

    # Find element in a bst
    print("----- Find element in a bst")
    print(f"Case 1: {find_element_bst(tree, -1)}")
    print(f"Case 2: {find_element_bst(tree, 7)}")
    print('\n')

    # Add node in a bst
    print("----- Add node in a bst")
    print(f"Before: {print_in_order(tree)}")
    tree = add_node_bst(tree, Tree(3))
    print(f"After: {print_in_order(tree)}")
    print('\n')
```

```
----- Height of a tree
Case 1: 4
Case 2: 3
Case 3: 3


----- Print in-order
Case 1: [2, 4, 4, 5, 6, 7]
Case 2: [2, 4, 5, 6]
Case 3: [2, 4, 10, 12]


----- Height of a Node
Case 1: 1
Case 2: No Founded Value
Case 3: No Founded Value


----- Equal Trees
Case 1: False
Case 2: True
Case 3: False
Case 4: False


----- From in-order to tree
Case 1: [2, 4, 4, 5, 6, 7]


----- From in-order to balanced tree
Case 4: [2, 5, 8, 10, 15, 20]
Is Case 4 a balanced tree: False
Case 4: [2, 5, 8, 10, 15, 20]
Is Case 4 a balanced tree: True


----- Tree in a Matrix Row
Case 1: True
Case 2: False
```

```
----- Tree in an Ordered Matrix Row
Case 1: True
Case 2: False


----- Tree in a Matrix Column
Case 1: True


----- Tree in an Ordered Matrix Column
Case 1: True


----- Find element in a bst
Case 1: False
Case 2: True


----- Add node in a bst
Before: [2, 4, 4, 5, 6, 7]
After: [2, 3, 4, 4, 5, 6, 7]
```

3 EXERCISES RE

1) write a pattern for recognizing a legal email address (imagining that the address can only end with .it or .com or .org)

[13]:

```
import re


def email_recognizer(email):
    """ write a pattern for recognizing a legal email address (imagining that␣
 ↪the
    address can only end with .it or .com or .org) """
    pattern = r'([a-z0-9]+)@([a-z]+)\.(com|it|org)'
    re_ex1 = re.search(pattern, str(email))
    print(re_ex1.start(), re_ex1.end())


email_recognizer(email='my email sinan@ciao.it')
```

```
9 22
```

2) write a pattern for recognizing a string which must contain at least two of the following words: legal, Trump, policy

[14]:

```
def orange_recognizer(text):
    """write a pattern for recognizing a string which must contain at least two␣
 ↪of the
    following words: legal, Trump, policy """
    pattern = r"(Trump|(?:\b)*legal|policy)"
    re_ex2 = re.findall(pattern, text)
    print(re_ex2)


orange_recognizer(text='The Trump\'s policy is not legal')


def orange_recognizer1(text):
    count = 0
    if re.search('Trump', text):
        count += 1
    if re.search('legal', text):
        count += 1
    if re.search('policy', text):
        count += 1
    if count >= 2:
        return True
    else:
        return False


print(orange_recognizer1(text='The Trump\'s policy is not legal'))
```

```
['Trump', 'policy', 'legal']
True
```

[15]:

```
""" -------------------------- REGULAR EXPRESSION -------------------------- """
import re
# match(p,s) searches the pattern p in s
# Does not look for subpattern
# the occurrence of the substring recognized must START at the beginning of s
re1=re.match("ab","cab")
re2=re.match("ab","ab")
# re1 is none while re2 not
# re2.start() returns 0 (start position of the string)
# re2.end() returns 0 (end position of the string)
re3=re.match("a*b","ab")
# to match b, ab, aab,...
re4=re.match("ab*","abbbc")
# re4.group() returns the substring recognized with max length instead of the␣
 ↪first
```

```
# Does look for subpattern
# search recognizes substring even if the substring is not at the begin
re5=re.search('ab*','bdacabb')
# re5.start() returns the index of the first occurrence of ab* recognized (a at␣
 ↪2)
re6=re.search('ab+','bdacabb')
# re6.start() returns 4
# re6.end() returns 6
# re6.span() returns span=(4,6)
```

[16]:

```
'''
special characters can be represented by adding the preceding ‘\’, with the␣
 ↪usual ‘escape’ meaning.
For patterns it is convenient to use the Python’s raw notation, which is␣
 ↪obtained by adding an ‘r’
before the pattern. This solves a problem of representation of the ‘backslash’␣
 ↪symbol in patterns.
In standard (non-raw) strings ‘\’ has to represented by a double ‘\\’
e.g. r"a\n" (a followed by newline) or r"\\" (matches one single backslash␣
 ↪‘character’)
Notice that r"\\" is equivalent to "\\\\" (much more cumbersome).


(See the manual for a complete presentation)
"." matches any single character
The parentheses can be used to identify groups
(abc)+ matches one or more consecutive repetitions of the group: 'abc',␣
 ↪'abcabc', etc...
x|y match either x or y (It makes the union of the recognised languages)
x* matches zero or more consecutive repetitions of x
E.g: "a*" --> '', 'a', 'aa', ecc
x+ matches one or more consecutive repetitions of x
x? matches zero or one x

x{m,n} matches i consecutive repetitions of x, where m i n
E.g: "a{3,5}" matches 'aaa', 'aaaa' e 'aaaaa'
\d matches one digit
\D matches one non-digit (it’s the complement of \d)

\s matches one space
\S matches one non space
\w matches one alphanumeric character
\W matches one non alphanumeric character
^ matches the beginning of a string
$ matches the end of a string
\b matches a word boundary (e.g. the change from \w to \W)
```

```
\B matches a position which is not a word boundary
[...] matches a set of characters.
e.g. [abcd] matches the character 'a' or 'b' or 'c' or ‘d’
SPECIAL CHARACTERS
to use them as characters it is necessary an escape symbol (backslash before the special character)
e.g. "pippo\.net" matches the target string "pippo.net"

findall(pattern, string)
it returns a list of all pattern occorrences in string
>>> re.findall("a","a,bacca")
['a', 'a', 'a']
sub(pattern, replacement, string[, count=0])
it replaces with ‘replacement’ all occurrences of pattern in string
>>> re.sub("p1","pippo","abp1bbp1")
'abpippobbpippo'
compile(pattern[, flags])
it creates a pattern object by compiling a regular expression pattern. It is then used for matching.
escape(string)
it does the escaping of all special characters in string
'''
```

[16]:

```
'\nspecial characters can be represented by adding the preceding ‘\\’, with the
usual ‘escape’ meaning.\nFor patterns it is convenient to use the Python’s raw
notation, which is obtained by adding an ‘r’\nbefore the pattern. This solves a
problem of representation of the ‘backslash’ symbol in patterns.\nIn standard
(non-raw) strings ‘\\’ has to represented by a double ‘\\’\ne.g. r"a\n" (a
followed by newline) or r"\\" (matches one single backslash ‘character’)\nNotice
that r"\\" is equivalent to "\\\\" (much more cumbersome).\n\n\n(See the manual
for a complete presentation)\n"." matches any single character\nThe parentheses
can be used to identify groups\n(abc)+ matches one or more consecutive
repetitions of the group: \'abc\', \'abcabc\', etc…\nx|y match either x or y
(It makes the union of the recognised languages)\nx* matches zero or more
consecutive repetitions of x\nE.g: "a*" --> \'\', \'a\', \'aa\', ecc\nx+ matches
one or more consecutive repetitions of x\nx? matches zero or one x\n\nx{m,n}
matches i consecutive repetitions of x, where m i n\nE.g: "a{3,5}" matches
\'aaa\', \'aaaa\' e \'aaaaa\'\n\\d matches one digit\n\\D matches one non-digit
(it’s the complement of \\d)\n\n\\s matches one space\n\\S matches one non
space\n\\w matches one alphanumeric character\n\\W matches one non alphanumeric
character\n^ matches the beginning of a string\n$ matches the end of a
string\n\x08 matches a word boundary (e.g. the change from \\w to \\W)\n\\B
matches a position which is not a word boundary\n[…] matches a set of
characters.\ne.g. [abcd] matches the character \'a\' or \'b\' or \'c\' or
‘d’\nSPECIAL CHARACTERS\nto use them as characters it is necessary an escape
symbol (backslash before the special character)\ne.g. "pippo\\.net" matches the
target string "pippo.net"\n\nfindall(pattern, string)\nit returns a list of all
```

```
pattern occorrences in string\n>>> re.findall("a","a,bacca")\n[\'a\', \'a\',
\'a\']\nsub(pattern, replacement, string[, count=0])\nit replaces with
‘replacement’ all occurrences of pattern in string\n>>>
re.sub("p1","pippo","abp1bbp1")\n\'abpippobbpippo\'\ncompile(pattern[,
flags])\nit creates a pattern object by compiling a regular expression pattern.
It is then used for matching.\nescape(string)\nit does the escaping of all
special characters in string\n'
```

[17]:

```python
re7=re.findall('a','abcada')
# re7 is a LIST of ['a','a','a']

# 'a\d' matches 'a3'
# 'a\D' matches 'ab' but not 'a9'
# '(abc)+' matches 'abc', 'abcabc',...

''' write a pattern for recognizing a legal email address (imagining that the
address can only end with .it or .com or .org) '''
re_ex1=re.search('([a-z0-9]+)@([a-z]+)\.(com|it|org)','la mia email sara95@ciao.
 ↪it')
# re_ex1.start() returns 13, re_ex1.end() returns 27

'''  write a pattern for recognizing a string which must contain at least two␣
 ↪of the
following words: legal, Trump, policy '''
re_ex2=re.findall('(Trump|(?:\b)*legal|policy)','The Trump\'s policy is not␣
 ↪legal')
def recog1(text):
    count=0
    if re.search('Trump',text):
        count+=1
    if re.search('legal',text):
        count+=1
    if re.search('policy',text):
        count+=1
    if count>=2:
        return True
    else:
        return False

''' given a string S and a positive integer N, checks if S contains occurrences
of a substring which begins with one initial substring 'begin' or 'start', end
with substring 'end' and has length strictly bigger than N. The function has to
print all occurrences of such substrings in S '''
def startendstring(S,N):
    L=re.findall(r'(?:start|begin)(?:.*?)end+?',S)
    for el in L:
        if len(el)>N:
            print(el)
```

[18]:

```
# check that a string contains only characters a-z, A-Z and 0-9
re_ex3=re.match(r'^[0-9a-zA-Z]*$','ciao95ehr45')
# find sequences of lowercase letters joined with a underscore
re_ex4=re.findall(r'[a-z]+\_','ciao_')
# find sequences of one upper case letter followed by lower case letters
re_ex5=re.findall(r'[A-Z]{1}[a-z]+','Mi chiamo Sara')
# match a string that has an 'a' followed by anything, ending in 'b'
re_ex6=re.match(r'a.*?b','acab a54%b')
# match a word at the beginning of a string
re_ex7=re.search(r'^\w+','Ciao sono Sara')
# match a word at end of string, with optional punctuation
re_ex8=re.search(r'\w+[!.?]*$','Io sono Groot!')
# matches a word containing 'z', not start or end of the word
re_ex9=re.search(r'\w+z\w+','zio ezio sono arazia ciao')
# search some literals strings 'fox', 'dog', 'horse' in a string 'The quick
# brown fox jumps over the lazy dog.'
re_ex10=re.findall(r'(?:fox|dog|horse)','The quick brown fox jumps over the␣
 ↪lazy dog.')
# find the location within the original string where the pattern 'fox' occurs
re_ex11=re.search(r'fox','The quick brown fox jumps over the lazy dog.')
# re_ex11.start() to have the location
# find all words starting with 'a' or 'e' in a given string
re_ex12=re.findall(r'\b(?:a|e)(?:.*?)\b','an empty space as it is')
# replace all occurrences of space, comma, or dot with a colon
re_ex13=re.sub(r'[\s,.]',':','Ciao Sara, sono io.')
# replace maximum 2 occurrences of space, comma, or dot with a colon
re_ex14=re.sub("[ ,.]", ":", 'Ciao Sara, sono io.', 2)
# find all five characters long word in a string
re_ex15=re.findall(r"\b\w{5}\b", 'The quick brown fox jumps over the lazy dog.')
# extract values between quotation marks of a string
re_ex16=re.findall(r'"(.*?)"', '"Python", "PHP", "Java"')
# remove multiple spaces in a string
re_ex17=re.sub(' +',' ','Python      Exercises')
# remove everything except alphanumeric characters from a string
re_ex18=re.sub(r'[\W_]','','**//Python Exercises// - _12. ')
# split a string at uppercase letters
re_ex19=re.findall('[A-Z][^A-Z]*', 'PythonTutorialAndExercises')
# split at spaces
re_ex20=re.split(r'\s','Python Tutorial And Exercises')
# insert spaces between words starting with capital letters
re_ex21=re.sub(r"(\w)([A-Z])", r"\1 \2", "PythonExercisesPracticeSolution")
# remove words from a string of length bhtween 1 and a given number
re_ex22=re.sub(r'\W*\b\w{1,3}\b','', "The quick brown fox jumps over the lazy␣
 ↪dog.")
# check a decimal with a precision of 2
```

```
re_ex23=re.search(r"""^[0-9]+(\.[0-9]{1,2})?$""",'123.45')
# bool(re_ex23)
```

[19]:

```
# find the occurrence and position of the substrings within a string
'''
text = 'Python exercises, PHP exercises, C# exercises'
pattern = 'exercises'
for match in re.finditer(pattern, text):
    s = match.start()
    e = match.end()
    print('Found "%s" at %d:%d' % (text[s:e], s, e))
'''
# replace whitespaces with an underscore and vice versa
text = 'Python Exercises'
text =text.replace (" ", "_")
text =text.replace ("_", " ")

# convert a date of yyyy-mm-dd format to dd-mm-yyyy format
'''
def change_date_format(dt):
        return re.sub(r'(\d{4})-(\d{1,2})-(\d{1,2})', '\\3-\\2-\\1', dt)
dt1 = "2026-01-02"
print("Original date in YYY-MM-DD Format: ",dt1)
print("New date in DD-MM-YYYY Format: ",change_date_format(dt1))
'''

# Example of sub
# re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
```

[19]:

```
'\ndef change_date_format(dt):\n        return
re.sub(r\'(\\d{4})-(\\d{1,2})-(\\d{1,2})\', \'\\3-\\2-\\1\', dt)\ndt1 =
"2026-01-02"\nprint("Original date in YYY-MM-DD Format: ",dt1)\nprint("New date
in DD-MM-YYYY Format: ",change_date_format(dt1))\n'
```

[20]:

```
''' COMMONLY USED:
Time in 24-hour format   ^([01]?[0-9]|2[0-3]):[0-5][0-9]$
Email                    ^.+@.+$
                         ^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})$
                         [\w-]+@([\w-]+\.)+[\w-]+
URL                      ^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.
↪-]*)*\/?$
                         [a-z]*[:.]+\S+
Re                       r'@((\S)+)'
hashtag                  r'#((\S)+)'
end of line              r'$'

-DIGITS
```

```
Positive Integer        ^\d+$
Negative Integer        ^-\d+$
Integer                 ^-?\d+$
Whole Numbers - /^\d+$/
Decimal Numbers - /^\d*\.\d+$/
Whole + Decimal Numbers - /^\d*(\.\d+)?$/
Negative, Positive Whole + Decimal Numbers - /^-?\d*(\.\d+)?$/

-ALPHANUMERIC CHARACTERS
Alphanumeric without space - /^[a-zA-Z0-9]*$/
Alphanumeric with space - /^[a-zA-Z0-9 ]*$/
'''

'''
Character classes                                     Quantifiers & Alternation
.        any character except newline                 a* a+ a?        0 or␣
 ↪more, 1 or more, 0 or 1
\w \d \s        word, digit, whitespace                 a{5}␣
 ↪a{2,}         exactly five, two or more
\W \D \S        not word, digit, whitespace             a{1,3}        between␣
 ↪one & three
[abc]         any of a, b, or c                        a+? a{2,}?        match␣
 ↪as few as possible
[^abc]         not a, b, or c                           ab|cd         match ab␣
 ↪or cd
[a-g]         character between a & g
Anchors                                                 Groups & Lookaround
^abc$         start / end of the string                (abc)         capture␣
 ↪group
\b        word boundary                                 \1        backreference␣
 ↪to group #1
Escaped characters                                     (?:
 ↪abc)         non-capturing group
\. \* \\   \ is used to escape special chars  (?=abc)       positive lookahead
\t \n \r        tab, linefeed, carriage return         (?!abc)        negative␣
 ↪lookahead
'''
```

[20]:

```
'\nCharacter classes\t\t                    Quantifiers & Alternation\n.\tany
character except newline\t         a* a+ a?\t0 or more, 1 or more, 0 or 1\n\\w
\\d \\s\tword, digit, whitespace\t         a{5} a{2,}\texactly five, two or
more\n\\W \\D \\S\tnot word, digit, whitespace\t     a{1,3}\tbetween one &
three\n[abc]\tany of a, b, or c\t                a+? a{2,}?\tmatch as few as
possible\n[^abc]\tnot a, b, or c\t                   ab|cd\tmatch ab or
cd\n[a-g]\tcharacter between a & g\nAnchors\t\t
Groups & Lookaround\n^abc$\tstart / end of the string\t         (abc)\tcapture
```

```
group\n\x08\tword boundary\t                           \x01\tbackreference to
group #1\nEscaped characters\t\t                       (?:abc)\tnon-capturing
group\n\\. \\* \\  \\ is used to escape special chars  (?=abc)\tpositive
lookahead\n\t \n \r\ttab, linefeed, carriage return\t (?!abc)\tnegative
lookahead\n'
```

[21]:

```
'''Esame 2018'''
'''write a function which, given a string and a positive integer N, checks if␣
 ↪string contains
occurrences of a substring which:  begins with 'begin' ends with 'end' and has␣
 ↪a length >N.
the function has to print the final substrings'''

def str_proc(string,N):
    L=[]
    rule1=r'(begin.+?end)'
    r1=re.findall(rule1, string)
    for e in r1: L.append(e)
    rule2=r'(start.+?end)'
    r2=re-findall(rule2,string)
    for e in r2: L.append(e)
    print(string, "--->",L)
    for i in L:
        if len(e) <=N: L.remove(e)
    return L

''' write a function which given two input binary search tree of ntegers T1 T2,␣
 ↪returns a boolean
value true if and only if: all the odd negative values in T2 are contained in␣
 ↪T2, all the positive
values in T2 are contained in T1'''

def neg_odd(x):
    return x%2!=0 and x<0
def pos(x):
    return x>0
def verify_values(T1,T2):
    Lt1=[]
    Lt2=[]

    Tree2List(T1,Lt1)
    print("List from T1 ->",Lt1)
    Tree2List(T2, Lt2)
    print("List from T2 ->",Lt2)

    pos_t2_list=list(filter(pos,Lt2))
    print("positive values in t2: ", pos_t2_list)
    neg_odd_t2_list=list(filter(neg_odd,Lt2))
    print("negative odd values in t2: ",neg_odd_t2_list)

    return all(neg_odd_elem in Lt1 for neg_odd_elem in neg_odd_t2_list) and all(pos_elem in Lt1 for pos_elem in pos_t2_list)
```

4 Exam 29.05.2019

Exercise 1) Write a Python function which takes in input two rectangular matrixes M, and M1 of integer values, having the same size NxM, and which returns a boolean value True if and only if there exist one column from M and one from M1 such that the sum of the values in each of the two columns is the same. For instance if M =[[3,2,1],[4,0,5]] and M1 =[[4,0,1],[6,6,6]] the returned value is True as the sum of the values in column 0 of M gives the value 7, as the sum of the values of column 2 in M1. If M1= [[4,0,5],[1,1,0]] the returned value is False.

[22]:

```
from typing import List


def transpose(matrix: List[List[int]]) -> List[List[int]]:
    """
    Compute the transpose of a matrix
    """
    _transposed = []
    for row in range(len(matrix[0])):
        _transposed.append([matrix[i][row] for i in range(len(matrix))])
    return _transposed


def solution_1(matrix_1: List[List[int]], matrix_2: List[List[int]])-> bool:
    _transposed_matrix1 = transpose(matrix_1)
    _transposed_matrix2 = transpose(matrix_2)
    for row_1 in _transposed_matrix1:
        row_1_sum = sum(element for element in row_1)
        for row_2 in _transposed_matrix2:
            if row_1_sum == sum(element for element in row_2):
                return True
    return False


if __name__ == "__main__":
    print(f"Case 1: Reference: True -> result: {solution_1([[3, 2, 1], [4, 0, 5]], [[4,0,1],[6,6,6]])}")
    print(f"Case 2: Reference: False -> result: {solution_1([[3,2,1],[4,0,5]],[[4,0,5],[1,1,0]])}")
```

```
Case 1: Reference: True -> result: True
Case 2: Reference: False -> result: False
```

Exercise 2) Write a Python function, which, given two input binary trees of integers T1 and T2, defined following class Tree: def **init**(self, elem=None, left=None, right=None): self.elem = elem self.left = left self.right = right returns a boolean value True if and only if all values in T1 appear at least twice in T2. The value False is returned otherwise. So, for instance if T1= Tree(11,Tree(9,Tree(2))), T2= Tree(9,Tree(2),Tree(9,Tree(2,Tree(11),Tree(5)),Tree(11))), returns True.

[23]:

```python
class Tree:
    """
    Class which represent a tree
    """

    def __init__(self, elem=None, left=None, right=None):
        """
        Constructor for a tree
        """
        self.left = left
        self.right = right
        self.elem = elem


def print_in_order(tree: Tree) -> list:
    """
    Perform the "in-order" traversal of a given tree
    """
    if tree is None:
        return []
    left = print_in_order(tree.left)
    right = print_in_order(tree.right)
    return left + [tree.elem] + right


def solution_2(tree_1: Tree, tree_2: Tree) -> bool:
    _tree_1_representation = print_in_order(tree_1)
    _tree_2_representation = print_in_order(tree_2)
    if len(_tree_2_representation) < (len(_tree_1_representation) * 2): return␣
 ↪False
    for node in _tree_1_representation:
        if len(list(filter((lambda node_2: node_2 == node),␣
 ↪_tree_2_representation))) < 2:
            return False
    return True


if __name__ == "__main__":
    print(f"Case 1: Reference: True -> result: {solution_2(Tree(11, Tree(9,␣
↪Tree(2))), Tree(9, Tree(2), Tree(9, Tree(2, Tree(11), Tree(5)),␣
↪Tree(11))))}")
```

```
Case 1: Reference: True -> result: True
```

Exercise 3) Consider the module “re” for defining Python regular expressions. Write a Python function which, given a string S and a positive integer N, checks if S contains occurrences of a substring which begins with one initial substring ‘xx’, contains the substring ‘yy’, ends with a substring ‘zz’ and has length >N. The function has to print all occurrences of such substrings in S. For instance if s = ‘abxxa1yydfczzbxxbbbyyxxzzcaaa12cccyy’, and N==8, it will print the two following substrings: ‘xxa1yydfczz’ and ‘xxbbbyyxxzz’.

[24]:

```
import re


def solution_3(target_string, length):
    matching = re.findall(r"xx.+?yy.+?zz",target_string)
    # Option 1
    print(list(map(lambda element: element,list(filter(lambda element: element␣
 ↪if len(element) > length else "",matching)))))
    # Option 2 - better, more compant and elegant
    print([element for element in matching if len(element) > length])


if __name__ == "__main__":
    print("Case 1: Reference : {xxa1yydfczz, xxbbbyyxxzz}")
    solution_3("abxxa1yydfczzbxxbbbyyxxzzcaaa12cccyyxxsdfgfgdfgdzz",8)
```

```
Case 1: Reference : {xxa1yydfczz, xxbbbyyxxzz}
['xxa1yydfczz', 'xxbbbyyxxzz']
['xxa1yydfczz', 'xxbbbyyxxzz']
```

5 Exam 25.06.2020

Exercise 1) Write a Python function which takes in input two rectangular matrices M, and M1 of integer values, having the same size NxM, and which returns a new matrix M2 of size NxK (K<=M) which contains the columns in M1 for which there exists at least a column (say) k in M such that the values in the column in M1 are a subset of the values in the column k of M, and the sum of the values in the column in M1 is less than the sum of the values in the column k of M. If there is no column in M1 which satisfies the above properties then M2 = []. For instance if M =[[3,2,1,4],[4,0,5,7],[4,4,0,-1]] and M1 =[[3, 0,1,7],[4,6,5,-1],[3,4,-2,-1]] the returned matrix is. M2 = [[3, 7],[4, -1],[3,-1]].

[25]:

```
from typing import List


def transpose(matrix: List[List[int]]) -> List[List[int]]:
    """
    Compute the transpose of a matrix
    :param matrix: The source matrix
    :return: The transposed version
    """
    _transposed = []
    for row in range(len(matrix[0])):
        _transposed.append([matrix[i][row] for i in range(len(matrix))])
    return _transposed


def solution_1(matrix_1: List[List[int]], matrix_2: List[List[int]])-> bool:
    _transposed_matrix1 = transpose(matrix_1)
    _transposed_matrix2 = transpose(matrix_2)
    _matrix_3 = []
    for row_1 in _transposed_matrix1:
        row_1_sum = sum(element for element in row_1)
        for row_2 in _transposed_matrix2:
            if all(element in row_1 for element in row_2) and sum(element for␣
 ↪element in row_2) < row_1_sum:
                _matrix_3.append(row_2)
    if len(_matrix_3) != 0:
        _transposed_matrix3 = transpose(_matrix_3)
        return _transposed_matrix3
    else:
        return _matrix_3


if __name__ == "__main__":
    print(f"Case 1: Reference: True -> result:␣
 ↪{solution_1([[3,2,1,4],[4,0,5,7],[4,4,0,-1]], [[3,␣
 ↪0,1,7],[4,6,5,-1],[3,4,-2,-1]])}")
    print(f"Case 2: Reference: False -> result:␣
 ↪{solution_1([[3,2,1,4],[4,0,5,7],[4,4,0,-1]], [[3,␣
 ↪0,1,5],[2,6,5,-1],[3,4,-2,-1]])}")
```

```
Case 1: Reference: True -> result: [[3, 7], [4, -1], [3, -1]]
Case 2: Reference: False -> result: []
```

Exercise 2) Write a Python function, which, given one input binary tree of integers T, defined following class Tree: def **init**(self, elem=None, left=None, right=None): self.elem = elem self.left = left self.right = right takes in input two binary trees T1 and T2 of integer values. No value is repeated in one of these trees. The function has to return the value True if and only if the trees have the same height and for each level k of the trees the sum of the values in level k of T1 is greater or equal to the sum of the values in level k of T2. So, for instance if T1= Tree(9,Tree(5,Tree(2),Tree(4))), T2= Tree(9,Tree(2,Tree(4))), it returns True.

[26]:

```
class Tree:
    """
    Class which represent a tree as a node, it use more or less the same␣
 ↪notation as we used in prolog,
    the only difference is that here we omit the nil value when there is an␣
 ↪empty node.
    """

    def __init__(self, elem=None, left=None, right=None):
        """
        Constructor for a node, the sub-trees can be omitted if there is no␣
 ↪value for these.
        :param elem: The node payload.
        :param left: the left sub-tree (defined as another Node)
        :param right: the right sub-tree (defined as another Node)
        """
        self.left = left
        self.right = right
        self.elem = elem


def print_in_order(tree: Tree) -> list:
    """
    Perform the "in-order" traversal of a given tree
    :param tree: the tree to be evaluated
    :return: a list which contains all the nodes of the tree
    """
    if tree is None:
        return []
    left = print_in_order(tree.left)
    right = print_in_order(tree.right)
    return left + [tree.elem] + right


def height_tree(tree: Tree) -> int:
    """
    Perform the height of a given tree
    :param tree: the tree to be evaluated
    :return: an int value which represent the height of the tree
    """
    if tree is None:
        return 0
    return 1 + max(height_tree(tree.left), height_tree(tree.right))


def nodes_at_level(tree: Tree, level: int) -> list:
    """
    Perform the operation of finding nodes of a given tree at the given level
    :param tree: the tree to be evaluated
    :param level: the level to be examined the finding nodes operaiton
    :return: a list which contains all the nodes of the tree at a specific level
    """
    if tree is not None:
        if level != 0:
            return nodes_at_level(tree.left, level - 1) + nodes_at_level(tree.
 ↪right, level - 1)
        else:
            return [tree.elem]
    else:
        return []

def solution_2(tree_1: Tree, tree_2: Tree) -> bool:
    _tree_1_representation = print_in_order(tree_1)
    _tree_2_representation = print_in_order(tree_2)
    if height_tree(tree_1) == height_tree(tree_2):
        if all(sum(nodes_at_level(tree_1, level)) >= sum(nodes_at_level(tree_2,␣
 ↪level)) for level in range(height_tree(tree_1))):
            return True
    return False


if __name__ == "__main__":
    print(f"Case 1: Reference: True -> result:␣
 ↪{solution_2(Tree(9,Tree(5,Tree(2),Tree(4))), Tree(9,Tree(2,Tree(4))))}")
    print(f"Case 2: Reference: True -> result:␣
 ↪{solution_2(Tree(9,Tree(5,Tree(2),Tree(4))), Tree(9,Tree(6,Tree(4))))}")
```

```
Case 1: Reference: True -> result: True
Case 2: Reference: True -> result: False
```

[ ]: