Ministry of Education of Republic of Moldova

Technical University of Moldova CIM Faculty

Anglophone Department

Rеport MD

Laboratory work Nr.1



|  |  |  |
| --- | --- | --- |
| **Performed by** |  | |
| **St. gr.FAF-212:** |  | **Nastas C.** |
| **Verifed by asis. univ.:** |  | **Fiștic C.** |

Chişinău 2021

# **1. Subsets**

Given an integer array set of **unique** elements, return *all possible subsets (the power set)*.

Return the solution in **any order**.

**Example 1:**

**Input:** set = [1,2,3]

**Output:** [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

**Example 2:**

**Input:** set = [0]

**Output:** [[],[0]]

def truthtable(n):  
 if n < 1:  
 return [[]]  
 subtable = truthtable(n-1)  
 return [ row + [v] for row in subtable for v in [0,1] ]  
  
def printPowerSet(set):  
 tt = truthtable(len(set))  
 print('[',end='')  
 for subtt in tt:  
 print('[',end='')   
 for idx in range(0,len(subtt)):  
 if subtt[idx]==1:  
 print(f'{ set[idx]} ',end ="")  
 if(tt[len(tt)-1]==subtt):  
 print("]",end='')  
 else:  
 print("]",end='')  
 print(']')  
 print(tt)  
set = [[1,2,3],2,3]  
  
printPowerSet(set)

Function truthtable(n) is used as a utility function to generate truth table for n variables which is going to be used for printing the power set later on.

printPowerSet(set) is our main function, we are taking all sub-truth tables of n variables which is the length of our initial set.

The idea behind this program is that we print the element from **set** at position **idx** where **subtt[idx] is 1**.

**subtt** – **sub truthtable**(example : for 3 variables x y z – a sub truth table will be 0 0 0 or 1 1 1)

and lastly some prints with **[ ]** to make it more visually appealing

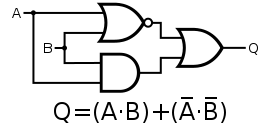
## **2. XNOR**

Create a program that would ask for two boolean values (true or false, 0 or 1) and would output the result for the XNOR operation performed on them.

You're allowed to use only `and`, `or` and `not` operations.

def xnor(x,y):  
 if ((x and y)or(not x and not y))==True:  
 return 1  
 return 0  
  
def main():  
 print(xnor(0,0))  
 print(xnor(0,1))  
 print(xnor(1,0))  
 print(xnor(1,1))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

Logic behind our program:



### **3. Regular Expression Matching**

Given an input string **string** and a pattern **pattern**, implement regular expression matching with support for **’ . ’** and **’ \* ’** where:

* **‘ - ‘** Matches any single character.​​​​
* **‘ \* ‘** Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial).

**Example 1:**

**Input:** string = "aa", pattern = "a"

**Output:** false

**Explanation:** "a" does not match the entire string "aa".

**Example 2:**

**Input:** string = "aa", pattern = "a\*"

**Output:** true

**Explanation:** '\*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

**Example 3:**

**Input:** string = "ab", pattern = ".\*"

**Output:** true

**Explanation:** ".\*" means "zero or more (\*) of any character (.)".

**Example 4:**

**Input:** string = "aab", pattern = "c\*a\*b"

**Output:** true

**Explanation:** c can be repeated 0 times, a can be repeated 1 time. Therefore, it matches "aab".

**Example 5:**

**Input:** string = "mississippi", pattern = "mis\*is\*p\*."

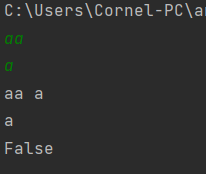
**Output:** false

def match(expression, pattern):  
 exp = expression  
 pat = pattern  
 print(exp,pat)  
 if not exp and not pat:  
 return True  
 elif exp and not pat:  
 return False  
 elif not exp and pat:  
 return False  
  
 if len(pat) >= 3 and pat[0] == '.' and pat[1] == '\*' and pat[2].isalpha():  
 pat = pat[2:]  
 while exp[0] != pat[0]:  
 exp = exp[1:]  
 if exp and pat:  
 p = pat[1:]  
 pat = p  
 return match(exp, pat)  
 elif len(pat) == 2 and pat[0] == '.' and pat[1] == '\*':  
 return True  
 elif len(pat) >= 2 and pat[1] == '\*':  
 t = pat[0]  
 pat = pat[2:]  
 if t == '.':  
 ch = exp[0]  
 while len(exp)>0 and exp[0] == ch:  
 exp = exp[1:]  
 return match(exp,pat)  
 if t == exp[0]:  
 while len(exp) > 0 and exp[0] == t:  
 exp = exp[1:]  
 return match(exp, pat)  
 return match(exp, pat)  
  
 elif pat[0] == '.':  
 pat = pat[1:]  
 exp = exp[1:]  
 return match(exp, pat)  
 elif pat[0].isalpha():  
 if exp[0] == pat[0]:  
 while exp != "" and pat != "" and exp[0] == pat[0]:  
 if len(pat) > 1 and (pat[1] == '\*' or pat[1] == '.'):  
 break  
 exp = exp[1:]  
 pat = pat[1:]  
 return match(exp, pat)  
 else:  
 return False  
  
  
  
  
  
expression = str(input())  
pattern = str(input())  
print(match(expression, pattern))

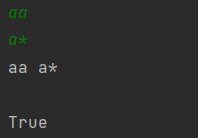
Logic behind our program is that we follow each step of the condition based on priority and recursively till the moment when either expression or pattern becomes an empty string or both do (or in some particular cases we return True/False based on the regulations)

In the following screenshots there is output on how my program is working for each example given:

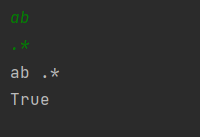
Example 1:



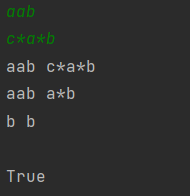
Example 2:



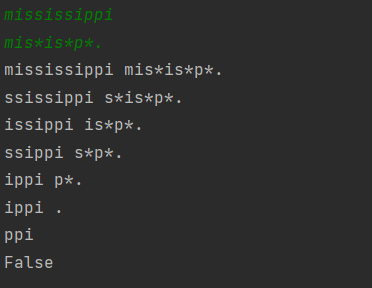
Example 3:



Example 4:



Example 5:



#### **4. Truth table solver**

You have to write a program that computes the truth table for various expressions. The set of expressions are limited to:

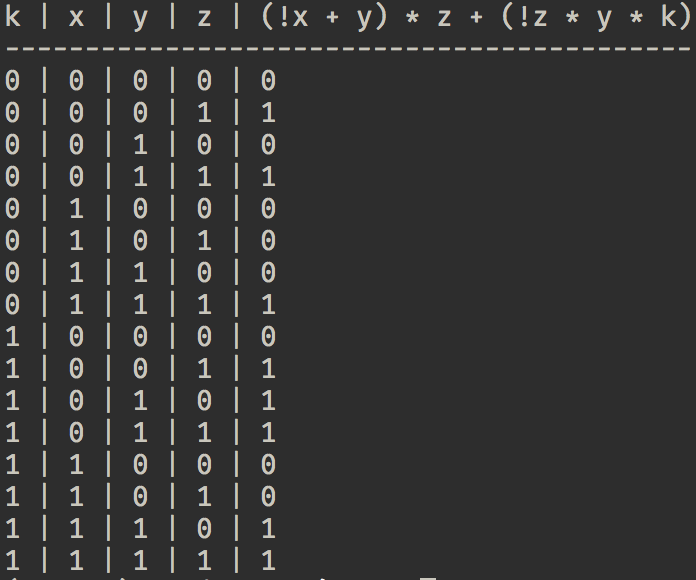
- `and` operation

- `or` operation

- `not` operation

- supports parenthesis

An example of your program input is `(!x + y) \* z + (!z \* y \* k)` and it should print out:



Here are some examples of input that your program should support

```

x + y

!x \* y

(!x + y) \* x + y \* !k

```

**Note:**

I strongly recommend to use the python **`eval`** function. Inventing math operations and their execution priority is **not** the aim of this exercise.

expression = input()  
dictionary ={}  
vars = []  
for i in range(0,len(expression)):  
 if expression[i].isalpha():  
 if expression[i] not in dictionary:  
 dictionary[expression[i]] = 0  
 vars.append(expression[i])  
for var in vars:  
 print(f'{var}|',end="")  
print(expression)  
expression=expression.replace('\*','or')  
expression=expression.replace('+','and')  
expression=expression.replace('!','not ')  
arr = [0]\*(len(dictionary))  
  
for i in range(0,2\*\*len(dictionary)):  
 for j in range(0,len(dictionary)):  
 val = (i & (1<<j))  
 if(val > 0):  
 val = 1  
 arr[j] = val  
 k=0  
 for key in dictionary:  
 dictionary[key] = arr[k]  
 k+=1  
 for elem in arr:  
 print(elem,end="|")  
 print(eval(expression,None,dictionary))

we show the function eval to take the values of variables from the dictionary which is constantly changed in the inner scope of the for (in range (0,len(dict))

values of the variables are changed by left bit shift operator

##### **5. Leibniz harmonic triangle**

Write a program that prints the harmonic triangle for the depth `n`, where `n` is an input value.

**Tip:**

If you're using Python you might look into **`fractions`** module.

def leibniz(size):  
 result = [[0 for rows in range(size + 1)] for cols in range(size + 1)]  
 i = 0  
 while (i <= size):  
 j = 0  
 while (j <= i):  
 if (j == i or j == 0):  
 result[i][j] = 1  
 else:  
 result[i][j] = result[i - 1][j - 1] + result[i - 1][j]  
 j += 1  
 i += 1  
 i = 1  
 while (i <= size):  
 j = i  
 while (j < size):  
 print("\t", end="")  
 j += 1  
 j = 1  
 while (j <= i):  
 if (i == 1 and j == 1):  
 print("1".center(8),end='')  
 else:  
 r = str('1/' + str(result[i - 1][j - 1] \* i)).center(8)  
 print(r,end='')  
 j += 1  
 print()  
 i += 1  
   
def main():  
 leibniz(10)  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

#

#  
#  
#

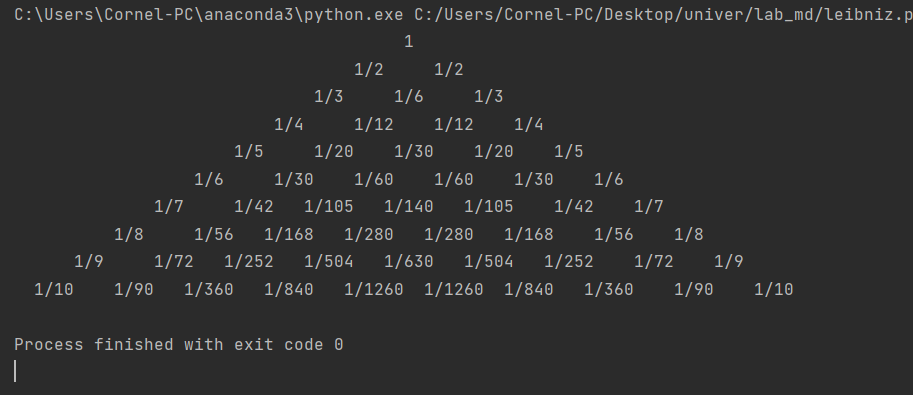
#

#

#

#

#

1

L+R if L or R is edge : C = L \* C\_n-1 If there’s no L or R element : C\_n=**n** of row \* 1

\ /

C \* ( current row)

## **Bonus: A game of life foreplay** (aka [[Elementary cellular automaton](https://en.wikipedia.org/wiki/Elementary_cellular_automaton)])

In this problem we're going to take a look at elementary cellular automaton. Every cell is like a small micro organism with a few primitive rules. When combining with other cells they form interesting patterns. There also is an interesting ([ted talk](https://www.youtube.com/watch?v=60P7717-XOQ)) given by Stephen Wolfram that touches on this topic.

Your task is to randomly generate a list (let's say of length 200, it's up to you in the end, just make sure to be long enough) containing only the numbers `0` and `1`. Then you start iterating over the list in order to compute the \*next generation\*. The rules that apply for the next generation are the following.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

For instance if the cells `1`, `2` and `3` have the value `1 1 0` the 2nd cell of the next generation will be  `1`.

**PRO Tip:**

READ THIS LINK:<https://natureofcode.com/book/chapter-7-cellular-automata/>

###### *Note:*

For computing the first and the last cell you can consider the missing parent to be `0`.

Now you have to compute the next 100 generations and print the resulting matrix with color for value `1` and with white for value `1`. Once You've done that try to change the first generation from randomly generated numbers to all values to be `0` and the last element is `1`. Observe the result.

The rule applied above is called [rule 110](https://en.wikipedia.org/wiki/Rule_110), there is actually a [list of rules](https://en.wikipedia.org/wiki/Elementary_cellular_automaton) that renders quite interesting patterns.

Change arbitrary the initial rule and observe the differences.

*Maybe you can find a new interesting pattern for Bunica's covor.*

###### ***Bonus task:***

Make your program in a way that it would be easy to change the number of pixels rendered for every cell. For instance my cell is 1 x 1 pixels. And by changing one or two variables my cell would change to 5 x 5 pixels.

import numpy as np  
import matplotlib.pyplot as plt  
  
  
def rule\_index(triplet):  
 L, C, R = triplet # triplet = (1,0,0) or (1,1,1) binary representation of a cell  
 index = 7 - (4\*L + 2\*C + R) #  
 return int(index)  
  
  
  
def next\_gen(initial, n\_generations, rule\_number):  
 rule\_string = np.binary\_repr(rule\_number, 8) # rule in binary format  
 rule = np.array([int(bit) for bit in rule\_string]) # make the rule in array format  
  
 m\_cells = len(initial) # 8 by default  
 nextgen = np.zeros((n\_generations, m\_cells)) # generate a new generations with m\_cells( length of initial state)  
 nextgen[0, :] = initial # replace first generated generation with initial generation  
 # print(rule)  
 for gen in range(1, n\_generations): # updating the generations according to rules number  
 all\_triplets = np.stack(  
 [  
 np.roll(nextgen[gen - 1, :], 1),  
 nextgen[gen - 1, :],  
 np.roll(nextgen[gen - 1, :], -1),  
 ]  
 )  
 nextgen[gen, :] = rule[np.apply\_along\_axis(rule\_index, 0, all\_triplets)]  
 # print(nextgen[gen])  
 # modifying according to the utility function created the first generation and the given triplet  
 return nextgen  
  
  
plt.rcParams['image.cmap'] = 'binary'  
rng = np.random.RandomState(0)  
rng = rng.randint(0, 2, 500)  
rng[-1] = 1  
data = next\_gen(rng, 300, 60)  
  
fig, ax = plt.subplots(figsize=(16,9))  
ax.matshow(data)  
  
plt.show()

# Rule 60

