# More Exercises: Lists Advanced

Additional exercises for the [Python Fundamentals Course @SoftUni](https://softuni.bg/trainings/3368/python-fundamentals-may-2021).

Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/1732>.

***Note: All the exercises are excluded from your homework!***

## Social Distribution

*A core idea of several left-wing ideologies is that the wealthiest should support the poorest, no matter what and that is exactly what you are called to do for this problem.*

On the first line you will be given the **population** (numbers separated by comma and space **", "**). On the second line you will be given the **minimum wealth**. You should **distribute** the wealth, so that there is no part of the population that has **less than the minimum** wealth. To do that, you should always take wealth from the **wealthiest part of the population**. There will be cases, where the distribution will **not be possible**. In that case, print **"No equal distribution possible"**.

population = [int(x) for x in input().split(', ')]  
minimum\_wealth = int(input())  
  
if sum(population) < len(population) \* minimum\_wealth:  
 print('No equal distribution possible')  
 exit()  
else:  
 for country in population:  
 if country < minimum\_wealth:  
 richest\_country = max(population)  
 needed\_wealth = minimum\_wealth - country  
 index\_of\_richest\_country = population.index(richest\_country)  
 index\_of\_country = population.index(country)  
 population[index\_of\_richest\_country] -= needed\_wealth  
 population[index\_of\_country] += needed\_wealth  
  
 print(population)

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2, 3, 5, 15, 75  5 | [5, 5, 5, 15, 70] |
| 1, 1, 1, 70, 75  20 | [20, 20, 20, 51, 37] |
| 2, 3, 5, 45, 45  30 | No equal distribution possible |

## Take/Skip Rope

Write a program, which reads a **string** and **skips** through it, extracting a **hidden message**. The algorithm you should implement is as follows:

Let us take the string “skipTest\_String044170” as an example.

Take every **digit** from the string and **store it** somewhere. After that, **remove** all the digits from the string. After this operation, you should have **two lists of items** - the **numbers list** and the **non-numbers list**:

* Numbers' list: [0, 4, 4, 1, 7, 0]
* Non-numbers: [s, k, i, p, T, e, s, t, \_, S, t, r, i, n, g]

After that, take every digit in the **numbers list** and split it up into a **take list** and a **skip list**, depending on whether the digit is in an **even** or an **odd** index:

* Numbers' list: [0, 4, 4, 1, 7, 0]
* Take list: [0, 4, 7]
* Skip list: [4, 1, 0]

Afterwards, **iterate** over both lists:

* **First**, **take m** characters from the **non-numbers list** and store it in a **result string**
* **Then,** **skip** n characters from the **non-numbers list**

Note that the skipped characters are **summed up** as they go. The process would look like this:

1. Current string: **"skipTest\_String"**. Take **0** characters and skip **4** characters:

* Taken string: **""**
* Skipped string: "skip"

1. The remaining string looks like this: **"**Test\_String". Take **4** characters and skip **1** character:
   * Taken string: **"Test"**
   * Skipped string: **"\_"**
2. The string looks like this: **"**String**"**. Take **7** characters and skip **0** characters:
   * Taken string: **"String"**
   * Skipped string: **""**
3. The final string is **"TestString"**.

After that, print the **final string** on the console.

### Constraints

* The count of digits in the input string will **always be even**.
* The encrypted message will contain any printable ASCII character.

word = list(input())  
  
numbers = []  
take\_list = []  
skip\_list = []  
non\_numbers = []  
skip\_index = 0  
result = ''  
  
for i in range(len(word)):  
 if word[i].isdigit():  
 numbers.append(word[i])  
 else:  
 non\_numbers.append(word[i])  
  
for i in range(len(numbers)):  
 if i % 2 == 0:  
 take\_list.append(numbers[i])  
 else:  
 skip\_list.append(numbers[i])  
  
for i in range(len(take\_list)):  
 take = int(take\_list[i])  
 skip = int(skip\_list[i])  
  
 if skip\_index + take > len(non\_numbers):  
 take = len(non\_numbers) - skip\_index  
  
 for i in range(take):  
 result += non\_numbers[skip\_index + i]  
  
 skip\_index += int(take) + skip  
  
print(result)

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| T2exs15ti23ng1\_3cT1h3e0\_Roppe | TestingTheRope |
| O{1ne1T2021wf312o13Th111xreve!!@! | OneTwoThree!!! |
| this forbidden mess of an age rating 0127504740 | hidden message |

## Kate's Way Out

*Kate is stuck into a maze. You should help her to find her way out.*

On the **first line** you will be given how many **rows** there are in the maze. On the **next n lines** you will be given the **maze itself**. Here is a legend for the maze:

* **"#"** - means a **wall**; Kate cannot go through there
* **" "** - means **empty** space; Kate can go through there
* **"k"** - the initial **position of Kate**; start looking for a way out from there

There are two options: Kate either gets out or not. If Kate **can get** out print the following:   
**"Kate got out in {number\_of\_moves} moves"**. Otherwise, print: **"Kate cannot get out"**.

def find\_position(maze):  
 position = []  
 for row in range(len(maze)):  
 for el in maze[row]:  
 if el == 'k':  
 position.append(row)  
 position.append(maze[row].find('k'))  
 return position  
  
  
def next\_free\_spot(maze):  
 free\_spots = []  
  
 for row in range(len(maze)):  
 for el in range(len(maze[row])):  
 tmp = []  
 if maze[row][el] == ' ':  
 tmp.append(row)  
 tmp.append(el)  
 free\_spots.insert(0, tmp)  
  
 return free\_spots  
  
  
def find\_path(position, next\_free, maze):  
 is\_blocked = True  
 step = 0  
 moves = 0  
  
 while step < len(next\_free):  
 x1 = next\_free[step][0]  
 x2 = next\_free[step][1]  
 temp = []  
 temp.append(x1)  
 temp.append(x2)  
 # moving left  
 if temp[0] == position[0] and position[1] - temp[1] == 1:  
 position = temp  
 moves += 1  
 next\_free.pop(step)  
 step = 0  
 # moving right  
 elif temp[0] == position[0] and temp[1] - position[1] == 1:  
 position = temp  
 moves += 1  
 next\_free.pop(step)  
 step = 0  
 # moving down  
 elif temp[0] - position[0] == 1 and position[1] == temp[1]:  
 position = temp  
 moves += 1  
 next\_free.pop(step)  
 step = 0  
 # moving up  
 elif position[0] - temp[0] == 1 and position[1] == temp[1]:  
 position = temp  
 moves += 1  
 next\_free.pop(step)  
 step = 0  
  
  
 else:  
  
 step += 1  
  
 if position[0] == 0 or position[0] == (len(maze) - 1) or position[1] == 0 or position[1] == len(maze[0]):  
 return f'Kate got out in {moves + 1} moves'  
 return f'Kate cannot get out'  
  
  
m\_rows = int(input())  
maze = []  
moves = 0  
free\_space = True  
for row in range(m\_rows):  
 maze.append(input())  
position = find\_position(maze)  
next\_free = next\_free\_spot(maze)  
movement = find\_path(position, next\_free, maze)  
print(movement)

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4  ######  ## k#  ## ###  ## ### | Kate got out in 5 moves |
| 5  ######  ## k#  ## ###  ######  ## ### | Kate cannot get out |

## Battle Ships

You will be given a number **n** representing the number of **rows of the field**. On the next **n** lines you will receive **each row** of the field as a **strin**g with **numbers separated by a space**. Each number greater than zero represents a **ship** with a **health** equal to the **number value**. After that you will receive the **squares** that are being **attacked** in the format: **"{row}-{col} {row}-{col}"**. Each time a square is being attacked, if there is a ship (number greater than 0) you should **reduce its value by 1**. If a ship's health **reaches zero**, it is **destroyed**. After the attacks have ended, print **how many ships** were **destroyed**.

rows\_count = int(input())  
destroyed\_ships\_count, ships = 0, []  
  
for \_ in range(rows\_count):  
 current\_ships = [int(n) for n in input().split()]  
 ships.append(current\_ships)  
  
command = input().split()  
  
for i in range(len(command)):  
 row = int(command[i][0])  
 col = int(command[i][2])  
  
 if ships[row][col] != 0:  
 ships[row][col] -= 1  
 if ships[row][col] == 0:  
 destroyed\_ships\_count += 1  
  
print(destroyed\_ships\_count)

### Example

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 3  1 0 0 1  2 0 0 0  0 3 0 1  0-0 1-0 2-1 2-1 2-1 1-1 2-1 | 2 | States after each attack:  First attack -> 1 ship destroyed  0 0 0 1  2 0 0 0  0 3 0 1  Second attack -> reduce ship health  0 0 0 1  1 0 0 0  0 2 0 1  Third attack -> reduce ship health  0 0 0 1  2 0 0 0  0 2 0 1  Fourth attack -> reduce ship health  0 0 0 1  2 0 0 0  0 1 0 1  Fifth attack -> another ship destroyed  0 0 0 1  2 0 0 0  0 0 0 1  Sixth and Seventh attack -> no ship destroyed |

## Dots

You will be given a number **n** representing the number of **rows of a board of dots and dashes**. On the next **n** lines you will receive **each row** of the board as a **strin**g with symbols (dots and dashes only), separated by a **single space**.

Your task is to find and print the **largest count of** **dots** that could be connected **at once**. You could only connect **horizontally or vertically.**

def check\_chain(x, y):  
 if field[x][y] and not visited[x][y]:  
 visited[x][y] = True  
 return check\_chain(x - 1, y) + check\_chain(x, y + 1) + check\_chain(x + 1, y) + check\_chain(x, y - 1) + 1  
 else:  
 return 0  
  
  
rows, field, points, visited, max\_length = int(input()), [], [], [], 0  
  
# import data and set an empty frame around matrix  
for i in range(rows):  
 tempo = ('- ' + input() + ' -').split(' ')  
 morph = [1 if x == '.' else 0 for x in tempo]  
 field.append(morph)  
field.append([0] \* len(field[0]))  
field.insert(0, [0] \* len(field[0]))  
  
# identify all starting dots  
for i in range(1, len(field) - 1):  
 for j in range(1, len(field[0]) - 1):  
 if field[i][j]:  
 points.append([i, j])  
  
# create another matrix, mirroring dimensions of the field - to be used for marking visited  
for row in range(len(field)):  
 tempo = []  
 for col in range(len(field[0])):  
 tempo.append(False)  
 visited.append(tempo)  
  
# iterating over the starting dots, to explore chain lengths and getting max length  
for p in points:  
 current\_chain\_length = check\_chain(p[0], p[1])  
 if current\_chain\_length > max\_length:  
 max\_length = current\_chain\_length  
print(max\_length)

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5  . . - - -  . . - - -  - - - - -  - - - . .  - - - . . | 4 |
| 6  . . - . - .  - . . . . .  - . - - - -  - . . - - -  - . . . . -  - - - . . - | 18 |
| 4  - . - . . –  . - . . - .  . - - - - -  - - - . - - | 4 |