# Exercise: Lists Advanced

Problems for exercise and homework for the [Python Fundamentals Course @SoftUni](https://softuni.bg/trainings/3204/python-fundamentals-january-2021). Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/1731>

## Which Are In?

Given **two lists** of strings print a **new list** of the strings that contains **words** from the **first list** which are **substrings** of **any of the strings** in the **second** list (**only unique** values)

### Input

There will be **2 lines** of input: the **two lists** separated by **", "**

### Output

Print the resulting list on the console

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| arp, live, strong  lively, alive, harp, sharp, armstrong | ['arp', 'live', 'strong'] |
| tarp mice bull  lively alive harp sharp armstrong | [] |

## Big Numbers Lover

*You really like big numbers, so you always find a way to form one from numbers given to you*

You will receive a single line containing numbers separated by a single space. **Sort** and **reverse** the numbers then return **them as a string**.

### Example

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 3 30 34 5 9 | 9534303 | The numbers sorted are 9 5 34 30 3 |
| 1 2 3 | 321 |  |

### Hint

* Search in the internet how to **sort list** of **string** in python
* Search in the internet how to sort in **reversed order**

## Next Version

*You're fed up about changing the version of your software manually. Instead, you will create a little script that will make it for you.*

You will be given a **version** as in this example: **"1.3.4"**. You have to find the **next version** and **print it** (**"1.3.5"** from the example). The only **rule** is that the numbers cannot be **greater than 9**. If that happens, set the **current number to 0** and **increase the number before it**. For more clarification, see the examples. ***Note: there will be no case where the first number will get greater than 9***

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1.2.3 | 1.2.4 |
| 1.3.9 | 1.4.0 |
| 3.9.9 | 4.0.0 |

## Office Chairs

*So you've found a meeting room - phew! You arrive there ready to present, and find that someone has taken one or more of the chairs!! You need to find some quick.... check all the other meeting rooms to see if all of the chairs are in use.*

You will be given a number **n** representing how **many rooms** there are. On the next **n lines** for each room you will get how many **chairs** there are and how many of them **will be taken**. The chairs will be represented by **"X"**s, then there will be a space **" "** and a **number** representing the **taken places**. ***Example:*** **"XXXXX 4"** (**5 chairs** and **1** of them is **left free**). **Keep track of the free chairs**, you will need them later. However if you get to a room where there are **more people than chairs**, print the following message: **"{needed\_chairs\_in\_room} more chairs needed in room {number\_of\_room}"**. If there is **enough chairs in each room** print: **"Game On, {total\_free\_chairs} free chairs left"**

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4  XXXX 4  XX 1  XXXXXX 3  XXX 3 | Game On, 4 free chairs left |
| 3  XXXXXXX 5  XXXX 5  XXXXXX 8 | 1 more chairs needed in room 2  2 more chairs needed in room 3 |

## Electron Distribution

*You are a mad scientist and you decided to play with electron distribution among atom's shells. You know that basic idea of electron distribution is that electrons should fill a shell until it's holding the maximum number of electrons.*

The **rules** for electron distribution are as follows:

* Maximum number of electrons in a shell is distributed with a rule of **2n^2** (**n** being **position** of a **shell** a.k.a. the list **index + 1**).
* For example, maximum number of electrons in **3rd** shield is **2\*3^2 = 18**.
* Electrons should fill the **lowest level** shell **first**.
* If the electrons have **completely filled** the **lowest level** shell, the other **unoccupied electrons** will fill the **higher level** shell and so on.

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| 10 | [2, 8] |
| 44 | [2, 8, 18, 16] |

## Group of 10's

Write a program that receives a **list of numbers** (string containing **integers** separated by **", "**) and **prints lists** with the numbers them into lists of **10's**.

**Examples**:

* The numbers **2 8 4 3** fall into the group under **10**
* The numbers **13 19 14 15** fall into the group under **20**

For more details, see the examples below

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| 8, 12, 38, 3, 17, 19, 25, 35, 50 | Group of 10's: [8, 3]  Group of 20's: [12, 17, 19]  Group of 30's: [25]  Group of 40's: [38, 35]  Group of 50's: [50] |
| 1, 3, 3, 4, 34, 35, 25, 21, 33 | Group of 10's: [1, 3, 3, 4]  Group of 20's: []  Group of 30's: [25, 21]  Group of 40's: [34, 35, 33] |

### Hints

* **Keep track of the group** using a variable to store it's **max value**
* Create a **loop** and **filter the elements** that are less than the group boundary and **remove** them from the **original list**
* **Increase** the **boundary by 10**
* **Loop until** the given **list is empty**

## Decipher This!

You are given a **secret message** you need to **decipher**. Here are the things you need to know to decipher it:

For **each word**:

* the **second** and the **last letter** are **switched** (e.g. Hello becomes Holle)
* the **first letter** is **replaced** by its **character code** (e.g. H becomes 72)

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| 72olle 103doo 100ya | Hello good day |
| 82yade 115te 103o | Ready set go |

## \* Moving Target

You are at the shooting gallery again and you need a program that helps you keep track of moving targets. On the first line, you will receive a **sequence of targets with their integer values**, split by a **single space**. Then, you will start receiving **commands for manipulating the targets**, until the **"End"** command. The commands are the following:

* **Shoot {index} {power}**
  + Shoot the target at the index, **if it exists** by **reducing** its **value** by the **given** **power** (**integer value**).A target is considered **shot** when **its value reaches 0**.
  + Remove the target, **if it is shot**.
* **Add {index} {value}**
  + Insert a target with the received value at the received **index, if it exist**. If not, print: **"Invalid placement!"**
* **Strike {index} {radius}**
  + Remove the **target at the given index** and the **ones before and after it depending on the radius, if such exist. If any of the indices in the range is invalid print:**

**"Strike missed!" and skip this command.**

**Example:** **Strike 2 2**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | {radius} | {radius} | {strikeIndex} | {radius} | {radius} |  |  |

* **End**
  + Print the sequence with targets in the following format:

**{target1}|{target2}…|{targetn}**

### Input / Constraints

* On the **first line** you will receive **the sequence of targets** – **integer values [1-10000]**.
* On the **next lines,** until the **"End"** will be receiving the command described above – **strings**.
* There will never be a case when **"Strike"** command would empty the whole sequence.

### Output

* Print the appropriate message in case of **"Strike"** command if necessary.
* In the end, print the sequence of targets in the format described above.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 52 74 23 44 96 110  Shoot 5 10  Shoot 1 80  Strike 2 1  Add 22 3  End | Invalid placement!  52|100 | The first command is "**Shoot**", so we reduce the target on **index** **5**, which is valid, with the given **power** – **10**.  Then we receive the same command but we need to reduce the target on the 1st index, with power 80. The value of this target is 74, so it is considered shot and we **remove** it.  Then we receive the "**Strike**" command on the 2nd index and we need to check if the range with radius 1 is valid:  **52 23 44 96 100**  And it is, so we **remove** the targets.  At last we receive the "**Add**" command, but the index is **invalid** so we print the appropriate **message** and in the end we have the following result:  **52|100** |
| 47 55 85 78 99 20  Shoot 1 55  Shoot 8 15  Strike 2 3  Add 0 22  Add 2 40  Add 2 50  End | Strike missed!  22|47|50|40|85|78|99|20 |  |

## \* Heart Delivery

*Valentine’s Day is coming, and Cupid has very limited time to spread some love across the neighborhood. Help him with his mission!*

You will receive a **string** with **even integers,** separated by a **"@".** This is our neighborhood. After that a series of **Jump** commands will follow, until you receive **"Love!"** Every house in the neighborhood needs a certain number of **hearts** delivered by Cupid, in order to be able to celebrate Valentine’s Day. Those needed hearts are indicated by the integers in the neighborhood.

Cupid starts at the position of the **first** **house** (index 0) and must jump by a **given length.** The jump commands will be in this format: **"Jump {length}"**.

Every time he jumps from one house to another, the needed hearts for the visited house are **decreased by 2**. If the needed hearts for a certain house become **equal to 0** , print on the console **"Place {houseIndex} has Valentine's day."** If **Cupid** jumps to a house where the needed hearts are **already** **0,** print on the console"**Place {houseIndex} already had Valentine's day.**".

Keep in mind that **Cupid** can have a **bigger jump length** than the **size of the neighborhood** and if he does jump **outside** of it, he should **start** from the **first house** again**.**

*For example, we are given this neighborhood: 6@6@6. Cupid is at the start and jumps with a length of 2. He will end up at index 2 and decrease the needed hearts there by 2: [6, 6, 4]. Next he jumps again with a length of 2 and goes outside the neighborhood, so he goes back to the first house (index 0) and again decreases the needed hearts there: [4, 6, 4].*

### Input

* On the first line you will receive a **string** with **even integers** separated by **"@"** –the neighborhood and the number of hearts for each house.
* On the next lines, until "**Love!**" is received, you will be getting jump commands in this format: "**Jump {length}**".

### Output

At the end print **Cupid's** **last position** and whether his mission was successful or not:

* "**Cupid's last position was {lastPositionIndex}.**"
* If **each house** has had a Valentine's day, print:
  + "**Mission was successful.**"
* If **not,** print the **count** of all houses that **didn`t** celebrate a Valentine's Day:
  + **"Cupid has failed {houseCount} places."**

### Constraints

* The **neighborhood`s** size will be in the range [1…20]
* Each **house** will need an **even number** of hearts in the range [2 … 10]
* Each **jump length** will be an integer in the range [1 … 20]

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 10@10@10@2  Jump 1  Jump 2  Love! | Place 3 has Valentine's day.  Cupid's last position was 3.  Cupid has failed 3 places. | Jump 1 ->> [10, 8, 10, 2]  Jump 2 ->> [10, 8, 10, 0] so we print "Place 3 has Valentine's day."  Next command is "Love!", so we print Cupid`s last position and the outcome of his mission. |
| 2@4@2  Jump 2  Jump 2  Jump 8  Jump 3  Jump 1  Love! | Place 2 has Valentine's day.  Place 0 has Valentine's day.  Place 0 already had Valentine's day.  Place 0 already had Valentine's day.  Cupid's last position was 1.  Cupid has failed 1 places. |  |

## \* Inventory

*As a young traveler, you gather items and craft new items.*

### Input / Constraints

You will receive a journal with some Collecting items, separated with **', '** (comma and space). After that, until receiving "Craft!" you will be receiving different commands.

Commands (split by **" - "**):

* "Collect - {item}" – Receiving this command, you should add the given item in your inventory. If the item already **exists**, you should **skip** this line.
* "Drop - {item}" – You should remove the item from your inventory, **if it exists**.
* "Combine Items - {oldItem}:{newItem}" – You should check if the **old item exists**, if so, **add** the new item **after** the old one. Otherwise, **ignore** the command.
* "Renew – {item}" – If the given item exists, you should change its position and **put it last** in your inventory.

### Output

After receiving "Craft!" print the items in your inventory, separated by **", "** (comma and space).

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Iron, Wood, Sword  Collect - Gold  Drop - Wood  Craft! | Iron, Sword, Gold |
| **Input** | **Output** |
| Iron, Sword  Drop - Bronze  Combine Items - Sword:Bow  Renew - Iron  Craft! | Sword, Bow, Iron |