# Exercise: Lists Advanced

Problems for exercise and homework for the [Python Fundamentals Course @SoftUni](https://softuni.bg/trainings/2442/python-fundamentals-september-2019). 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. Form the biggest number possible from them by **sorting them as strings**.

### 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 |

## \*\* Feed the Animals

*The sanctuary needs to provide food for the animals and feed them, so your task is to help with the process*

Create a program that organizes the **daily feeding** of **animals**. You need to keep information about **animals**, their **daily food limit** and the **areas** of the Wildlife Refuge **they** **live** **in**. You will be receiving **lines** with commands until you receive the **"Last Info"** message. There are two **possible** commands:

* **"Add:{animalName}:{dailyFoodLimit}:{area}":**
  + **Add** the **animal** and **its** **daily food limit** to your records. It is guaranteed that the **names** of the animals are **unique** and there will **never** be animals with the **same** name. **If** it already **exists**, just increase the value of the **daily** **food** **limit** with the **current** one that is **given**.
* **"Feed:{animalName}:{food}:{area}":**
  + **Check** if the animal **exists** and if **it does**, **reduce** its daily **food limit** with the given **food** **for** **feeding**. If its **limit** reaches **0** or **less**, the **animal** is considered **successfully fed** and you need to **remove** it from your **records** and **print** the following **message**:
    - **"{animalName} was successfully fed"**

You need to know **the count of** **hungry** **animals** there are left in **each area** in the end. If an animal has daily food **limit above 0**, it is considered **hungry**.

In the end, you have to **print each animal** with its **daily** food **limit** sorted in **descending order** by the **daily food limit** and **then by** its **name** in **ascending** order in the following format:

**Animals:**

**{animalName} -> {dailyFoodLimit}g**

**{animalName} -> {dailyFoodLimit}g**

Afterwards, **print** the **areas** with the **count of animals**, which are **not** **fed** in **descending** order by the **count** of **animals.** If an **area** has **0** **hungry animals** in it, **don't** print it. The **output** must be in the following **format**:

**Areas with hungry animals:**

**{areaName} : {countOfUnfedAnimals}**

**{areaName} : {countOfUnfedAnimals}**

### Input / Constraints

* You will be receiving linesuntil you receive the **"Last Info"** command.
* The **food** comes in **grams** and is an **integer** number in the range [1...100000].
* The input will **always** be **valid**.
* There will never be a case, in which an animal is in two or more areas at the same time.

### Output

* Print the appropriate message after the **"Feed"** command, **if** an **animal** is **fed**.
* Print the animals with their **daily food limit** in the **format** described above.
* Print the **areas** with the **count of unfed** **animals** in them in the **format** described above.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Add:Maya:7600:WaterfallArea  Add:Bobbie:6570:DeepWoodsArea  Add:Adam:4500:ByTheCreek  Add:Jamie:1290:RiverArea  Add:Gem:8730:WaterfallArea  Add:Maya:1230:WaterfallArea  Add:Jamie:560:RiverArea  Feed:Bobbie:6300:DeepWoodsArea  Feed:Adam:4650:ByTheCreek  Feed:Jamie:2000:RiverArea  Last Info | Adam was successfully fed  Jamie was successfully fed  Animals:  Maya -> 8830g  Gem -> 8730g  Bobbie -> 270g  Areas with hungry animals:  WaterfallArea : 2  DeepWoodsArea : 1 |
| **Comments** | |
| First, we receive the "**Add**" command, so we **add** "**Maya**" to our **records** and we keep her **daily food limit** - **7600**. We know that she is in **WaterfallArea**. We keep adding the new animals until we receive "**Maya**" **again** and we have to **increase** her food **limit** with **1230**, so it becomes **8830**. After that we receive "**Jamie**" and we need to **increase** his daily food **limit** with **560**, after which it **becomes** **1850**. Then we start receiving "**Feed**" commands. First, we must **decrease** **Bobbie's** food **limit** with **6300**, so it becomes **270**. Then, we need to decrease **Adam's** food **limit** with **4650**. It **becomes** **less than zero** and we **remove** **him** from the collection – he is **considered fed**, respectively that is **one less hungry** **animal** in the **area** that he is in – **ByTheCreek**. Then we "**Feed**" **Jamie** with **2000** and his **limit** becomes **less than zero**, so we print "**Jamie was successfully fed**" and we **remove** him from our records and note that there is **one** **less** **hungry animal** in his area – **RiverArea**. In the end, we **print the animals** we still have in our collection, with their daily food **limits** in **descending order** by the food **limits**. Afterwards we print only the **areas** in which there are **remaining** **hungry** **animals** and their **count** in **descending** order. | |
|  | |
| Add:Bonie:3490:RiverArea  Add:Sam:5430:DeepWoodsArea  Add:Bonie:200:RiverArea  Add:Maya:4560:ByTheCreek  Feed:Maya:2390:ByTheCreek  Feed:Bonie:3500:RiverArea  Feed:Johny:3400:WaterFall  Feed:Sam:5500:DeepWoodsArea  Last Info | Sam was succesfully fed  Animals:  Maya -> 2170g  Bonie -> 190g  Areas with hungry animals:  RiverArea : 1  ByTheCreek : 1 |

## \*\* On the Way to Annapurna

*You’ve hired a Sherpa and he has a list of supplies you both need to go on the way. He has passed you some notes and you have to order them correctly in a diary before you start circling around the town’s stores.*



Create a program, that lists **stores** and the **items** that can be found in them. You are going to be receiving **commands** with the information you need until you get the "**End**" command. There are **three possible commands**:

* "**Add**->{Store}->{Item}"
  + **Add** the **store** and the **item** in your diary. If the store already **exists**, add just the item.
* **"Add**->{Store}->{Item},{Item1}…,{ItemN}"
  + **Add the store and the items to** your notes. **If the store already exists** in the diary – **add just the items** to it.
* "**Remove**->{Store}"
  + **Remove the store** and its items from your diary, **if it exists**.

In the end, print the collection **sorted by the count of the items** in **descending order** and **then by the names of the stores**, again, **in descending order** in the following format:

**Stores list:**

**{Store}**

**<<{Item}>>**

**<<{Item}>>**

**<<{Item}>>**

### Input / Constraints

* You will be receiving information until the “**END**” command is given.
* There will always be **at least one** store in the diary.
* Input will always be **valid**, there is no need to check it explicitly.

### Output

* Print the list of stores in the format given above.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Add->PeakSports->Map,Navigation,Compass  Add->Paragon->Sunscreen  Add->Groceries->Dried-fruit,Nuts  Add->Groceries->Nuts  Add->Paragon->Tent  Remove->Paragon  Add->Pharmacy->Pain-killers  END | Stores list:  PeakSports  <<Map>>  <<Navigation>>  <<Compass>>  Groceries  <<Dried-fruit>>  <<Nuts>>  <<Nuts>>  Pharmacy  <<Pain-killers>> |
| **Comments** | |
| First, we receive the "**Add**" command with a couple of items and we have to add the store and the items to. We keep doing that for each line of input and when we receive the "**Remove**" command, we delete the store and its items from our records. In the end we print the stores sorted by the **count** of their **items** and **then by** their **names**. | |
|  | |
| Add->Peak->Waterproof,Umbrella  Add->Groceries->Water,Juice,Food  Add->Peak->Tent  Add->Peak->Sleeping-Bag  Add->Peak->Jacket  Add->Groceries->Lighter  Remove->Groceries  Remove->Store  END | Stores list:  Peak  <<Waterproof>>  <<Umbrella>>  <<Tent>>  <<Sleeping-Bag>>  <<Jacket>> |

## \*\* Man O War

Create a program that **tracks** the **battle** and either chooses a **winner** or prints a **stalemate**. On the **first line** you will receive the **status** of the **pirate ship**, which is a **string** representing **integer sections** separated by **'>'**. On **the second line** you will receive the **same** type of status, but for the **warship**:

**"{section1}>{section2}>{section3}… {sectionn}"**

On the **third line** you will receive the **maximum health capacity** a section of the ship can reach.

The following lines represent commands **until** **"Retire"**:

* **Fire {index} {damage} –** the pirate ship **attacks** the warship with the **given damage** at that section. Check if the **index is valid** and if not **skip** the command. If the section **breaks** (health <= 0) the warship **sinks**, print the following and **stop** the program:

**"You won! The enemy ship has sunken."**

* **Defend {startIndex} {endIndex} {damage} -** the warship **attacks** the pirate ship with the **given damage** at that **range** (**indexes are inclusive)**. Check if both **indexes are valid** and if not **skip** the command. If the section **breaks** (health <= 0) the pirate ship **sinks**, print the following and **stop** the program:

**"You lost! The pirate ship has sunken."**

* **Repair {index} {health} -** the crew **repairs** a section of the **pirate ship** with the **given health**. Check if the **index is valid** and if not **skip** the command. The health of the section **cannot** exceed the **maximum health capacity**.
* **Status –** prints the **count** of all sections of the **pirate ship** that need repair soon, which are all sections that are **lower than 20%** of the **maximum** **health capacity**. Print the following:

**"{count} sections need repair."**

In the end if a **stalemate** occurs print the **status** of **both** ships, which is the **sum** of their individual sections in the following format:

**"Pirate ship status: {pirateShipSum}"**

**"Warship status: {warshipSum}"**

## Input

* On the **1st line** you are going to receive the **status** of the **pirate ship** (**integers** separated by **'>'**)
* On the **2nd line** you are going to receive the **status** of the **warship**
* On the **3rd line** you are going receive the **maximum health** a section of a ship can reach.
* On the next **lines**, until **"Retire"**, you will be receiving commands.

## Output

* Print the output in the **format** **described** **above**.

## Constraints

* The **section numbers** will be integers in the range [**1**….**1000**]
* The **indexes** will be integers [**-200**….**200**]
* The **damage** will be an integer in the range [**1**….**1000**]
* The **health** will be an integer in the range [**1**….**1000**]

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 12>13>11>20>66  12>22>33>44>55>32>18  70  Fire 2 11  Fire 8 100  Defend 3 6 11  Defend 0 3 5  Repair 1 33  Status  Retire | 2 sections need repair.  Pirate ship status: 135  Warship status: 205 |
| **Comments** | |
| First, we receive the command "**Fire 2 11**" and damage the warship at section index 2 which is currently 33 and after reduction the status of the warship is the following:  **12 22 22 44 55 32 18**  The **second** and **third** command have **invalid indexes**, so we skip them.  The **fourth** command **"Defend 0 3 5"** damages **4 sections** of the pirate ship with **5** which results in the following status:  **7 8 6 15 66**  The **fifth** command **"Repair 1 33"** repairs the pirate ship section and adds **33 health** to the current **8** which results in **41**  Only **2 sections** of the pirate ship (**7** and **6**) need repair soon.  In the end there is a **stalemate,** so we print both ship statuses (**sum** of all sections). | |
| **Input Output** | |
| 2>3>4>5>2  6>7>8>9>10>11  20  Status  Fire 2 3  Defend 0 4 11  Repair 3 18  Retire | 3 sections need repair.  You lost! The pirate ship has sunken. |