# Exercise: Lists Advanced

Problems for exercise and homework 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/1731>.

## Which Are In?

You will be given **two sequences** of strings, separated by **", "**. Print a **new list** containing only the **strings** from the **first input line** which are **substrings** of **any string** in the **second input line**.

first\_input\_list = input().split(', ')  
second\_input\_text = input()  
  
output\_list = [x for x in first\_input\_list if x in second\_input\_text]  
  
print(output\_list)

### Example

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

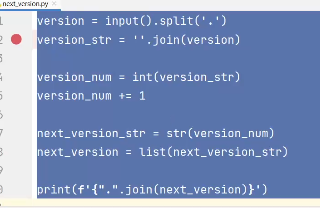
## Next Version

*You are fed up with 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 string representing the **version** of your software in the format: **"{n1}.{n2}.{n3}"**. Your task is to **print** the **next version.** For example, if the current version is **"1.3.4"**, the next version will be **"1.3.5"**.

The only **rule** is that the numbers cannot be **greater than 9**. If it happens, set the **current number to 0** and **increase the previous number**. For more clarification, see the examples below.

***Note: there will be no case in which the first number will become greater than 9.***

******

num = int(input().replace(".", ""))  
new\_version = num + 1  
  
print('.'.join(str(new\_version)))

### Example

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

## Word Filter

Using a **comprehension**, write a program that receives some **text**, separated by **space**, and take only those words, whose length is **even**. Print each word on a new line.

input\_list = input().split()  
output\_list = []  
  
for word in input\_list:  
 if len(word) % 2 == 0:  
 output\_list.append(word)  
  
# output\_words = ' '.join(str(x) for x in output\_list)  
  
for i in range(len(output\_list)):  
 print(output\_list[i])

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| kiwi orange banana apple | kiwi  orange  banana |
| pizza cake pasta chips | cake |

## Number Classification

Using a **list comprehension**, write a program that receives **numbers**, separated by comma and space **", "**, and prints all the **positive**, **negative**, **even,** and **odd** numbers on separate lines as shown below.

***Note: Zero is counted for a positive number***

input\_list = input().split(', ')  
input\_ints = list(map(int, input\_list))  
positive = []  
negative = []  
even = []  
odd = []  
  
for num in input\_ints:  
 if num >= 0:  
 positive.append(num)  
 if num < 0:  
 negative.append(num)  
 if num % 2 == 0:  
 even.append(num)  
 if num % 2 != 0:  
 odd.append(num)  
  
positive\_str = ", ".join(str(x) for x in positive)  
negative\_str = ", ".join(str(x) for x in negative)  
even\_str = ", ".join(str(x) for x in even)  
odd\_str = ", ".join(str(x) for x in odd)  
  
print(f"Positive: {positive\_str}")  
print(f"Negative: {negative\_str}")  
print(f"Even: {even\_str}")  
print(f"Odd: {odd\_str}")

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1, -2, 0, 5, 3, 4, -100, -20, 12, 19, -33 | Positive: 1, 0, 5, 3, 4, 12, 19  Negative: -2, -100, -20, -33  Even: -2, 0, 4, -100, -20, 12  Odd: 1, 5, 3, 19, -33 |

## Office Chairs

*You are a facility manager at a large business center and one of your responsibilities is to check if each conference room in the center have enough chairs for the visitors.*

On the first line, you will be given an integer **n** representing **the number of rooms in the business center**. On the next **n lines** for each room, you will receive information about the **chairs** in the room and the **number of visitors**. Each **chair** will be presented with the char **"X"**. Next, there will be a **single space** and at the end, the **number** of **visitors**. For example: **"XXXXX 4"** (**5 chairs** and **4 visitors**).

Keep track of the free chairs:

* If there are not enough chairs in a certain room, print the following message: **"{needed\_chairs\_in\_room} more chairs needed in room {number\_of\_room}"**.
* Otherwise, print: **"Game On, {total\_free\_chairs} free chairs left"**

n = int(input())  
  
count\_free\_seats = 0  
needed\_chairs = 0  
number\_of\_room = 1  
  
for i in range(0, n):  
 chairs, people = input().split()  
  
 if len(chairs) > int(people):  
 free\_seats = len(chairs) - int(people)  
 count\_free\_seats += free\_seats  
 elif len(chairs) < int(people):  
 needed\_chairs = abs(len(chairs) - int(people))  
 count\_free\_seats -= needed\_chairs  
 print(f"{needed\_chairs} more chairs needed in room {number\_of\_room}")  
  
 number\_of\_room += 1  
  
if count\_free\_seats >= 0:  
 print(f"Game On, {count\_free\_seats} 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 |
| 3  XXXXXXX 5  XXXXXX 8  XXXX 5 |  |

## Electron Distribution

*You are a mad scientist and you have 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.*

You will receive a single integer – **number of electrons**. Your task is to **fill shells** **until there are no more electrons** **left**. The **rules** for electron distribution are as follows:

* The maximum number of electrons in a shell can be **2n2**, where **n** is the **position** of a **shell** (**starting from 1**). For example, the maximum number of electrons in the **3rd** shield can be **2\*32 = 18**.
* You should start **filling** the shells from the **first one** at the first position.
* If the electrons are enough to **completely fill** the **first** shell, the left **unoccupied electrons** should fill the **next** shell and so on.

At the end, **print a list with the filled shells**.

total\_electrons = int(input())  
  
electron\_config = []  
current\_electron\_layer = 1  
  
while total\_electrons > 0:  
 electrons\_per\_layer = 2 \* pow(current\_electron\_layer, 2)  
  
 if total\_electrons >= electrons\_per\_layer:  
 electron\_config.append(electrons\_per\_layer)  
 else:  
 electron\_config.append(total\_electrons) # Add the remaining electrons.  
  
 total\_electrons -= electrons\_per\_layer  
  
 current\_electron\_layer += 1 # After each iteration, increase the number of layers !  
  
print(electron\_config)

### Example

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

## Group of 10's

Write a program which receives a **sequence of numbers** (string containing **integers** separated by **", "**) and **prints** the **numbers sorted into lists** of **10's** in the format **"Group of {group}'s: {list\_of\_numbers}"**.

**Examples**:

* The numbers **2, 8, 4 and 10** fall into the group of **10's**.
* The numbers **13, 19, 14 and 15** fall into the group of **20's**.

For more clarification, see the examples below.

initial\_group = list(map(int, input().split(", ")))  
group = 10  
  
while len(initial\_group) > 0:  
 list\_of\_numbers = [x for x in initial\_group if x <= group]  
 print(f"Group of {group}'s: {list\_of\_numbers}")  
 initial\_group = [x for x in initial\_group if x not in list\_of\_numbers]  
 list\_of\_numbers = []  
 group += 10

### 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 or equal to 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 should **decipher**. To do that**,** you need to know that **in each word**:

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

def replace\_character(word: str):  
 char\_code\_str = ''  
 new\_word = []  
  
 for char in word:  
 if char.isnumeric():  
 char\_code\_str += char  
 else:  
 new\_word.append(char)  
  
 char\_at\_beginning = chr(int(char\_code\_str))  
 new\_word.insert(0, char\_at\_beginning)  
 return new\_word  
  
  
def decipher(word: str):  
 new\_word = replace\_character(word)  
 new\_word[1], new\_word[-1] = new\_word[-1], new\_word[1]  
 return ''.join(new\_word)  
  
  
input\_words = input().split()  
  
deciphered\_words = []  
  
for item in input\_words:  
 deciphered\_words.append(decipher(item))  
  
print(" ".join(deciphered\_words))

### 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.

def is\_index\_valid(collection: list, index: int):  
 if 0 <= index < len(collection):  
 return True  
  
 return False  
  
  
def shoot\_at\_target(collection: list, target\_index: int, power: int):  
 if is\_index\_valid(collection, target\_index):  
 collection[target\_index] -= power  
  
 if collection[target\_index] <= 0:  
 collection.pop(target\_index)  
  
 return collection  
  
  
def add\_at\_index(collection: list, append\_index: int, value: int):  
 if not is\_index\_valid(collection, append\_index):  
 print('Invalid placement!')  
  
 return collection  
  
 collection.insert(append\_index, value)  
  
 return collection  
  
  
def strike\_at\_index(collection: list, strike\_index: int, radius: int):  
 left\_strike\_index = strike\_index - radius  
 right\_strike\_index = strike\_index + radius  
  
 if is\_index\_valid(collection, left\_strike\_index) and is\_index\_valid(collection, strike\_index) and is\_index\_valid(  
 collection, right\_strike\_index):  
 # Perform a strike  
 for i in range(left\_strike\_index, right\_strike\_index + 1):  
 collection.pop(left\_strike\_index)  
  
 # del collection[left\_strike\_index:right\_strike\_index + 1]  
 else:  
 print('Strike missed!')  
  
 return collection  
  
  
numbers\_str\_list = input().split()  
numbers = [int(num) for num in numbers\_str\_list]  
  
command = input()  
while command != "End":  
 cmd\_args = command.split()  
  
 cmd\_type = cmd\_args[0]  
 cmd\_index = int(cmd\_args[1])  
 cmd\_value = int(cmd\_args[2])  
  
 if cmd\_type == "Shoot":  
 numbers = shoot\_at\_target(numbers, cmd\_index, cmd\_value)  
 elif cmd\_type == "Add":  
 numbers = add\_at\_index(numbers, cmd\_index, cmd\_value)  
 elif cmd\_type == "Strike":  
 numbers = strike\_at\_index(numbers, cmd\_index, cmd\_value)  
  
 command = input()  
  
numbers\_str\_list = [str(num) for num in numbers]  
print('|'.join(numbers\_str\_list))

### 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]

neighborhood = list(map(int, input().split('@')))  
length\_neighborhood = len(neighborhood)  
house = 0  
while True:  
 command = input()  
 if command == 'Love!':  
 break  
 cupidon = command.split()  
 jump\_length = int(cupidon[1])  
 house += jump\_length  
 if house >= length\_neighborhood:  
 house = 0;  
 if neighborhood[house] == 0:  
 print(f"Place {house} already had Valentine's day.")  
 else:  
 neighborhood[house] -= 2  
 if neighborhood[house] == 0:  
 print(f"Place {house} has Valentine's day.")  
  
print(f"Cupid's last position was {house}.")  
count = 0  
for house in range(len(neighborhood)):  
 if neighborhood[house] > 0:  
 count += 1  
  
if count == 0:  
 print("Mission was successful.")  
else:  
 print(f"Cupid has failed {count} places.")

### 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.*

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 |

def check\_item\_exist(check\_item, all\_items):  
 all\_items = set(all\_items)  
 if check\_item in all\_items:  
 return True  
 else:  
 return False  
  
  
items = input().split(", ")  
  
command = input()  
  
while not command == "Craft!":  
 type\_command = command.split(" - ")[0]  
 item = command.split(" - ")[1]  
 is\_exist = check\_item\_exist(item, items)  
 if type\_command == "Collect" and not is\_exist:  
 items.append(item)  
 elif type\_command == "Drop" and is\_exist:  
 items.remove(item)  
 elif type\_command == "Combine Items":  
 old\_item = item.split(":")[0]  
 new\_item = item.split(":")[1]  
 is\_exist = check\_item\_exist(old\_item, items)  
 if is\_exist:  
 index = items.index(old\_item)  
 items.insert(index + 1, new\_item)  
 elif type\_command == "Renew" and is\_exist:  
 index = items.index(item)  
 x = items.pop(index)  
 items.append(x)  
 command = input()  
  
print(", ".join(items))