# Exercise: Dictionaries, Lambdas and Functional Programming

Problems for exercises and homework for the [“Python Fundamentals” course @ SoftUni](https://softuni.bg/opencourses/python-fundamentals-course).

You can check your solutions here: <https://judge.softuni.bg/Contests/945/>.

## Odd Occurrences

Write a program that extracts from a given sequence of words all elements that present in it **odd number of times** (case-insensitive).

* Words are given in a single line, space separated.
* Print the result elements in lowercase, in their order of appearance.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Java C# PHP PHP JAVA C java | java, c#, c |
| 3 5 5 hi pi HO Hi 5 ho 3 hi pi | 5, hi |
| a a A SQL xx a xx a A a XX c | a, SQL, xx, c |

### Hints

* Use a **dictionary** (string 🡪 int) to count the occurrences of each word (just like in the previous problem).
* Pass through all **key-value pairs** in the dictionary and append to the results list all **keys** that have **odd value**.
* Print the results list.

## Count Real Numbers

Read a **list of real numbers** and **print them in ascending order** along with their **number of occurrences**.

### Examples

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |  | **Input** | **Output** |
| 8 2.5 2.5 8 2.5 | 2.5 -> 3 times  8 -> 2 times | 1.5 5 1.5 3 | 1.5 -> 2 times  3 -> 1 times  5 -> 1 times | -2 0.33 0.33 2 | -2 -> 1 times  0.33 -> 2 times  2 -> 1 times |

### Hints

* Use dictionary (key=nums, value=count) named counts.
* Pass through each input number num and increase counts[num] (when num exists in the dictionary) or assign counts[num] = 1 (when num does not exist in the dictionary).
* Pass through all numbers num in the dictionary (counts.keys()) and print the number num and its count of occurrences counts[num].

## Letter Repetition

You will be given a **single string**, containing **random ASCII character**. Your task is to **print every character**, and how **many times** it has been used in the **string**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| aaabbaaabbbccc | a -> 6  b -> 5  c -> 3 |
| The quick brown fox jumps over the lazy dog | T -> 1  h -> 2  e -> 3  -> 8  q -> 1  u -> 2  i -> 1  c -> 1  k -> 1  b -> 1  r -> 2  o -> 4  w -> 1  n -> 1  f -> 1  x -> 1  j -> 1  m -> 1  p -> 1  s -> 1  v -> 1  t -> 1  l -> 1  a -> 1  z -> 1  y -> 1  d -> 1  g -> 1 |

## Dict-Ref

You have been tasked to create a referenced dictionary, or in other words a dictionary, which knows how to reference itself.

You will be given several input lines, in one of the following formats:

* {name} = {value}
* {name} = {secondName}

The **names** will always be **strings**, and the **values** will always be **integers**.

In case you are given a **name** and a **value**, you must store the **given name** and its **value**. If the name already **EXISTS**, you must **CHANGE** its **value** with the **given one**.

In case you are given a **name** and a **second name**, you must **store** the **given name** with the **same value** as the **value** of the **second name**. If the given **second name** **DOES NOT** exist, you must **IGNORE** that input.

When you receive the command “**end**”, you must print **all** **entries** with their **value**, by **order** of **input**, in the following format:

{entry} === {value}

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Cash = 500  Johny = 5  Cash = 200  Johnny = 200  Car = 150  Plane = 2000000  end | Cash === 200  Johny === 5  Johnny === 200  Car === 150  Plane === 2000000 |
| Entry1 = 10000  Entry2 = 12345  Entry3 = 10101  Entry4 = Entry1  Entry2 = Entry3  Entry3 = Entry4  end | Entry1 === 10000  Entry2 === 10101  Entry3 === 10000  Entry4 === 10000 |

## Mixed Phones

You will be given several phone entries, in the form of strings in format:

{firstElement} : {secondElement}

The first element is usually the person’s name, and the second one – his phone number. The phone number consists ONLY of digits, while the person’s name can consist of any ASCII characters.

Sometimes the phone operator gets distracted by the Minesweeper she plays all day, and gives you first the phone, and then the name. **e.g. : 0888888888 : Pesho**. You must store them correctly, even in those cases.

When you receive the command “**Over**”, you are to **print all names** you’ve stored with their phones. The names must be printed in **alphabetical order**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 14284124 : Alex  Gosho : 088423123  Ivan : 412048192  123123123 : Nanyo  Pesho : 150925812  Over | Alex -> 14284124  Gosho -> 88423123  Ivan -> 412048192  Nanyo -> 123123123  Pesho -> 150925812 |
| Ivan : 13213456  GeorgeThe2nd : 131313  Johnny : 5556322312  Donald : 3212  Over | Donald -> 3212  GeorgeThe2nd -> 131313  Ivan -> 13213456  Johnny -> 5556322312 |

## Exam Shopping

A supermarket has **products** which have **quantities**. Your task is to stock the shop before **Exam Sunday**. Until you receive the command “shopping time”, **add** the various **products** to the shop’s **inventory**, keeping track of their **quantity** (for inventory purposes). When you receive the aforementioned command, the students start pouring in before the exam and **buy** various **products**.

The format for **stocking** a product is: “stock {product} {quantity}”.

The format for **buying** a product is: “buy {product} {quantity}”.

If a student **tries** to buy a product, which **doesn’t exist**, print “{product} doesn't exist”. If it does exist, but it’s **out of stock**, print “{product} out of stock”. If a student tries buying **more** than the quantity of that product, sell them **all** **the** **quantity** of the product (the product is left out of stock, **don’t** print anything).

When you receive the command “exam time”, your task is to **print** the **remaining** inventory in the following format: “{product} -> {quantity}”. If a product is out of stock, **do not** print it.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| stock Boca\_Cola 10  stock Boca\_Cola 10  stock Kay's 3  stock Kay's 2  shopping time  buy Boca\_Cola 5  buy Kay's 5  exam time | Boca\_Cola -> 15 |
| stock Lobster\_Energy 20  stock Loreni 30  stock Loreni 30  stock Lobster\_Energy 10  shopping time  exam time | Lobster\_Energy -> 30  Loreni -> 60 |
| stock Boca\_Cola 16  stock Kay's\_Chips 33  stock Lobster\_Energy 60  stock Boca\_Cola 4  stock Loreni 15  stock Loreni 15  stock Loreni 15  stock Loreni 15  shopping time  buy Boca\_Bola 2  buy Lobster\_Energy 20  buy Boca\_Cola 1  buy Boba\_Bola 12  exam time | Boca\_Bola doesn't exist  Boba\_Bola doesn't exist  Boca\_Cola -> 19  Kay's\_Chips -> 33  Lobster\_Energy -> 40  Loreni -> 60 |

## User Logins

Write a program that receives a **list** of **username-password pairs** in the format “{username} -> {password}”. If there’s already a user with that username, **replace their password**. After you receive the command “login”, **login requests** start coming in, using the **same format**. Your task is to print the status of user login, using different messages as per the conditions below:

* If the password matches with the user’s password, print “{username}: logged in successfully”.
* If the user doesn’t exist, or the password doesn’t match the user, print “{username}: login failed”.

When you receive the command “end”, print the **count** of **unsuccessful** login attempts, using the format “unsuccessful login attempts: {count}”.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| ivan\_ivanov -> java123  pesh0 -> qwerty  stamat4e -> 111111  princess\_penka -> MyPrince  **login**  pesh0 -> qwertt  ivan\_ivanov -> java123  stamat4e -> 111112  princess\_penka -> MyPrince  end | pesh0: login failed  ivan\_ivanov: logged in successfully  stamat4e: login failed  princess\_penka: logged in successfully  unsuccessful login attempts: 2 |
| johnny\_bravo05 -> woahMama  **login**  johnny\_bravo06 -> woahMama  johnny\_bravo05 -> woahmama  johnny\_bravo05 -> WoahMama  johnny\_bravo05 -> wohMama  johnny\_bravo05 -> woahMama  end | johnny\_bravo06: login failed  johnny\_bravo05: login failed  johnny\_bravo05: login failed  johnny\_bravo05: login failed  johnny\_bravo05: logged in successfully  unsuccessful login attempts: 4 |
| walter\_white58 -> iamthedanger  krazy\_8 -> ese  jesseABQ -> bword  **login**  krazy\_8 -> ese  krazy\_8 -> ese  jesse -> bword  jesseabq -> bword  walter\_white58 -> IAmTheDanger  walter\_white58 -> iamthedanger  end | krazy\_8: logged in successfully  krazy\_8: logged in successfully  jesse: login failed  jesseabq: login failed  walter\_white58: login failed  walter\_white58: logged in successfully  unsuccessful login attempts: 3 |

### Hints

* **Parse the commands** by splitting by space. The first element is the username, while the third element is the password.
* Store the **user entries** in **dictionary** with key **{username}** and value **{password}**.

## Filter Base

You have been tasked to sort out a database full of information about employees. You will be given several input lines containing information in one of the following formats:

* {name} -> {age}
* {name} -> {salary}
* {name} -> {position}

As you see you have 2 parameters. There can be only 3 cases of input:  
If the second parameter is an **integer**, you must store it as **name** and **age**.

If the second parameter is a **floating-point number**, you must store it as **name** and **salary**.

If the second parameter is a **string**, you must store it as **name** and **position**.

You must read input lines, then parse and store the information from them, **until** you receive the command   
“**filter base**”. When you receive that command, the **input sequence** of **employee information** should **end**.

On the last line of input you will receive a string, which can either be “**Age**”, “**Salary**” or “**Position**”. Depending on the case, you must **print all entries** which **have been stored** as **name** and **age**, **name** and **salary**, or **name** and **position**.

In case, the given last line is “**Age**”, you must print every employee, stored with its **age** in the following format:

Name: {name1}  
Age: {age1}  
====================  
Name: {name2}  
. . .

In case, the given last line is “**Salary**”, you must print every employee, stored with its **salary** in the following format:

Name: {name1}  
Salary: {salary1}  
====================  
Name: {name2}  
. . .

In case, the given last line is “**Position**”, you must print every employee, stored with its **position** in the following format:

Name: {name1}  
Position: {position1}  
====================  
Name: {name2}  
. . .

**NOTE:** Every entry is **separated** with the **other**, with a **string** of **20 character** **‘=**’.

There is **NO** particular order of printing – the data must be printed in **order** of **input**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Isacc -> 34  Peter -> CEO  Isacc -> 4500.054321  George -> Cleaner  John -> Security  Nina -> Secretary  filter base  Position | Name: Peter  Position: CEO  ====================  Name: George  Position: Cleaner  ====================  Name: John  Position: Security  ====================  Name: Nina  Position: Secretary  ==================== |
| Ivan -> Chistach  Pesho -> Ohrana  Pesho -> 1323.0456  Ivan -> 732.404  Georgi -> 21  Georgi -> 21.02  filter base  Salary | Name: Pesho  Salary: 1323.0456  ====================  Name: Ivan  Salary: 732.404  ====================  Name: Georgi  Salary: 21.02  ==================== |

### Hints:

Use int(), float() and try-except structure to find out in **which case** are you and where to **store** the **data**.

## Wardrobe

You just bought a new wardrobe. The weird thing about it is that it came prepackaged with a big box of clothes (just like those refrigerators, which ship with free beer inside them)! So, you’ll need to find something to wear.

### Input

On the first line of the input, you will receive n – the **number of lines** of clothes, which came prepackaged for the wardrobe.

On the next n lines, you will receive the clothes for each color in the format:

* “{color} -> {item1},{item2},{item3}…”

If a color is added a **second** time, **add** **all items** from it and **count** the **duplicates**.

**Finally**, you will receive the **color** and **item** of the clothing, that you need to look for.

### Output

Go through all the **colors** of the clothes and print them in the following format:

|  |
| --- |
| **{color}** clothes:  \* **{item1}** - **{count}**  \* **{item2}** - **{count}**  \* **{item3}** - **{count}**  …  \* **{itemN}** - **{count}** |

If the **color** lines up with the **clothing item**, print “(found!)” alongside the item. See the examples to better understand the output.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4  Blue -> dress,jeans,hat  Gold -> dress,t-shirt,boxers  White -> briefs,tanktop  Blue -> gloves  Blue dress | Blue clothes:  \* dress - 1 (found!)  \* jeans - 1  \* hat - 1  \* gloves - 1  Gold clothes:  \* dress - 1  \* t-shirt - 1  \* boxers - 1  White clothes:  \* briefs - 1  \* tanktop - 1 |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4  Red -> hat  Red -> dress,t-shirt,boxers  White -> briefs,tanktop  Blue -> gloves  White tanktop | Red clothes:  \* hat - 1  \* dress - 1  \* t-shirt - 1  \* boxers - 1  White clothes:  \* briefs - 1  \* tanktop - 1 (found!)  Blue clothes:  \* gloves - 1 |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5  Blue -> shoes  Blue -> shoes,shoes,shoes  Blue -> shoes,shoes  Blue -> shoes  Blue -> shoes,shoes  Red tanktop | Blue clothes:  \* shoes - 9 |

## Shellbound

Vladi is a crab. Crabs are scared of almost anything, which is why they usually hide in their shells. Vladi is a rich crab though. He has acquired many outer shells, in different regions, in which he can hide – each with a different size.

However, it is really annoying to look after all your shells when they are so spread out. That is why Vladi decided to **merge** **all** shells in **each** region, so that he has **one** **big** **shell** for **every** **region**. He needs your help to do that.

You will be given several input lines containing a **string** and an **integer**, **separated** by a **space**. The **string** will represent the **region’s name**, and the **integer** – the shell in the **given region**, **size**.

You must store all shells in their corresponding regions.  
If the region **already exists**, **add** the **new shell** to it. Make sure you have **NO** duplicate shells (shells with **same size**). Vladi doesn’t like shells to have the same size.

When you receive the command “Aggregate”, you must stop reading input lines, and you must print **every region**, **all of the shells** in that region, and the **size** of the **giant shell** after you’ve merged them in the following format:

{regionName} -> {shell 1, shell 2…, shell n…} ({giantShell})

The giant shell size is calculated when you **subtract** the **average** of the shells from the **sum** of the shells.  
Or in other words: (sum of shells) – ((sum of shells) / (count of shells)).

### Constraints

* All numeric data will be represented with **integer** variables in **range** [0…1.000.000.000].

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Sofia 50  Sofia 20  Sofia 30  Varna 10  Varna 20  Aggregate | Sofia -> 50, 20, 30 (67)  Varna -> 10, 20 (15) |
| Sofia 2  Sofia 1  Plovdiv 100  Plovdiv 50  Aggregate | Sofia -> 2, 1 (2)  Plovdiv -> 100, 50 (75) |