Exercise: Functions Advanced

This document defines the exercises for the "Python Advanced" course at @SoftUni Global.

Please, submit your source code solutions for the described problems to the Judge System.

1. Negative vs Positive

You will receive a sequence of **numbers** (integers) separated by a **single space**. **Separate** the negative numbers from the positive. Find the **total sum of the negatives and positives**, and **print the following:**

- On the first line, **print the sum** of the negatives
- On the second line, **print the sum** of the positives
- On the third line:
  - If the **absolute negative number** is larger than the **positive number**:

    `"The negatives are stronger than the positives"`
  - If the **positive number** is larger than the **absolute negative number**:

    `"The positives are stronger than the negatives"`

Note: you will not receive any zeroes in the input.

Example

Input	Output

`1 2 -3 -4 65 -98 12 57 -84`	`-189`
	`137`
	`The negatives are stronger than the positives`

`1 2 3`	`0`
	`6`
	`The positives are stronger than the negatives`

2. Keyword Arguments Length

Create a function called **`kwargs_length()`** that can receive some **keyword arguments** and **return** their **length**.

Submit only your function in the judge system.

Examples

Test Code	Output

`dictionary = {'name': 'Peter', 'age': 25}`
 	
`print(kwargs_length(**dictionary))`
	`2`

`dictionary = {}`
 	
`print(kwargs_length(**dictionary))`
	`0`

3. Even or Odd

Create a function called **even_odd()** that can receive a different **quantity of numbers** and a **command** at the end. The command can be **"even"** or **"odd"**. **Filter** the numbers depending on the command and **return** them in a **list**. Submit only the function in the judge system.

Submit only your function in the judge system.

Examples

Test Code	Output
`print(even_odd(1, 2, 3, 4, 5, 6, "even"))`	`[2, 4, 6]`
`print(even_odd(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, "odd"))`	`[1, 3, 5, 7, 9]`

4. Numbers Filter

Create a function called **even_odd_filter()** that can receive a different **number of named arguments**. The keys will be **"even" and/or "odd"**, and the values will be a **list of numbers**.

When the key is **"odd", you should extract** only the **odd numbers** from its list. When the key is **"even", you should extract** only the **even numbers** from its list.

The function should **return a dictionary** sorted by the **length of the lists** in **descending** order. There will be no case of lists with the same length.

Submit only your function in the judge system.

Example

Input	Output

`print(even_odd_filter(`
`  odd=[1, 2, 3, 4, 10, 5],`
`  even=[3, 4, 5, 7, 10, 2, 5, 5, 2],`
`))`
	`{'even': [4, 10, 2, 2],`
	`'odd': [1, 3, 5]}`

`print(even_odd_filter(`
`  odd=[2, 2, 30, 44, 10, 5],`
`))`
	`{'odd': [5]}`

5. Concatenate

Write a **concatenate()** function that **receives** some **strings as arguments** and some **named arguments** (the **key** will be a **string,** and the **value** will be another **string**).

First, you should concatenate all arguments successively. Next, take each key successively, and if it is present in the resulting string, **change all matching parts** with the key's **value**. In the end, **return** the **final string**.

See the examples for more clarification.

Submit only your function in the judge system.

Examples

Test Code	Output

`print(concatenate("Soft", "UNI", "Is", "Grate", "!", UNI="Uni", Grate="Great"))`	`SoftUniIsGreat!`
`print(concatenate("I", " ", "Love", " ", "Cythons", C="P", s="", java='Java'))`	`I Love Python`

6. Function Executor

Create a function called **`func_executor()`** that can receive a different number of **tuples**, each of which will have exactly **2 elements**: the first will be a **function,** and the second will be a **tuple of the arguments** that need to be passed to that function. You should execute **each function** and **return its result** in the format:

`"{function name} - {function result}"`

For more clarification, see the examples below.

Submit only your function in the judge system.

Examples

Test Code	Output

`def sum_numbers(num1, num2):`
`  return num1 + num2`
 	
`def multiply_numbers(num1, num2):`
`  return num1 * num2`
 	
`print(func_executor(`
`  (sum_numbers, (1, 2)),`
`  (multiply_numbers, (2, 4))`
`))`
	`sum_numbers - 3`
	`multiply_numbers - 8`

`def make_upper(*strings):`
`  result = tuple(s.upper() for s in strings)`
`  return result`
 	
`def make_lower(*strings):`
`  result = tuple(s.lower() for s in strings)`
`  return result`
 	
`print(func_executor(`
`  (make_upper, ("Python", "softUni")),`
`  (make_lower, ("PyThOn",)),`
`))`
	`make_upper - ('PYTHON', 'SOFTUNI')`
	`make_lower - ('python', )`

7. Grocery

Create a function called **`grocery_store()`** that receives a different number of **key-value** pairs. The **key** will be the **product's name** and the **value** - its **quantity**.

You should **return** a **sorted receipt** for all products. They should be sorted by their **quantity** in **descending order**. If there are two or more products with the **same quantity**, sort them by their **name's length** in **descending order**. If there are two or more products with the **same name's length**, sort them by **their name** in **ascending order** (alphabetically). In the end, **return a string** in the following format:

`"{product_name1}: {product_quantity1}`

{product_name2}: {product_quantity2}

...

{product_nameN}: {product_quantityN}"

Examples

Test Code	Output

`print(grocery_store(`
`  bread=5,`
`  pasta=12,`
`  eggs=12,`
`))`
	`pasta: 12`
	`eggs: 12`
	`bread: 5`

`print(grocery_store(`
`  bread=2,`
`  pasta=2,`
`  eggs=20,`
`  carrot=1,`
`))`
	`eggs: 20`
	`bread: 2`
	`pasta: 2`
	`carrot: 1`

8. Age Assignment

Create a function called **`age_assignment()`** that receives a different number of **names** and a different number of **key-value** pairs. The **key** will be a **single letter** (the first letter of each name) and the **value** - a **number** (age). Find its **first letter** in the **key-value** pairs for each name and **assign** the **age to the person's name**.

Then, sort the **names** in **ascending order (alphabetically)** and **return** the information for each person **on a new line** in the format: **`"{name} is {age} years old."`**

Submit only the function in the judge system.

Examples

Test Code	Output
`print(age_assignment("Peter", "George", G=26, P=19))`	`George is 26 years old.` `Peter is 19 years old.`

`print(age_assignment("Amy", "Bill", "Willy", W=36, A=22, B=61))`	`Amy is 22 years old.`
	`Bill is 61 years old.`
	`Willy is 36 years old.`

9. Recursion Palindrome

Write a **recursive** function called **`palindrome()`** that will receive a **word** and an **index** (**always 0**). Implement the function, so it returns **`"{word} is a palindrome"`** if the word is a palindrome and **`"{word} is not a palindrome"`** if the word is not a palindrome using **recursion**. Submit only the function in the judge system.

Examples

Test Code	Output
`print(palindrome("abcba", 0))`	`abcba is a palindrome`
`print(palindrome("peter", 0))`	`peter is not a palindrome`

10. *Fill the Box

Write a function called **`fill_the_box`** that receives a different number of arguments representing:

- the **height** of a box
- the **length** of a box
- the **width** of a box
- different numbers - each representing the quantity of **cubes**. Each **cube** has an exact **size** of **1 x 1 x 1**
- a string **`"Finish"`**

Your task is to fill the box with the given cubes **until the current argument equals `"Finish"`**.

Note: Submit only the function in the judge system

Input

- There will be **no input**. Just parameters passed to your function.

Output

The function should return a string in the following format:

- If, in the end, there is free space left in the box, print:
  - **`"There is free space in the box. You could put {free space in cubes} more cubes."`**
- If there is no free space in the box, print:
  - **`"No more free space! You have {cubes left} more cubes."`**

Examples

Test Code	Output	Comment
`print(fill_the_box(2, 8, 2, 2, 1, 7, 3, 1, 5, "Finish"))`	`There is free space in the box. You could put 13 more cubes.`	The size of the box: 2 * 8 * 2 = 32 We put the cubes consistently. In the end, there is more free space left.
`print(fill_the_box(5, 5, 2, 40, 11, 7, 3, 1, 5, "Finish"))`	`No more free space! You have 17 more cubes.`	The size of the box: 5 * 5 * 2 = 50 We put the cubes consistently. First, we put 40 cubes and there is free space left. Then we try to put 11 cubes, but there is only space for 10. Cubes left: 1 + 7 + 3 + 1 + 5 = 17
`print(fill_the_box(10, 10, 10, 40, "Finish", 2, 15, 30))`	`There is free space in the box. You could put 960 more cubes.`	

11. *Math Operations

Write a function named **`math_operations`** that receives a different number of **floats** as arguments and 4 keyword arguments. The keys will be single letters: **`"a"`**, **`"s"`**, **`"d"`**, and **`"m"`**, and the values will be numbers.

You need to take **each float argument** from the sequence and do mathematical operations as follows:

- The **first** element should be **added** to the value of the key **`"a"`**

- The **second** element should be **subtracted** from the value of the key **"s"**
- The **third** element should be **divisor** to the value of the key **"d"**
- The **fourth** element should be **multiplied** by the value of the key **"m"**
- Each **result** should **replace the value** of the corresponding key
- You must **repeat** the same steps **consecutively** until you run out of numbers

Beware: **You cannot divide by 0**. If the operation **could throw an error**, you should **skip the operation** and **continue to the next one**.

After you finish calculating all numbers, **sort** the four elements by **their values** in **descending order**. If **two or more values are equal**, sort them by their **keys** in **ascending** order (**alphabetically**).

In the end, **return** each **key-value pair** in the format **"{key}: {value}"** on **separate lines**. Each value should be formatted to the **first decimal point**.

For more clarifications, see the examples below.

Note: Submit only the function in the judge system

Input

- There will be **no input**. Just parameters passed to your function.
- All of the given numbers will be valid integers in the range [-100, 100]

Output

- The function should **return the final dictionary**

Examples

Test Code	Output

`print(math_operations(2.1, 12.56, 0.0, -3.899, 6.0, -20.65, a=1, s=7, d=33, m=15))`	`d: 33.0`
	`s: 15.1`
	`a: 9.1`
	`m: -58.5`

`print(math_operations(-1.0, 0.5, 1.6, 0.5, 6.1, -2.8, 80.0, a=0, s=(-2.3), d=0, m=0))`	`a: 5.1`
	`d: 0.0`
	`m: 0.0`
	`s: 0.0`

`print(math_operations(6.0, a=0, s=0, d=5, m=0))`	`a: 6.0`
	`d: 5.0`
	`m: 0.0`
	`s: 0.0`