# Exercise: Functions Advanced

Problems for exercise and homework for the [Python Advanced Course @SoftUni](https://softuni.bg/courses/python-advanced).

Submit your solutions in the SoftUni judge system at <https://judge.softuni.org/Contests/1839>.

## 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 |

## Odd or Even

On the first line, you will receive a **command - "Odd"** or **"Even"**. You will receive a sequence of **numbers** (integers) on the second line, separated by a **single space**.

* If the command is **"Odd"**, print the **sum of the odd** numbers **multiplied** by the **count of all numbers**.
* If the command is **"Even"**, print the **sum of the even** numbers **multiplied** by the **count of all numbers**.

### Example

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| Odd  1 3 5 34 7 9 12 11 13 10 | 490 | The sum of all odd numbers is: **1 + 3 + 5 + 7 + 9 + 11 + 13 = 49**.  Multiply the sum: **49 \* 10 = 490.** |
| Even  1 3 5 7 9 13 | 0 |  |

## Arguments Length

Create a function called args\_length() that returns the **number of the arguments**. Submit only the function in the judge system.

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| print(args\_length(1, 32, 5)) | 3 |
| print(args\_length("john", "peter")) | 2 |
| print(args\_length([1, 2, 3])) | 1 |

## Concatenate

Write a function called concatenate() that **receives** some strings, **concatenates** them, and **returns** the result.

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| print(concatenate("Soft", "Uni", "Is", "Great", "!")) | SoftUniIsGreat! |
| print(concatenate("I", " ", "Love", " ", "Python")) | I Love Python |

## 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.

### 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] |

## Function Executor

Create a function called func\_executor() that can receive a different number of **tuples**, each of which will have exactly **2 elements**: first will be a **function,** and the second will be a **tuple of the arguments** that need to be passed to that function. Create a **list** that will contain all the **results** of the **executed functions** with their corresponding **arguments** and **return** itafter executing all functions. 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)))) | [3, 8] |

## Keyword Arguments Length

Create a function called kwargs\_length() that can receive some **keyword arguments** and **return** their **length**. Submit only the 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 |

## 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**. It the end, **return a dictionary** with all the **names and ages** as shown in the example. Submit only the function in the judge system.

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| print(age\_assignment("Peter", "George", G=26, P=19)) | {'Peter': 19, 'George': 26} |
| print(age\_assignment("Amy", "Bill", "Willy", W=36, A=22, B=61)) | {'Amy': 22, 'Bill': 61, 'Willy': 36} |

## 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 |

## Recursive Power

Create a **recursive** function called recursive\_power() which should receive a **number** and a **power**. Using **recursion,** **return** the result of **number \*\* power**. Submit only the function in the judge system.

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| print(recursive\_power(2, 10)) | 1024 |
| print(recursive\_power(10, 100)) | 10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 |

## \*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
* n-times a different number of cubes with exact **size 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, at 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. At 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. |  |

## \*Math Operations

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

You need to take **each integer 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 **delete the element** from the sequence and **continue to the next operation**.

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

* Thefunction should **return the final dictionary**

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| print(math\_operations(2, 12, 0, -3, 6, -20, -11, a=1, s=7, d=33, m=15)) | {'a': 9, 's': 15, 'd': -3.0, 'm': -45} |
| **Comment** | |
| We create 1 args list: **[2, 12, 0, -3, 6, -20, -11]** and 1 kwargs dict: **{'a': 1, 's': 7, 'd': 33, 'm': 15}**. We start calculating from the first number and the first key-value:  1) 1 + 2 = 3 -> add 3 to the key **'a'**  2) 7 – 12 = -5 -> add -5 to the key **'s'**  3) 33 / 0 throws **ZeroDivisionError** -> remove 0 and continue to the next operation  4) 15 \* (-3) = (-45) ->add -45 to the key **'m'**  5) 3 + 6 = 9 -> add 3 to the key **'a'**  6) (-5) - (-20) = 15 -> add 3 to the key **'s'**  7) 33 / (-11) = (-3.0) -> add 3 to the key **'d'** | |
| print(math\_operations(-1, 0, 1, 0, 6, -2, 80, a=0, s=0, d=0, m=0)) | {'a': 5, 's': 2, 'd': 0.0, 'm': 0} |
| print(math\_operations(6, a=0, s=0, d=0, m=0)) | {'a': 6, 's': 0, 'd': 0, 'm': 0} |