# Exercise: Decorators

Problems for exercise and homework for the [Python OOP Course @SoftUni](https://softuni.bg/courses/python-oop). Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/1947>

## Logged

Create a decorator called logged. It should **return** the name of the function that is being called and its parameters. It should also return the **result of the execution** of the function being called. See the examples for more clarification.

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| @logged  def func(\*args):  return 3 + len(args)  print(func(4, 4, 4)) | you called func(4, 4, 4)  it returned 6 |
| @logged  def sum\_func(a, b):  return a + b  print(sum\_func(1, 4)) | you called sum\_func(1, 4)  it returned 5 |

### Hints

* Use {func}.\_\_name\_\_ to get the name of the function
* Call the function to get the result
* Return the result

## Type Check

Create a decorator called type\_check. It should receive a type (int**/**float**/**str**/…**) and it should check if the parameter passed to the decorated function is of the **type** given to the decorator. If it is, **execute** the function and **return the result**, otherwise **return "**Bad Type**"**.

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| @type\_check(int)  def times2(num):  return num\*2  print(times2(2))  print(times2('Not A Number')) | 4  Bad Type |
| @type\_check(str)  def first\_letter(word):  return word[0]  print(first\_letter('Hello World'))  print(first\_letter(['Not', 'A', 'String'])) | H  Bad Type |

## Cache

Create a decorator called cache. It should store all the returned values of e **recursive function** fibonacci. You are provided with this code:

**def** cache(func):  
 *#* ***TODO: Implement***@cache  
**def** fibonacci(n):  
 **if** n < 2:  
 **return** n  
 **else**:  
 **return** fibonacci(n-1) + fibonacci(n-2)

You need to create a **dictionary** called log that will store all the **n**'s (**keys**) and the **returned results** (**values**) and **attach** that dictionary to the fibonacci function as a variable called **log**, so when you call it, it returns that dictionary. For more clarification, see the examples

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| fibonacci(3)  print(fibonacci.log) | {1: 1, 0: 0, 2: 1, 3: 2} |
| fibonacci(4)  print(fibonacci.log) | {1: 1, 0: 0, 2: 1, 3: 2, 4: 3} |

## HTML Tags

Create a decorator called tags. It should receive an html tag as a parameter, **wrap** the result of a function with the given tag and **return the new result**. For more clarification, see the examples below

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| @tags('p')  def join\_strings(\*args):  return "".join(args)  print(join\_strings("Hello", " you!")) | <p>Hello you!</p> |
| @tags('h1')  def to\_upper(text):  return text.upper()  print(to\_upper('hello')) | <h1>HELLO</h1> |

## Execution Time

Create a decorator called exec\_time. It should calculate how much **time** a function needs to be **executed**. See the examples for more clarification.

***Note: You might have different results from the given ones. Only the functionality of the code will be checked in this problem***

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| @exec\_time  def loop(start, end):  total = 0  for x in range(start, end):  total += x  return total  print(loop(1, 10000000)) | 0.8342537879943848 |
| @exec\_time  def concatenate(strings):  result = ""  for string in strings:  result += string  return result  print(concatenate(["a" for i in range(1000000)])) | 0.14537858963012695 |

### Hints

* Use the time library to start a timer
* Execute the function
* Stop the timer and return the result