Exercise: Decorators

Problems for exercise and homework for the Python OOP Course @SoftUni.
Submit your solutions in the SoftUni judge system at https://judge.softuni.org/Contests/1947/Decorators-Exercise.

1. 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

2. Even Parameters

Create a decorator function called **`even_parameters`**. It should check if **all parameters** passed to a function are **even numbers** and only then **execute** the function and **return** the result. Otherwise, **don't execute** the function and return **`"Please use only even numbers!"`**

Examples

Test Code	Output

`@even_parameters`
`def add(a, b):`
`    return a + b`
 	
`print(add(2, 4))`
`print(add("Peter", 1))`
	`6`
	`Please use only even numbers!`

`@even_parameters`
`def multiply(*nums):`
`    result = 1`
	`384`
	`Please use only even numbers!`

```
    for num in nums:
        result *= num
    return result

print(multiply(2, 4, 6, 8))
print(multiply(2, 4, 9, 8))
```

3. Bold, Italic, Underline

Create **three decorators**: **make_bold**, **make_italic**, **make_underline**, which will have to **wrap** a **text** returned from a function in ****, **** and **<u></u>** **respectively**.

Examples

Test Code	Output

`@make_bold`
`@make_italic`
`@make_underline`
`def greet(name):`
`  return f"Hello, {name}"`
 	
`print(greet("Peter"))`
	`<b><i><u>Hello, Peter</u></i></b>`

`@make_bold`
`@make_italic`
`@make_underline`
`def greet_all(*args):`
`  return f"Hello, {', '.join(args)}"`
 	
`print(greet_all("Peter", "George"))`
	`<b><i><u>Hello, Peter, George</u></i></b>`

Note: Submit all the decorator functions in the judge system

4. 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`

5. Cache

Create a decorator called **cache**. It should store all the returned values of the **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}`

6. 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>`

7. *Store Results

Create a **class** called **`store_results`**. It should be used as a **decorator** and **store information** about the executed functions in a **file** called **`results.txt`** in the format: **`"Function {func_name} was called. Result: {func_result}"`**

Note: The solutions to this problem cannot be submitted in the judge system

Examples

Test Code	results.txt

`@store_results`
`def add(a, b):`
`    return a + b`
 	
`@store_results`
`def mult(a, b):`
`    return a * b`
 	
`add(2, 2)`
`mult(6, 4)`
	`Function 'add' was called. Result: 4`
	`Function 'mult' was called. Result: 24`

8. Execution Time

Import the **`time`** module. 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. The solutions to this problem cannot be submitted in the judge system.

Examples

Test Code	Output

`@exec_time`
`def loop(start, end):`
`    total = 0`
`    for x in range(start, end):`
`        total += x`
	`0.8342537879943848`

`    return total` `print(loop(1, 10000000))`	

`@exec_time`
`def concatenate(strings):`
`    result = ""`
`    for string in strings:`
`        result += string`
`    return result`
`print(concatenate(["a" for i in range(1000000)]))`
	`0.14537858963012695`

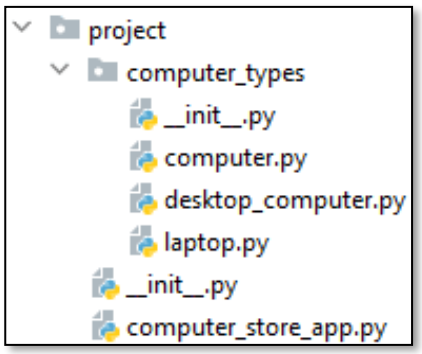
`@exec_time`
`def loop():`
`    count = 0`
`    for i in range(1, 9999999):`
`        count += 1`
`print(loop())`
	`0.4199554920196533`

Hints

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

*9. Computer Store

For this task, you will be provided with a **skeleton** that includes all the folders and files you need.



Note: You cannot change the folder and file structure and their names!

Judge Upload

Create a **`zip`** file with the **`project` folder** and **upload it** to the judge system.

You do not need to include **in the zip file** your **venv**, **`.idea`**, **pycache**, and **__MACOSX** (for Mac users), so you do not exceed **the maximum allowed size** of **`16.00` KB**.

Description

Your friend is the owner of one of the best computer stores in the world. Recently he started building computers, and he asked you as a programmer to create a program for his store so that he can track the computer's building process and the sale process. Your app should have the following structure and functionality.

1. Class Computer

In the **computer.py** file, the class **Computer** should be implemented. It is a **base class** for any **type of computer,** and it **should not be able to be instantiated**.

Structure

The class should have the following attribute:

- **manufacturer: str**
  - A string that represents the **manufacturer's name**.
  - If the string is **empty or contains only whitespaces**, raise **ValueError** with the message: **"Manufacturer name cannot be empty."**
- **model: str**
  - A string that represents the **computer's model name**.
  - If the string is **empty or contains only whitespaces**, raise **ValueError** with the message: **"Model name cannot be empty."**
- **processor: str**
  - A string that represents the **computer's processor**.
  - Should be **set to None** upon initialization
- **ram: int**
  - An integer that represents the **computer's RAM memory**.
  - Should be **set to None** upon initialization
- **price: int**
  - An integer that represents the **computer's price**.
  - Should be **set to 0** upon initialization

Methods

__init__(manufacturer: str, model: str)

- In the **__init__** method, all the needed attributes must be set.

configure_computer(processor: str, ram,: int)

- Every type of computer **should be configurable**
- Valid types: **"Laptop", "Desktop Computer"**

__repr__()

- Representsts the class as: **"{ manufacturer } { model } with { processor } and { ram }GB RAM"**

2. Class DesktopComputer

In the **desktop_computer.py** file, the class **DesktopComputer** should be implemented.

Methods

__init__(manufacturer: str, model: str)

- In the **__init__** method, all the needed attributes must be set.

configure_computer(processor: str, ram,: int)

- Desktop computers can be built only with the available processors for desktop computers, which are:
  - **AMD Ryzen 7 5700G: 500$**
  - **Intel Core i5-12600K: 600$**
  - **Apple M1 Max: 1800$**
- Desktop computers can have a **max RAM of 128GB**
  - Valid RAM sizes are 2, 4, 8…128. In other words, all the **powers of the number 2 to the max size**.
  - RAM **price is defined by the power of the number 2, which gives the RAM size, multiplied by 100**. ***For example: 2GB RAM will cost 100$ because 2 = 2^1 and 1 * 100 = 100. 4GB will be 200$.***
- If a processor is **not in the available processors**, raise **ValueError** with the message: **"{ processor } is not compatible with desktop computer { manufacturer name } { model name }!"**
- If **RAM is not a valid size or is above the max size**, raise **ValueError** with the message: **"{ RAM }GB RAM is not compatible with desktop computer { manufacturer name } { model name }!"**
- If **everything is valid**, attach the processor to the computer, attach the RAM, and update the price. Return the following message: **"Created { manufacturer name } { model name } with { processor } and { ram }GB RAM for { computer price }$."**

3. Class Laptop

In the **laptop.py** file, the class **Laptop** should be implemented.

Methods

__init__(manufacturer: str, model: str)

- In the **__init__** method, all the needed attributes must be set.

configure_computer(processor: str, ram: int)

- Laptops can be built only with the available processors for laptops, which are:
  - **AMD Ryzen 9 5950X: 900$**
  - **Intel Core i9-11900H: 1050$**
  - **Apple M1 Pro: 1200$**
- Laptops can have a **max RAM of 64GB**
  - Valid RAM sizes are 2, 4, 8…64. In other words, all the **powers of the number 2 to the max size**.
  - RAM **price is defined by the power of the number 2, which gives the RAM size, multiplied by 100**. ***For example: 2GB RAM will cost 100$ because 2 = 2^1 and 1 * 100 = 100. 4GB will be 200$.***
- If a processor is **not in the available processors**, raise **ValueError** with the message: **"{ processor } is not compatible with laptop { manufacturer name } { model name }!"**
- If **RAM is not a valid size or is above the max size**, raise **ValueError** with the message: **"{ RAM }GB RAM is not compatible with laptop { manufacturer name } { model name }!"**
- If **everything is valid**, attach the processor to the computer, attach the RAM, and update the price. Return the following message: **"Created { manufacturer name } { model name } with { processor } and { ram }GB RAM for { computer price }$."**

4. Class ComputerStoreApp

In the **computer_store_app.py** file, the class **ComputerStoreApp** should be implemented. It will contain all the functionality of the project.

Structure

The class should have the following attribute:

- **`warehouse: list`**
  - A list that will **store the built computers**.
  - Should be **empty** upon initialization
- **`profits: int`**
  - An integer that represents the **store profits**.
  - Should be **set to `0`** on initialization

Methods

`__init__()`

- In the **`__init__`** method, all the needed attributes must be set.

build_computer(type_computer: str, manufacturer: str, model: str, processor: str, ram: int)

- Valid types of computers are: **"Desktop Computer", "Laptop"**
- If a **computer type isn't valid**, raise **`ValueError`** with the message: **`"{ type computer } is not a valid type computer!"`**
- Otherwise, configure the computer, add it to the warehouse, and return the result from the configuration.

sell_computer(client_budget: int, wanted_processor: str, wanted_ram: int)

- **Search for a computer** in the warehouse. To sell a computer, it has to meet the following criteria:
  - Computer's price is **less than or equal to** the client's budget.
  - The computer has the **same processors** as the one requested by the client.
  - The computer's RAM is **more or equal to** the one requested by the client.
- If you **can't find a computer to sell**, raise an **Exception** with the message: **`"Sorry, we don't have a computer for you."`**
- If you **find a computer** that meets the criteria, **sell it** at the **client's budget price**, **add the difference between the sale price and the build price** to the store **profits**, and return the following message: **`"{ computer } sold for { client budget }$."`**

Examples

Input

`from project.computer_store_app import ComputerStoreApp`
 	
`computer_store = ComputerStoreApp()`
`print(computer_store.build_computer("Laptop", "Apple", "Macbook", "Apple M1 Pro", 64))`
`print(computer_store.sell_computer(10000, "Apple M1 Pro", 32))`

Output
`Created Apple Macbook with Apple M1 Pro and 64GB RAM for 1800$.` `Apple Macbook with Apple M1 Pro and 64GB RAM sold for 10000$.`

"Hey man, where are the SSDs?"