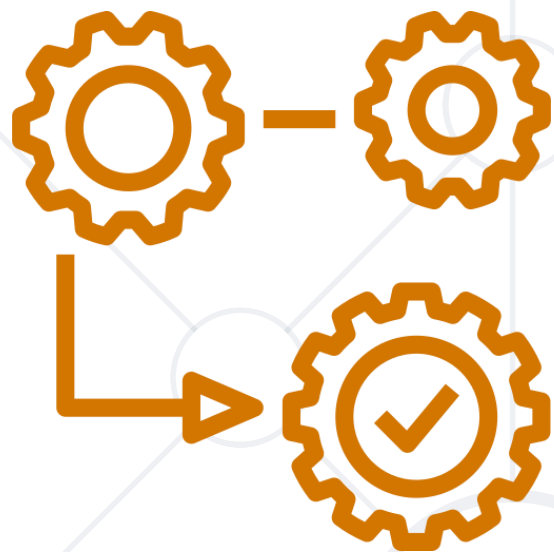


# Functions Advanced



**SoftUni Team**  
**Technical Trainers**



**SoftUni**



**Software University**

<https://softuni.bg>

[sli.do](https://sli.do)

**#python-advanced**

## 1. Packing Arguments

- **\*args** and **\*\*kwargs**

## 2. Unpacking Arguments

- Unpacking Lists and Tuples
- Unpacking Dictionaries

## 3. Recursion





**\*args**  
**\*\*kwargs**

# **Packing Arguments**

**\*args and \*\*kwargs**

# What is Packing?

```
def some_func(*args, **kwargs):  
    pass
```

- This operation is called **packing**
- We pack all the arguments into one **single variable**
- We use packing when we don't know how many arguments need to be passed to a function



# Packing Arguments into Tuple

- We use **\*args** to pack arguments into tuple

```
def some_func(*args):  
    print(args)
```

```
some_func(1, 2, 3)           # (1, 2, 3)  
some_func("peter", "george") # ("peter", "george")  
some_func(True, False)      # (True, False)  
some_func()                  # ()
```



# Packing Arguments into Dictionary

- **\*\*kwargs** allows you to pass **keyworded** variable length of arguments to a function

```
def greet_me(**kwargs):  
    for key, value in kwargs.items():  
        print(f"{value}, {key}")
```

```
greet_me(Peter="Hello", George="Bye")  
# Hello, Peter  
# Bye, George
```



# Formal Args, \*args and \*\*kwargs

- You can also use **keyword** arguments and **\*args**

```
def some_func (arg1, *rest_args):  
    print(arg1 + sum(rest_args))  
some_func(5, 5, 10)    # 20  
some_func()            # Error
```

The function requires  
at least 1 argument

- So, if you want to use all three of these in argument types then the order is

```
some_func(fargs, *args, **kwargs)
```



# Problem: Multiplication Function

- Write a function called **multiply()** that can receive any number of numbers (integers) as different parameters
- The function should return the result of the multiplication of all of them
- Submit only your function in judge

```
print(multiply(1, 4, 5))  
print(multiply(4, 5, 6, 1, 3))  
print(multiply(2, 0, 1000, 5000))
```



```
20  
360  
0
```

# Solution: Multiplication Function

```
def multiply(*args):  
    result = 1  
    for num in args:  
        result *= num  
    return result
```





\*

\* \*

# Unpacking Arguments

Unpack Lists, Tuples and Dictionaries

# What is Unpacking?

- We can use `*` to unpack the list so that all elements of it can be passed as **different parameters**
- And we can use `**` to unpack a dictionary, so all of its elements are passed as **keyworded arguments**



# Unpacking Lists


- Note that the **length** of the list, that you unpack, must be **the same** as the number of **parameters** in the function

```
def print_nums(a, b, c):  
    print(a, b, c)  
nums = [1, 2, 3]  
print_nums(*nums)      # 1 2 3
```



# Unpacking Dictionaries

- Note that the **keys** of the dictionary must **match** the **names** of the **parameters** of the function
- The **order** of the keys in the dictionary does **not matter**



```
def some_func(name, age):  
    print(f"{name} is {age} years old")  
person = {'age': 20, 'name': "Peter"}  
some_func(**person) # Peter is 20 years old
```

# Problem: Person Info

- Write a function called **get\_info** that receives a name, age and town, and returns a string in the format:  
**"This is {name} from {town} and he is {age} years old"**
- Use **dictionary unpacking** when testing your function

```
kwargs = {"name": "John", "town": "Sofia", "age": 20}  
print(get_info(**kwargs))
```



```
This is John from Sofia and he is 20 years old
```

# Solution: Person Info

```
def get_info(name, age, town):  
    return f"This is {name} from {town} and he is {age} years old"  
  
# TEST CODE  
print(get_info(**{"name": "George", "town": "Sofia", "age": 20}))
```







# Advanced Sorting

- The **sorted()** method sorts the elements of a given iterable - Ascending or Descending

By default

```
sorted(iterable, key=None, reverse=False)
```

By default

- **iterable** - sequence or collection or any iterator
- **key** - function that serves as a key for the sort comparison
- **reverse** - If =True, the sorted list is reversed (or sorted in Descending order)

# Sorting Dictionary by Key

- Using **lambda** to sort by key element

```
my_dict = {'Peter': 21, 'George': 18, 'John': 45}
sorted_dict = sorted(my_dict.items(), key=lambda x: x[0])
# [('George', 18), ('John', 45), ('Peter', 21)]
```

- Using **reverse** to sort dictionary by key in descending order

```
reversed_dict = sorted(my_dict.items(),
                       key=lambda x: x[0],
                       reverse=True)
# [('Peter', 21), ('John', 45), ('George', 18)]
```

# Sorting Dictionary by Value

- Using **lambda** to sort by value element

```
my_dict = {'Peter': 21, 'George': 18, 'John': 45}
sorted_dict = sorted(my_dict.items(), key=lambda x: x[1])
# [('George', 18), ('Peter', 21), ('John', 45)]
```

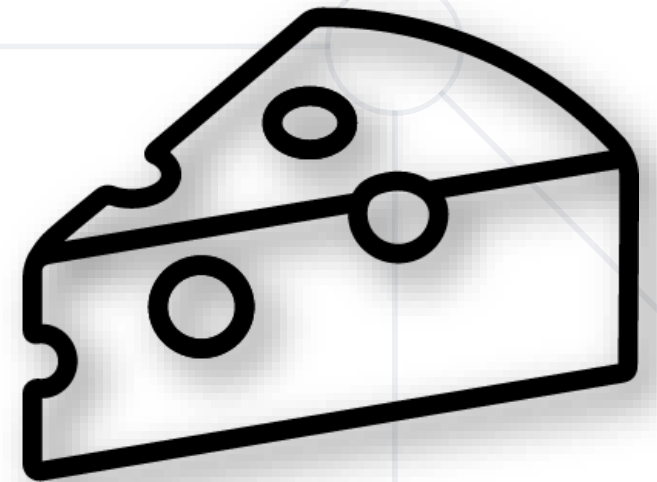
- You could use "-" instead of reverse when sorting descending

```
reversed_dict = sorted(my_dict.items(), key=lambda x: -x[1])
# [('John', 45), ('Peter', 21), ('George', 18)]
```

Works only with numbers

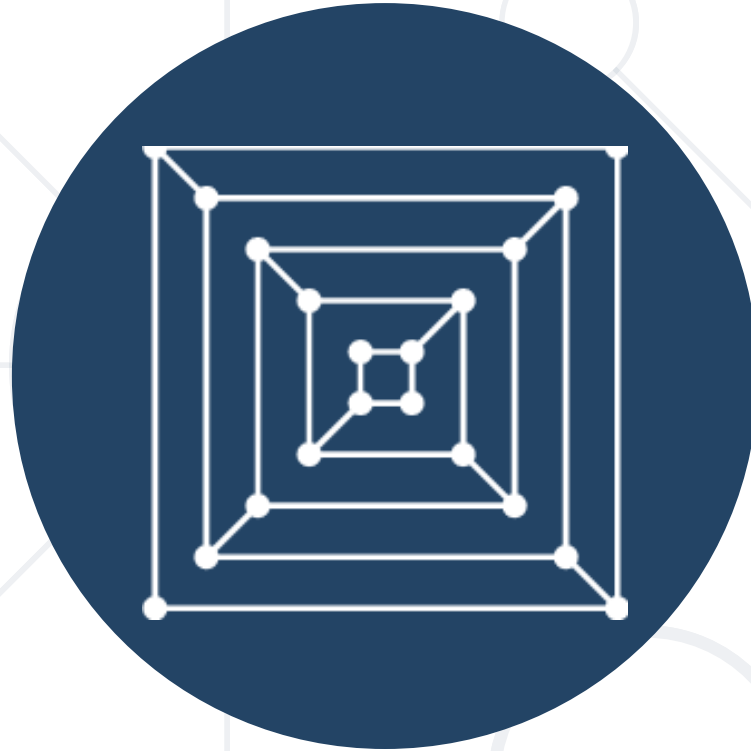
# Problem: Cheese Showcase

- Read the problem description [here](#)
- Create a function as described in the problem description and test it with the given examples
- Submit only your function in the Judge system



# Solution: Cheese Showcase

```
def sorting_cheeses(**cheeses_dict):  
    cheeses_dict = sorted(  
        cheeses_dict.items(),  
        key=lambda x: (-len(x[1]), x[0]))  
  
    result = []  
  
    for (cheese_name, quantities) in cheeses_dict:  
        result.append(cheese_name)  
        quantity_list = sorted(quantities, reverse=True)  
        result += quantity_list  
  
    return "\n".join([str(x) for x in result])
```



# **Nested Functions**

Inner Functions and Closures

# Functions Can be Nested

- Defined **inside** other functions
- The inner function does **not exist outside** the function in which it's defined



```
def outside_function():  
    ...  
    def inside_function():  
        ...  
    ...
```

Inner  
Function

Outer  
Function




# Inner Function Example

```
def factorial(number):  
    if not isinstance(number, int) or number < 0:  
        return f"Sorry. 'number' is incorrect."  
    def inner_factorial(n):  
        fact = 1  
        for i in range(1, n + 1):  
            fact = fact * i  
        return fact  
    return inner_factorial(number)
```

Return the result  
of calling the inner  
function

# Functions Can Return Functions

- The inner function is **no longer "hidden"**
- The outer function returns **behavior**



```
def outside_function():  
    ...  
    def inside_function():  
        ...  
    return inside_function
```

# Function Returning Function Example


```
def calculator(operator):  
    def addition(a, b):  
        return a + b  
    def subtraction(a, b):  
        return a - b  
    if operator == "+":  
        return addition  
    elif operator == "-":  
        return subtraction
```

```
operation = calculator("+")  
result = operation(2, 3)  
print(result)  
# 5
```

Returns a function  
depending on the  
operator

# Lexical Closures

- The inner function can capture and carry some of the **parent function's** state



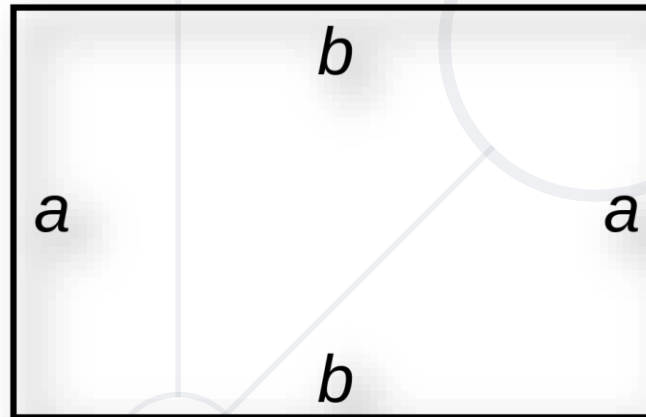
```
def outside_function(number):  
    def inside_function():  
        return number  
    return inside_function  
  
print(outside_function(10)()) # 10
```

# Closures Example

```
def greeting(name):  
    hello = "Hello, "  
    def say_hi():  
        return hello + name  
    return say_hi  
  
print(greeting("Peter"))  
# Hello, Peter
```

# Problem: Rectangle

- Read the problem description [here](#)
- Create a function as described in the problem description and test it with the given examples
- Submit only your function in the Judge system





# Recursion

Function Calling Itself

# What is Recursion?

- The process in which a function calls itself is called **recursion**
- The function that is calling itself is called a **recursive function**
- A recursive function has the following structure
  - A **base** case
  - A **recursive** case





# Base Case and Recursive Case

- The base case in a recursion returns a value **without** making any other **recursive calls**
  - It is the **condition** for the recursion to stop
- The recursive case is the **central part** of the recursive function
  - It is the **solution** to the bigger problem expressed in terms of **smaller problems**



- Factorial recursive representation

```
def fact(n):  
    if n == 1:  
        return 1  
    return n * fact(n - 1)
```

Base Case

Recursive Case

```
factorial(n):
```

```
    if n == 1:  
        return 1  
    else:  
        return n * factorial(n-1):  
        if n == 1:  
            return 1  
        else:
```

*factorial(n) =*

# Problem: Recursive Power

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

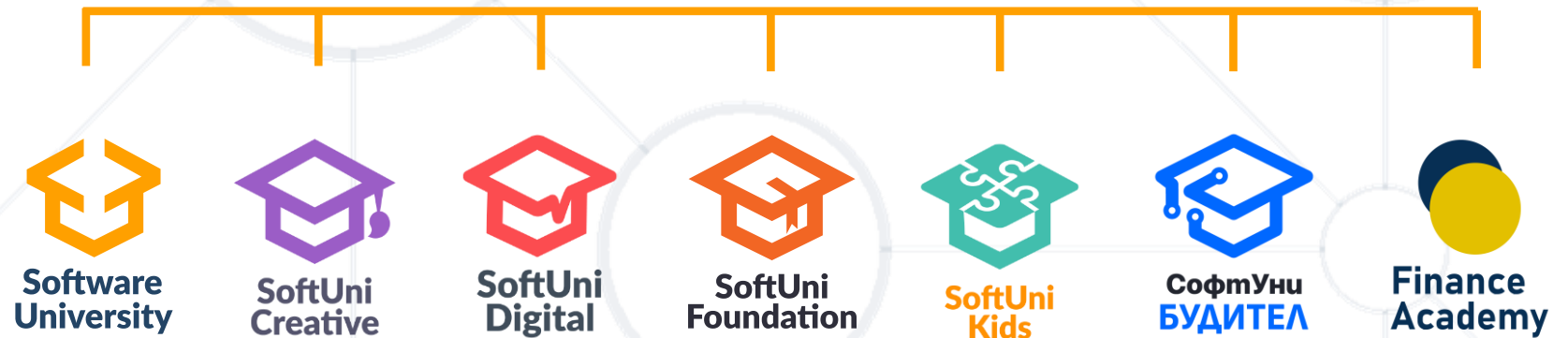
# Solution: Recursive Power

```
def recursive_power(x, y):  
    result = 1  
    if y == 0:  
        return result  
    result = x * recursive_power(x, y - 1)  
    return result
```

- **Packing arguments into:**
  - Tuple
  - Dictionary
- **Unpacking arguments into:**
  - Tuple
  - Dictionary



# Questions?



# SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [softuni.org](http://softuni.org)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)





- This course (slides, examples, demos, exercises, homework, documents, videos, and other assets) is **copyrighted content**
- Unauthorized copy, reproduction, or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>

