LAB-5

1)

a) Write a module named **utils.py** that contains one function called **process_item**. The function will have one parameter, **x**, and will return the least prime number greater than **x**. When run, the module will request an input from the user, convert it to a number and it will display the output of the **process_item** function.

b) Write a module named **app.py**. When this module is run, it will run in an infinite loop, waiting for inputs from the user. The program will convert the input to a number and process it using the function **process_item** implented in **utils.py**. You will have to import this function in your module. The program stops when the user enters the message **"q"**.

2) Create a function and an anonymous function that receive a variable number of arguments. Both will return the sum of the values of the keyword arguments.

Example:

For the call my_function(1, 2, c=3, d=4) the returned value will be 7.

3) Using functions, anonymous functions, list comprehensions and filter, implement three methods to generate a list with all the vowels in a given string.

For the string "Programming in Python is fun" the list returned will be ['o', 'a', 'i', 'i', 'o', 'i', 'u'].

4) Write a function that receives a variable number of arguments and keyword arguments. The function returns a list containing only the arguments which are dictionaries, containing minimum 2 keys and at least one string key with minimum 3 characters.

Example:

```
my_function(
    {1: 2, 3: 4, 5: 6},
    {'a': 5, 'b': 7, 'c': 'e'},
    {2: 3},
    [1, 2, 3],
    {'abc': 4, 'def': 5},
    3764,
    dictionar={'ab': 4, 'ac': 'abcde', 'fg': 'abc'},
    test={1: 1, 'test': True}
) will return: [{'abc': 4, 'def': 5}, {1: 1, 'test': True}]
```

5) Write a function with one parameter which represents a list. The function will return a new list containing all the numbers found in the given list.

Example: my_function([1, "2", {"3": "a"}, {4, 5}, 5, 6, 3.0]) will return [1, 5, 6, 3.0]

6) Write a function that receives a list with integers as parameter that contains an equal number of even and odd numbers that are in no specific order. The function should return a list of pairs (tuples of 2 elements) of numbers (Xi, Yi) such that Xi is the i-th even number in the list and Yi is the i-th odd number

Example:

my_function([1, 3, 5, 2, 8, 7, 4, 10, 9, 2]) will return [(2, 1), (8, 3), (4, 5), (10, 7), (2, 9)]

7) Write a function called process that receives a variable number of keyword arguments

The function generates the first 1000 numbers of the Fibonacci sequence and then processes them in the following way:

If the function receives a parameter called **filters**, this will be a list of predicates (function receiving an argument and returning **True**/**False**) and will retain from the generated numbers only those for which the predicates are **True**.

If the function receives a parameter called **limit**, it will return only that amount of numbers from the sequence.

If the function receives a parameter called **offset**, it will skip that number of entries from the beginning of the result list.

The function will return the processed numbers.

Example:

```
def sum_digits(x):

return sum(map(int, str(x)))

process(

filters=[lambda item: item % 2 == 0, lambda item: item == 2 or 4 <= sum_digits(item) <= 20],

limit=2,

offset=2

) returns [34, 144]
```

Explanation:

```
# Fibonacci sequence will be: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610...

# Valid numbers are: 2, 8, 34, 144, 610, 2584, 10946, 832040

# After offset: 34, 144, 610, 2584, 10946, 832040

# After limit: 34, 144
```

8)

a) Write a function called **print_arguments** with one parameter named **function**. The function will return one new function which prints the arguments and the keyword arguments received and will return the output of the function receives as a parameter.

Example:

```
def multiply_by_two(x):

return x * 2


def add_numbers(a, b):

return a + b


augmented_multiply_by_two = print_arguments(multiply_by_two)

x = augmented_multiply_by_two(10) # this will print: Arguments are: (10,), {} and will return 20.


augmented_add_numbers = print_arguments(add_numbers)
```

```
x = augmented_add_numbers(3, 4) # this will print: Arguments are: (3, 4), {} and will return 7.
```

b) Write a function called **multiply_output** with one parameter named **function**. The function will return one new function which returns the output of the function received multiplied by **2**.

Example:

```
def multiply_by_three(x):

return x * 3



augmented_multiply_by_three = multiply_output(multiply_by_three)

x = augmented_multiply_by_three(10) # this will return 2 * (10 * 3)
```

c) Write a function called **augment_function** with two parameters named **function** and **decorators**. **decorators** will be a list of functions which will have the same signature as the previous functions (**print_arguments**, **multiply_output**). **augment_function** will create a new function which is augmented using all the functions in the decorators list.

Example:

```
def add_numbers(a, b):

return a + b

decorated_function = augment_function(add_numbers, [print_arguments, multiply_output])

x = decorated_function(3, 4) # this will print: Arguments are: (3, 4), {} and will return (2 * (3 + 4))
```

9) Write a function that receives a list of pairs of integers (tuples with 2 elements) as parameter (named **pairs**). The function should return a list of dictionaries for each pair (in the same order as in the input list) that contain the following keys (as strings): *sum* (the value should be sum of the 2 numbers), *prod* (the value should be product of the two numbers), *pow* (the value should be the first number raised to the power of the second number)

Example:

f9(**pairs** = [(5, 2), (19, 1), (30, 6), (2, 2)]) will return [{'sum': 7, 'prod': 10, 'pow': 25}, {'sum': 20, 'prod': 19, 'pow': 19}, {'sum': 36, 'prod': 180, 'pow': 729000000}, {'sum': 4, 'prod': 4, 'pow': 4}]