```python
# log decorator

def log(func):        # func --> add
  def wrapper(*args, **kwargs):      # args -> (1, 2), kwargs -> {}
    print(f"executing {func.__name__} function")
    result = func(*args, **kwargs)
    return result
  return wrapper



@log          # add = log(add) = wrapper
def add(a, b):
  return a + b

# print(add(1, 2))          # add --> wrapper

@log
def sub(a, b):
  return a - b

# print(sub(1, 2))

############################################################
# positive decorator

def positive(func):          # func -> sub
  def wrapper(*args, **kwargs):
    result = func(*args, **kwargs)
    return abs(result)
  return wrapper
```

```python
@positive      # sub = positive(sub)   = wrapper
def sub(a, b):
  return a - b
# print(sub(1, 2))

#########################################################
# decorator to count the number of arguments passed to a function

def no_of_args(func):
  def wrapper(*args, **kwargs):      # args -> (1, 2), kwargs -> {c: 3, d: 4}
    print(f"number of arguments are {len(args) + len(kwargs)}")
    return func(*args, **kwargs)
  return wrapper

@no_of_args
def spam(a, b, c, d):
  return "in spam"
# print(spam(1, 2, c=3, d=4))

#########################################################
#  reverse decorator

def reverse(func):
  def wrapper(*args, **kwargs):
    result = func(*args, **kwargs)
    if isinstance(result, str):
      return result[::-1]
    return result

  return wrapper
```

```python
@reverse
def demo():
  return "hello"

# print(demo())      # olleh

@reverse
def add(a, b):
  return a + b

# print(add(1, 2))

##########################################################
# decorator to execute a function 3 times

def execute_3_times(func):
  def wrapper(*args, **kwargs):
    for _ in range(3):
      print(func(*args, **kwargs))
  return wrapper


@execute_3_times
def greet():
  return "hello world"

# greet()

##########################################################
# decorator to count the number of function calls
```

```
function_count = {}

def count_function_calls(func):
  def wrapper(*args, **kwargs):
    key = func.__name__

    if key not in function_count:
      function_count[key] = 1
    else:
      function_count[key] += 1

    return func(*args, **kwargs)
  return wrapper


@count_function_calls
def add(a, b):
  return a + b

print(add(1, 2))      # {"add": 1}
print(add(5, 8))      # {"add": 2}


@count_function_calls
def sub(a, b):
  return a - b

print(sub(1, 2))      # {"add": 2, sub: 1}
print(sub(5, 8))
print(sub(15, 8))
```

```python
print(sub(15, 8))
print(sub(15, 8))
print(add(10, 20))      # {add: 3, sub: 3}

print(function_count)
# {add: 2, sub: 3}

##########################################################
from collections import defaultdict

# d = dict()  # {}
d = defaultdict(int)

def count_function_calls(func):
  def wrapper(*args, **kwargs):
    key = func.__name__
    d[key] = d[key] + 1
    return func(*args, **kwargs)
  return wrapper

@count_function_calls
def add(a, b):
  return a + b

print(add(1, 2))       # {"add": 1}
print(add(5, 8))       # {"add": 2}


@count_function_calls
def sub(a, b):
  return a - b
```

```
print(sub(1, 2))      # {"add": 2, sub: 1}
print(sub(5, 8))
print(sub(15, 8))
print(sub(15, 8))
print(sub(15, 8))
print(add(10, 20))    # {add: 3, sub: 3}

print(d)
# {add: 2, sub: 3}

##########################################################

"""
1. decorator that accepts only positional arguments

2. decorator to convert the string output of a function to upper case (the output
  must be a string)

3. decorator that accepts only keyword arguments

4. decorator that reverses the output of a function only if the output is a
  sequence(list, str, tuple)

5. decorator that creates a dictionary of arguments passed to a function and their result
pairs
"""

##########################################################
###############
```