```
# decorator that accepts only positional arguments


def pos_only(func):
  def wrapper(*args, **kwargs):        # args -> (1, 2, 3) (1, 2), kwargs -> {}, {c: 3}
    if len(kwargs) == 0:
      print("in wrapper")
      return func(*args, **kwargs)
    else:
      # print("no kwargs are allowed")
      raise TypeError("keyword arguments are not allowed")

  return wrapper


@pos_only
def add(a, b, c):
  return a + b + c


# print(add(1, 2, 3))
# print(add(1, 2, c=3))


def pos_only(func):
  def wrapper(*args):        # args -> (1, 2, 3) (1, 2)
    print("in wrapper")
    return func(*args)
  return wrapper


# print(add(1, 2, 3))
```

```python
# print(add(1, 2, c=3))


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

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


def upper_(func):
  def wrapper(*args, **kwargs):
    result = func(*args, **kwargs)

    if not isinstance(result, str):
      raise ValueError("result must be a string")
    return result.upper()

    # if isinstance(result, str):
    #  return result.upper()
    # else:
    #  raise ValueError("result must be a string")

  return wrapper


@upper_
def full_name(fname, lname):
  return fname + lname



# print(full_name("John", "doe"))
# print(full_name([1, 2], [3, 4]))
```

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

# decorator that creates a dictionary of arguments passed to a function and their result pairs

"""

4 - 1, 4, 2
5 - 1, 5
7 - 1, 7

"""

def is_prime(num):
  factors = 0

  for i in range(1, num+1):      # 1, 2, 3, 4, 5
    if num % i == 0:
      factors += 1

  if factors == 2:
    return True
  else:
    return False

# print(is_prime(123))
```

```
d = {}

def cache(func):

  def wrapper(*args):

    if args not in d:      # (12,) -> False

      result = func(*args)    # False

      d[args] = result       # {(12,) : False}

    return d[args]

  return wrapper




@cache

def isprime(num):      # 14 // 2 = 7

  print("executing isprime...")

  for i in range(2, num//2 + 1):       # 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

    if num % i == 0:

      return False


  return True


# print(isprime(513))

# print(isprime(513))

# print(isprime(513))
```

```
from functools import lru_cache


@lru_cache
def isprime(num):      # 14 // 2 = 7
  print("executing isprime...")
  for i in range(2, num//2 + 1):        # 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
    if num % i == 0:
      return False

  return True


# print(isprime(12))
# print(isprime(12))


##################################################################
# parameterized decorator
import time


def outer(n):
  def delay(func):
    def wrapper(*args, **kwargs):
      print(f"{func.__name__} has delay of {n}")
      time.sleep(n)
      return func(*args, **kwargs)
    return wrapper
  return delay
```

```python
@outer(2)       # @outer(2) ==> @delay => spam = delay(spam)
def spam():
    print("in spam")

# spam()


@outer(3)
def greet():
    print("hello world")

# greet()

#########################################################################
# log decorator that takes user log message

def logging(message="hello world"):
    def log(func):
        def wrapper(*args, **kwargs):
            print(message)
            return func(*args, **kwargs)
        return wrapper
    return log
```

```
@logging("Haii")
def add(a, b):
  return a + b


# print(add(1, 2))


##################################################################
# execute a function n times


##################################################################
# type validator decorator


def type_check(type1, type2):      # type1 = int, type2 = int
  def type_validator(func):      # func = add
    def wrapper(*args):       # args = (1, 2)
      if isinstance(args[0], type1) and isinstance(args[1], type2):
        return func(*args)
      else:
        print("not same")
    return wrapper
  return type_validator


# syntax 2
def type_check(*types):          # types = (int, float, int)
  def type_validator(func):      # func = add
    def wrapper(*args):       # args = (1, 2, 3)
```

```
      for i in range(len(args)):     # i -> 0, 1, 2
        if not isinstance(args[i], types[i]):
          raise TypeError("not same")

      return func(*args)

    return wrapper
  return type_validator

# syntax 3

def type_check(*types):  # types = (int, float, int)
  def type_validator(func):  # func = add
    def wrapper(*args):  # args = (1, 2, 3)
      for arg, type_ in zip(args, types):
        if not isinstance(arg, type_):
          raise TypeError(f"{arg} is not an instance of {type_}")
      return func(*args)
    return wrapper
  return type_validator


@type_check(int, float, int)
def add(a, b, c):
  return a + b + c
```

```
# print(add(1, 2, 3))

# print(add("hi", "hello"))


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

from functools import wraps


def log(func):

  @wraps(func)

  def wrapper():

    print("in wrapper")

    return func()

  return wrapper



@log

def spam():

  """This is spam function and it does nothing"""

  return "in spam"



print(spam())

print(spam.__name__)

print(spam.__doc__)
```