

Functions

#5

Serdar ARITAN

Department of Computer Graphics
Hacettepe University,
Ankara,
Turkey



What will be the output if you run this code?

```
1 numbers = range(3)
2 output = {numbers}
3 print(output)
```

- A. {range}
- B. (range(0, 3))
- C. [0, 1, 2]
- D. (0, 1, 2)
- E. {0, 1, 2}

What will be the output if you run this code?

```
1 numbers = range(3)
2 output = {*numbers}
3 print(output)
```

- A. {range}
- B. (range)
- C. [0, 1, 2]
- D. (0, 1, 2)
- E. {0, 1, 2}

What will be the output if you run this code?

```
1 print("{2}, {1}, {0}".format("abc"))
```

- A. a, b, c
- B. c, b, a
- C. **IndexError**: Replacement index 2 out of range for positional args tuple
- D. {0}, {1}, {2}
- E. {2}, {1}, {0}

What will be the output if you run this code?

```
1 print("{2}, {1}, {0}".format(*"abc"))
```

- A. a, b, c
- B. c, b, a
- C. **IndexError**: Replacement index 2 out of range for positional args tuple
- D. {0}, {1}, {2}
- E. {2}, {1}, {0}



```
1 >>> def my_func(*args):  
2 ...     print(args)  
3 ...  
4 >>> my_func(*"abc")  
5 ('a', 'b', 'c')
```

I have never considered Python to be heavily influenced by functional languages, no matter what people say or think. I was much more familiar with imperative languages such as C and Algol 68 and although I had made functions first-class objects, I didn't view Python as a **functional programming** language.

— Guido van Rossum
Python BDFL

“**Origins of Python's Functional Features**”, from Guido's The History of Python blog.

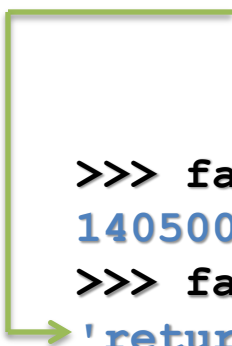
Functions in Python are first-class objects. Programming language theorists define a “*first-class object*” as a program entity that can be:

- Created at runtime
- Assigned to a variable or element in a data structure
- Passed as an argument to a function
- Returned as the result of a function

Treating a Function Like an **Object**

factorial, in mathematics, the product of all positive integers less than or equal to a given positive integer and denoted by that integer and an exclamation point.

```
>>> def factorial(n):  
    '''returns n!'''  
    return 1 if n < 2 else n * factorial(n - 1)  
  
>>> factorial(42)  
1405006117752879898543142606244511569936384000000000  
>>> factorial.__doc__  
'returns n!'  
>>> type(factorial)  
<class 'function'>
```



When to Use a Function

Only one purpose: A function should be the encapsulation of a single, identifiable operation.

Readable: A function should be readable.

Not too long: A function shouldn't be too long.

Reusable: A function should be reusable in contexts other than the program it was written for originally.

Complete: A function should be complete, in that it works in all potential situations. If you write a function to perform one thing, you should make sure that *all the cases* where it might be used are taken into account.

Able to be refactored: Refactoring is the process of taking existing code and modifying it such that its structure is somehow improved but the **functionality of the code remains the same**.

```
>>> def times(x, y):    # Create and assign function
    return x * y        # Body executed when called
```

```
>>> times('Ni', 4)      # Functions are "typeless"
'NiNiNiNi'
```

```
>>> def intersect(seq1, seq2):
    res = []              # Start empty list
    for x in seq1:        # Scan seq1
        if x in seq2:     # Common item?
            res.append(x)  # Add to end
    return res
```

```
>>> s1 = "SPAM"  
>>> s2 = "SCAM"  
>>> intersect(s1, s2) # Strings  
['S', 'A', 'M']
```

the function could be replaced with a single list
comprehension expression

```
>>> [x for x in s1 if x in s2]  
['S', 'A', 'M']
```



Like all good functions in Python, intersect is
polymorphic

```
>>> x = intersect([1, 2, 3], (1, 4)) # Mixed types  
>>> x # Saved result object  
[1]
```


We can even create more functions using the existing ones.

```
>>> def sum_of_squares(x, y):  
    return square(x) + square(y)
```

```
>>> sum_of_squares(2, 3)  
13
```

```
>>> def square(x):  
    return x * x
```

Functions are just like other values, they can be assigned, passed as arguments to other functions etc.

```
>>> f = square
```

```
>>> f(4)
```


```
16
```

```
>>> def fxy(f, x, y):  
    return f(x) + f(y)
```

```
>>> fxy(square, 2, 3)
```

```
13
```

```
>>>def cube(x):  
    return x * x * x  
>>> fxy(cube, 2, 3)  
35
```



There is another way of creating functions, using the *lambda* operator.

```
>>>forthpower = lambda x: x ** 4 # inline function  
>>> fxy(forthpower, 2, 3)  
97  
>>> fxy(lambda x: x ** 5, 2, 3) # fifth power  
275
```

 **lambda**

The *lambda* operator becomes handy when writing small functions to be passed as arguments etc.

Lambda functions are anonymous functions that are not defined in the namespace. Roughly speaking, they are functions without names, intended for single use.

lambda <arguments> : <return expression>

A lambda function can have one or multiple arguments, separated by commas.

```
>>> print((lambda x: x + 3)(3))
```

6

```
>>> print((lambda x, y: x + y)(3, 4))
```

7



```
>>> is_anagram = lambda x1, x2: sorted(x1) == sorted(x2)
```

```
>>> print(is_anagram("elvis", "lives"))
```

```
???
```

```
>>> print(is_anagram("elvis", "livees"))
```

```
???
```

```
>>> print(is_anagram("elvis", "dead"))
```

```
???
```

Functions

Anna madam did
eve level Hannah
deed pop pull up
noon radar Mom
wow Bob

```
>>> is_palindrome = lambda phrase: phrase == phrase[::-1]
```

```
>>> print(is_palindrome("anna"))
```

```
???
```

```
>>> print(is_palindrome("kdljfasjf"))
```

```
???
```

```
>>> print(is_palindrome("rats live on no evil star"))
```

```
???
```

What does this function do? What does it return if $x = 5$?

```
>>>def func(x):  
    total = 0  
    for i in range(x):  
        total += i * (i-1)  
    return total
```

What does this function do? What number is returned by this function?

```
>>>def func():  
    number = 1.0  
    total = 0  
    while number < 100:  
        total = 1//number  
        number+=1  
    return total
```



What will be the output of the following programs?

```
x = 1
```

```
def f():
```

```
    x = 2
```

```
    return x
```

```
print(x) ??
```

```
print(f()) ??
```

```
<CTRL>F6 # restart
```

```
x = 1
```

```
def f():
```

```
    y = x
```

```
    x = 2
```

```
    return x + y
```

```
print(x) ??
```

```
print(f()) ??
```



What will be the output of the following program?

```
x = 2
```

```
def f(a):
```

```
    x = a*a
```

```
    return x
```

```
>>> y = f(3)
```

```
>>> print(x, y) ?
```




What will be the output of the following program?

```
>>> b = 6
>>> def f1(a):
    print(a)
    print(b)
>>> f1(3) ??
```

<CTRL>F6 # restart

```
>>> b = 6
>>> def f2(a):
    print(a)
    print(b)
    b = 9
>>> f2(3) ??
```

UnboundLocalError: cannot access local variable 'b' where it is not associated with a value

Comparing Bytecodes

```
>>> from dis import dis
```

```
>>> dis(f1)
```

| | | | |
|---|-----------------|------------------|--------------------------------|
| 1 | 0 RESUME | 0 | |
| 2 | 2 LOAD_GLOBAL | 1 (NULL + print) | Load global name print |
| | 12 LOAD_FAST | 0 (a) | Load local name a |
| | 14 CALL | 1 | (1 positional, 0 keyword pair) |
| | 22 POP_TOP | | |
| 3 | 24 LOAD_GLOBAL | 1 (NULL + print) | (print) |
| | 34 LOAD_GLOBAL | 2 (b) | Load global name b |
| | 44 CALL | 1 | (1 positional, 0 keyword pair) |
| | 52 POP_TOP | | |
| | 54 RETURN_CONST | 0 (None) | |

```
def f1(a):  
    print(a)  
    print(b)
```

Comparing Bytecodes

```
>>> dis(f2)
```

| | | | |
|---|--------------------|------------------|--------------------------------|
| 1 | 0 RESUME | 0 | |
| 2 | 2 LOAD_GLOBAL | 1 (NULL + print) | Load global name print |
| | 12 LOAD_FAST | 0 (a) | Load local name a |
| | 14 CALL | 1 | (1 positional, 0 keyword pair) |
| | 22 POP_TOP | | |
| 3 | 24 LOAD_GLOBAL | 1 (NULL + print) | |
| | 34 LOAD_FAST_CHECK | 1 (b) | Load local name b |
| | 36 CALL | 1 | |
| | 44 POP_TOP | | |
| 4 | 46 LOAD_CONST | 1 (9) | |
| | 48 STORE_FAST | 1 (b) | |
| | 50 RETURN_CONST | 0 (None) | |

```
def f2(a):  
    print(a)  
    print(b)  
    b = 9
```

This shows that the compiler considers **b** a local variable, even if the assignment to **b** occurs later, because the nature of the variable— whether it is local or not—cannot change the body of the function.



```
#import the disfunction
from dis import dis
#pass code to dis() as a string
```

```
dis("while True : x=10")
```

```
while True:
    x=10
```

| | | | |
|---|----|-----------------|----------|
| 0 | | 0 RESUME | 0 |
| 1 | >> | 2 LOAD_CONST | 1 (10) |
| | | 4 STORE_NAME | 0 (x) |
| | | 6 JUMP_BACKWARD | 3 (to 2) |

What will be the output of the following program?

```
x = 99
def func1():
    global x
    x = 88
def func2():
    global x
    x = 77
>>> x ?
>>> func1()
>>> x ?
>>> func2()
>>> x ?
```

What will be the output of the following program?

```
def scoping():  
    x = 0  
    def increment():  
        x = 5  
        # nonlocal x  
        x += 1  
    increment()  
    increment()  
    return x
```

→ # x = 5
nonlocal x

print(scoping()) ??

global - nonlocal



Write a function which calculates the arithmetic mean of a variable number of values.

```
def arithmetic_mean(x, *liste):  
    """ The function calculates the arithmetic mean of a non-  
    empty arbitrary number of numbers """  
    sum = x  
    for i in liste:  
        sum += i  
    return sum / (1.0 + len(liste))
```

Functions can be called with keyword arguments.

```
>>> def difference(x, y):  
    return x - y
```

```
>>> difference(5, 2)
```

```
3
```

```
>>> difference(x=5, y=2)
```

```
3
```

```
>>> difference(5, y=2)
```

```
3
```

```
>>> difference(y=2, x=5)
```

```
3
```




And some arguments can have **default** values.

```
>>> def increment(x, amount=1):  
    return x + amount
```

```
>>> increment(10)
```

```
11
```

```
>>> increment(10, 5)
```

```
15
```

```
>>> increment(10, amount=2)
```

```
12
```

```
>>> increment(amount=2, x=15)
```

```
17
```

CAUTION
WATCH YOUR
STEP
$$n! = n * (n-1)!, \text{ if } n > 1 \text{ and } f(1) = 1$$

Example:

$$4! = 4 * 3!$$
$$3! = 3 * 2!$$
$$2! = 2 * 1$$

```
>>> def factorial(n):  
    f = 1  
    while (n > 0):  
        f = f * n  
        n = n - 1  
    return f
```

Recursion has
something to
do with infinity



```
>>> def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n-1)
```

```
>>> f= lambda x: x and x * f(x - 1) or 1
```

Functions

Recursion

```
def main():  
    loopnum = int(input("How many times would you like to loop?"))  
    counter = 1  
    recurr(loopnum, counter)  
  
def recurr(loopnum, counter):  
    if loopnum > 0:  
        print("This is loop iteration", counter)  
        recurr(loopnum - 1, counter + 1)  
    else:  
        print("The loop is complete.")  
  
main()
```

Enter :10, 100 then 1200

Functions

Recursion

The Fibonacci sequence is a sequence of numbers such that any number, except for the first and second, is the sum of the previous two: 0, 1, 1, 2, 3, 5, 8, 13, 21....

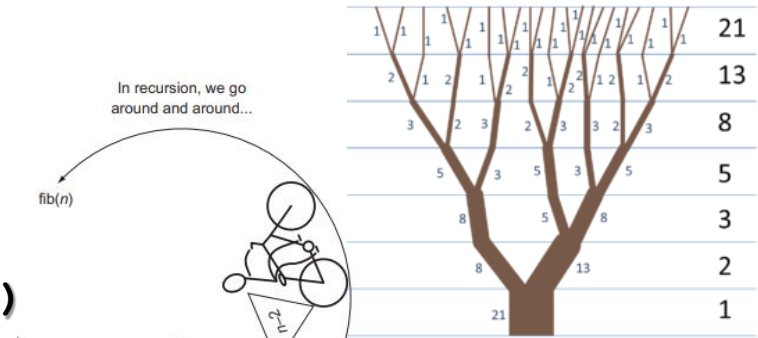
```
fib(n) = fib(n - 1) + fib(n - 2)
```

```
>>> def fib1(n):  
    return fib1(n-1) + fib1(n-2)  
>>> fib1(5)
```

```
return fib1(n-1) + fib1(n-2)
```

[Previous line repeated 1022 more times]

RecursionError: maximum recursion depth exceeded



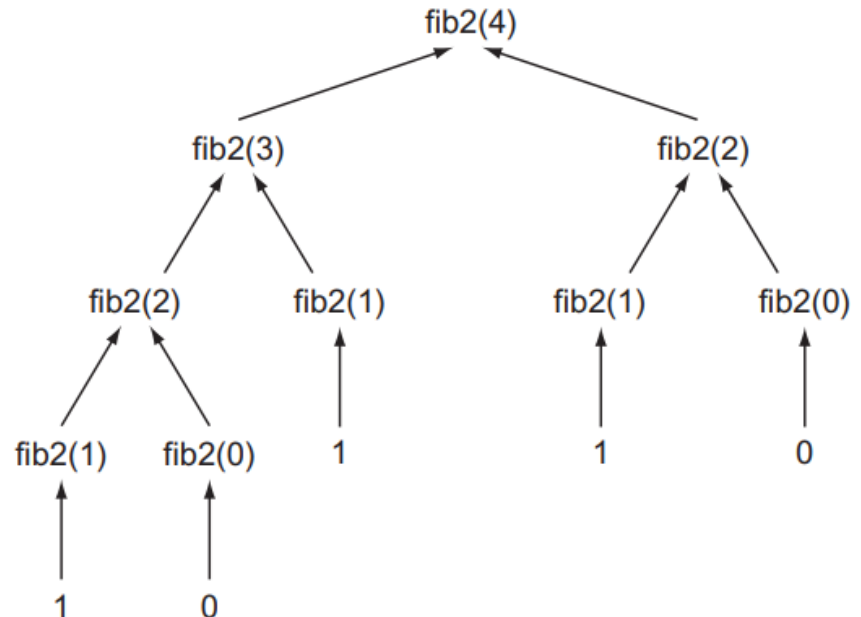
The recursive function `fib(n)` calls itself with the arguments `n-2` and `n-1`.

Functions

Recursion

```
def fib2(n):  
    if n < 2:  
        return n  
    return fib2(n-1) + fib2(n-2)  
print(fib2(4))
```

fib2(4) -> fib2(3), fib2(2)
fib2(3) -> fib2(2), fib2(1)
fib2(2) -> fib2(1), fib2(0)
fib2(2) -> fib2(1), fib2(0)
fib2(1) -> 1
fib2(1) -> 1
fib2(1) -> 1
fib2(0) -> 0
fib2(0) -> 0



Functions

Recursion

Dependencies

```
from functools import reduce
```

```
n = 10
```

```
fibs = reduce(lambda x, _: x + [x[-2] + x[-1]], [0] * (n - 2), [0, 1])
```

```
print(fibs)
```

The `reduce(fun, seq)` function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along. This function is defined in “`functools`” module.



Functions Recursion

```
# python code to demonstrate working of reduce()
# importing functools for reduce()
import functools

# initializing list
lis = [1, 3, 5, 6, 2]

# using reduce to compute sum of list
print("The sum of the list elements is : ", end="")
print(functools.reduce(lambda a, b: a+b, lis))

# using reduce to compute maximum element from list
print("The maximum element of the list is : ", end="")
print(functools.reduce(lambda a, b: a if a > b else b, lis))
```




Built-in (Python)

Names preassigned in the built-in names module: `open`, `range`, `SyntaxError`....

Global (module)

Names assigned at the top-level of a module file, or declared `global` in a `def` within the file.

Enclosing function locals

Names in the local scope of any and all enclosing functions (`def` or `lambda`), from inner to outer.

Local (function)

Names assigned in any way within a function (`def` or `lambda`), and not declared `global` in that function.

Higher-Order Functions : A function that **takes a function as argument** or returns a function as the result is a higher-order function.

```
>>> fruits=['strawberry','fig','apple','cherry','raspberry','banana']
>>> sorted(fruits, key=len)
['fig', 'apple', 'cherry', 'banana', 'raspberry', 'strawberry']
>>> def reverse(word):
    return word[::-1]
>>> reverse('serdar')
'radres'
>>> sorted(fruits, key=reverse)
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>> sorted(fruits, key=lambda word: word[::-1])
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
```

Function objects have many **attributes** beyond `__doc__`. See what the `dir` function reveals about our `factorial`:

```
>>> dir(factorial)
['__annotations__', '__call__', '__class__', '__closure__',
 '__code__', '__defaults__', '__delattr__', '__dict__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__get__',
 '__getattr__', '__globals__', '__gt__', '__hash__',
 '__init__', '__kwdefaults__', '__le__', '__lt__', '__module__',
 '__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__']
```



Adding arbitrary **metadata** annotations to Python functions. Allowing us to attach **metadata** to functions describing their **parameters** and **return values**.


```
funcdef: 'def' NAME parameters ['->' test] ':' suite
```

```
>>> def kinetic_energy(m: 'in KG', v: 'in M/S')->'Joules':  
...     return 1/2*m*v**2  
...  
>>> kinetic_energy.__annotations__  
{'return': 'Joules', 'v': 'in M/S', 'm': 'in KG'}
```



```
def kinetic_energy(m: 'in KG', v: 'in M/S') -> 'Joules':  
    return 1/2*m*v**2
```

```
print(kinetic_energy.__annotations__)  
{'return': 'Joules', 'v': 'in M/S', 'm': 'in KG'}
```



This is a dictionary
key : value

```
print(kinetic_energy.__annotations__['return'])  
Joules
```

```
print("Kinetic Energy is {} {}".format(kinetic_energy(5, 20),  
                                       kinetic_energy.__annotations__['return']))
```

```
Kinetic Energy is 1000.0 Joules
```

```
def int_return(x) -> int:  
    return int(x)
```

the `-> int` just tells that `int_return()` returns an integer (but it **doesn't force** the function to return an integer).

```
print(int_return(5)) # Ok  
print(int_return('S'))  
ValueError: invalid literal for int() with base 10: 'S'
```

```
def int_return(x: (int, float) ) -> int:  
    if isinstance(x, (int_return.__annotations__['x'])):  
        return int(x)
```

The `isinstance()` function returns **True** if the specified object is of the specified type, otherwise **False**.

`isinstance(object, type)`



```
def int_return(x: (int, float) )-> int:  
    if isinstance(x, (int_return.__annotations__['x'])):  
        return int(x)
```

```
print(int_return.__annotations__['x'])  
(<class 'int'>, <class 'float'>)
```

```
print(int_return(5))  
5
```

```
print(int_return(5.5))  
5
```

```
print(int_return('S'))  
None
```

It is also possible to have a python data structure rather than just a string:

```
Rd = {'type':float,'units':'Joules','docstring':'Given mass and velocity returns kinetic energy in Joules'}
```

```
def f()->rd:  
    pass
```

```
print(f.__annotations__['return']['type'])  
<class 'float'>  
print(f.__annotations__['return']['units'])  
Joules  
print(f.__annotations__['return']['docstring'])  
Given mass and velocity returns kinetic energy in Joules
```




Decorators are a way to modify the behavior of an object by wrapping it within a function.

```
def my_function():  
    print("Function called")
```

```
print("Script start")  
my_function()  
print("Script end")
```

----- RUN -----

```
Script start  
Function called  
Script end
```



A **decorator** is a callable that takes **another function** as argument (the decorated function).

```
def my_decorator(fn):  
    def wrapper():  
        print("entering the function...")  
        fn()  
        print("exiting the function...")  
    return wrapper
```

```
@my_decorator  
def my_function():  
    print("inside the function...")
```

`my_function()`



run

entering the function..
inside the function..
exiting the function..



Decorators can also be stacked

```
def decorator_1(fn):  
    def wrapper():  
        print("entering the decorator 1...")  
        fn()  
        print("exiting the decorator 1...")  
    return wrapper  
  
def decorator_2(fn):  
    def wrapper():  
        print("entering the decorator 2...")  
        fn()  
        print("exiting the decorator 2...")  
    return wrapper  
  
def decorator_3(fn):  
    def wrapper():  
        print("entering the decorator 3...")  
        fn()  
        print("exiting the decorator 3...")  
    return wrapper
```



```
@decorator_1
@decorator_2
@decorator_3
def my_function():
    print("inside the function.")
```

```
>>> my_function()
entering decorator 1...
entering decorator 2...
entering decorator 3...
inside the function
exiting decorator 3...
exiting decorator 2...
exiting decorator 1...
```



It's important to note that **decorators** are a **syntactic sugar**, meaning they make a particular design pattern more elegant, but they **do not add any additional functionality** to the language.

```
def dec(fn):  
    def wrapper():  
        fn()  
    return wrapper
```

```
# syntactic sugar, because this...
```

```
@dec
```

```
def fn():  
    pass
```

```
# is functionally equivalent to this.
```

```
fn = dec(fn)
```



A decorator usually replaces a function with a different one. A key feature of decorators is that they run right after the decorated function is defined.

```
registry = []
def register(func):
    print(f'running register{func}')
    registry.append(func)
    return func
@register
def f1():
    print('running f1()')
@register
def f2():
    print('running f2()')
def f3():
    print('running f3()')
def main():
    print('running main()')
    print('registry ->', registry)
    f1()
    f2()
    f3()

if __name__ == '__main__':
    main()
```

registry will hold references to functions decorated by @register
register takes a function as argument
Display what function is being decorated, for demonstration
Include func in registry
we must return a function; here we return the same received as argument
f1 are decorated by @register

f2 are decorated by @register

f3 is not decorated

main displays the registry, then calls f1(), f2(), and f3()

main() is only invoked if registration.py runs as a script



Note that register runs (twice) before any other function in the module. When register is called, it receives as an argument the function object being decorated—for example, `<function f1 at 0x100631bf8>`. After the module is loaded, the registry holds references to the two decorated functions: `f1` and `f2`. These functions, as well as `f3`, are only executed when explicitly called by main.

```
$ python3 registration.py
running register(<function f1 at 0x100631bf8>)
running register(<function f2 at 0x100631c80>)
running main()
registry -> [<function f1 at 0x100631bf8>, <function f2 at 0x100631c80>]
running f1()
running f2()
running f3()
```



```
# adding decorator to the functions
```

```
@myLog_decorator
```

```
def twoSum(x, y):  
    return x + y
```

```
@myLog_decorator
```

```
def twoMultiply(x, y):  
    return x * y
```

```
a, b = 5, 3
```

```
# getting the value through return of the function
```

```
print("Sum =", twoSum(a, b))
```

```
twoMultiply(a, b)
```

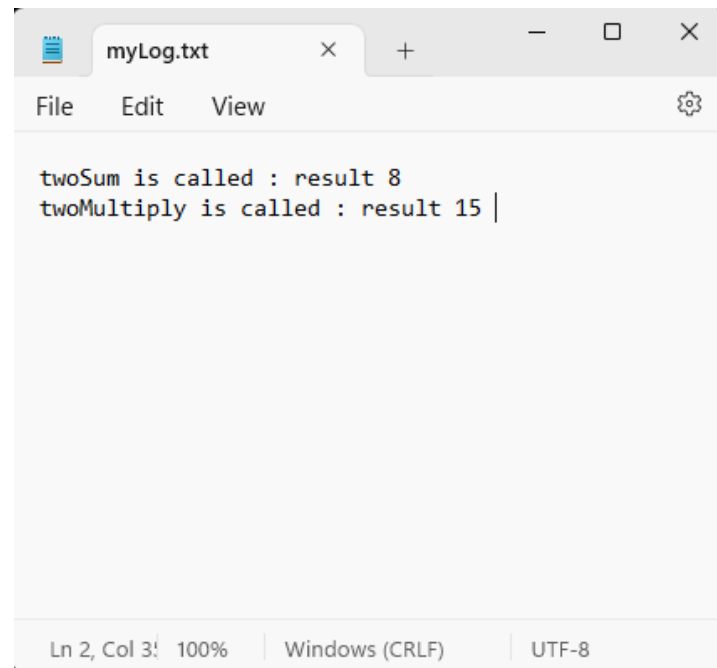



```
def myLog_decorator(func):  
    def inner(*args, **kwargs):  
  
        print("1 : before Execution")  
  
        # getting the returned value  
        returned_value = func(*args, **kwargs)  
        with open('myLog.txt', 'a') as f:  
            f.write(func.__name__ +  
                    ' is called : result ' +  
                    str(returned_value) + ' \n')  
  
        print("2 : after Execution")  
  
        # returning the value to the original frame  
        return returned_value  
  
    return inner
```



```
1 : before Execution
2 : after Execution
Sum = 8
1 : before Execution
2 : after Execution
```

Functions



```
twoSum is called : result 8
twoMultiply is called : result 15 |
```

Ln 2, Col 3 | 100% | Windows (CRLF) | UTF-8



How to unpack more than one variable when functions return multiple values.

```
def get_stats(numbers):  
    minimum = min(numbers)  
    maximum = max(numbers)  
    return minimum, maximum  
  
lengths = [63, 73, 72, 60, 67, 66, 71, 61, 72, 70]  
minimum, maximum = get_stats(lengths) # Two return values  
print(f'Min: {minimum}, Max: {maximum}')
```



Keyword Arguments

```
def describe_pet(animal_type, pet_name):  
    """Display information about a pet."""  
    print(f"\nI have a {animal_type}.")  
    print(f"My {animal_type}'s name is {pet_name}.")
```

```
describe_pet('dog', 'Bruno')  
describe_pet('Bruno', 'dog')
```

```
I have a dog.  
My dog's name is Bruno.
```

```
I have a bruno.  
My bruno's name is Dog.
```



When you use keyword arguments, be sure to use the exact names of the parameters in the function's definition.

```
describe_pet(animal_type='dog', pet_name='Bruno')  
describe_pet(pet_name='Bruno', animal_type='dog')
```

```
I have a dog.  
My dog's name is Bruno.
```

```
I have a dog.  
My dog's name is Bruno.
```

Default Values

```
def describe_pet(pet_name, animal_type='dog'):  
    """Display information about a pet."""  
    print(f"\nI have a {animal_type}.")  
    print(f"My {animal_type}'s name is {pet_name}.")
```

```
describe_pet(pet_name='Bruno')
```

```
I have a dog.
```

```
My dog's name is Bruno.
```

Making an Argument Optional

```
def get_formatted_name(first_name, middle_name, last_name):  
    """Return a full name, neatly formatted."""  
    full_name = f"{first_name} {middle_name} {last_name}"  
    return full_name.title()
```

```
musician = get_formatted_name('john', 'lee', 'hooker')  
print(musician)
```

John Lee Hooker

Making an Argument Optional

```
def get_formatted_name(first_name, last_name, middle_name = ''):
    """Return a full name, neatly formatted."""
    if middle_name:
        full_name = f"{first_name} {middle_name} {last_name}"
    else:
        full_name = f"{first_name} {last_name}"
    return full_name.title()
```

```
musician = get_formatted_name('jimi', 'hendrix')
print(musician)
musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```




Passing a List

```
def greet_users(names):  
    """Print a simple greeting to each user in the list."""  
    for name in names:  
        msg = f"Hello, {name.title()}!"  
        print(msg)
```


```
username = ['bruno', 'kuki', 'malika']  
greet_users(username)
```

```
Hello, Bruno!  
Hello, Kuki!  
Hello, Malika!
```




Modifying a List in a Function

```
def greet_users(names):  
    """Print a simple greeting to each user in the list."""  
    names.append('benek')  
    for name in names:  
        msg = f"Hello, {name.title()}!"  
        print(msg)
```



```
username = ['bruno', 'kuki', 'malika']  
greet_users(username)  
print(username)  
Hello, Bruno!  
Hello, Kuki!  
Hello, Malika!  
Hello, Benek!  
['bruno', 'kuki', 'malika', 'benek']
```





Modifying a List in a Function

```
def greet_users(names):  
    """Print a simple greeting to each user in the list."""  
    names = names + ['benek']  
    for name in names:  
        msg = f"Hello, {name.title()}!"  
        print(msg)
```

```
usernames = ['bruno', 'kuki', 'malika']  
greet_users(usernames)  
print(usernames)  
Hello, Bruno!  
Hello, Kuki!  
Hello, Malika!  
Hello, Benek!  
['bruno', 'kuki', 'malika']
```



Modifying a List in a Function


```
def greet_users(names):  
    """Print a simple greeting to each user in the list."""  
    names += ['benek']  
    for name in names:  
        msg = f"Hello, {name.title()}!"  
        print(msg)
```

```
usernames = ['bruno', 'kuki', 'malika']  
greet_users(usernames)  
print(usernames)  
Hello, Bruno!  
Hello, Kuki!  
Hello, Malika!  
Hello, Benek!  
['bruno', 'kuki', 'malika', 'benek']
```



Modifying a List in a Function

```
def greet_users(names):  
    """Print a simple greeting to each user in the list."""  
    names.append('benek')  
    for name in names:  
        msg = f"Hello, {name.title()}!"  
        print(msg)
```



```
usernames = ['bruno', 'kuki', 'malika']
```

```
greet_users(usernames[:])
```

```
print(usernames)
```

```
Hello, Bruno!
```

```
Hello, Kuki!
```

```
Hello, Malika!
```

```
Hello, Benek!
```

```
['bruno', 'kuki', 'malika']
```

Shallow copy of list



Same List





Passing an Arbitrary Number of Arguments

```
def make_pizza(*toppings):  
    """Print the list of toppings that have been requested."""  
    print(toppings)
```

```
make_pizza('pepperoni')
```

```
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

```
('pepperoni',)
```

```
('mushrooms', 'green peppers', 'extra cheese')
```



Passing an Arbitrary Number of Arguments

```
def make_pizza(*toppings):  
    """Summarize the pizza we are about to make."""  
    print("\nMaking a pizza with the following toppings:")  
    for topping in toppings:  
        print(f"- {topping}")
```

```
make_pizza('pepperoni')
```

```
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

```
Making a pizza with the following toppings:
```

```
- pepperoni
```

```
Making a pizza with the following toppings:
```

```
- mushrooms
```

```
- green peppers
```

```
- extra cheese
```



Passing an Arbitrary Number of Arguments

```
def make_pizza(size, *toppings):  
    """Summarize the pizza we are about to make."""  
    print(f"\nMaking a {size}-inch pizza with the following toppings:")  
    for topping in toppings:  
        print(f"- {topping}")
```

```
make_pizza(16, 'pepperoni')
```

```
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
```

```
- pepperoni
```

```
Making a 12-inch pizza with the following toppings:
```

```
- mushrooms
```

```
- green peppers
```

```
- extra cheese
```


Importing an Entire Module

```
def make_pizza(size, *toppings):  
    """Summarize the pizza we are about to make."""  
    print(f"\nMaking a {size}-inch pizza with the following toppings:")  
    for topping in toppings:  
        print(f"- {topping}")
```

} save as "pizza.py"

```
import pizza  
pizza.make_pizza(16, 'pepperoni')  
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Making a 16-inch pizza with the following toppings:

- pepperoni

Making a 12-inch pizza with the following toppings:

- mushrooms
- green peppers
- extra cheese



Importing Specific Functions

```
from pizza import make_pizza
```

```
make_pizza(16, 'pepperoni')
```

```
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
from pizza import make_pizza as mp
```

```
mp(16, 'pepperoni')
```

```
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```



```
def drawBox():  
    print("*****")  
    print("      *")  
    print("      *")  
    print("*****")
```

The `drawBox` function works correctly. It draws the particular box that it was intended to draw, but it is not flexible, and as a result, it is not as useful as it could be.



```
def drawBox(width, height):  
    ''' draw a box by using '*' with given width / height  
    box that is smaller than 2x2 cannot be drawn by  
    this function '''  
  
    if width < 2 or height < 2:  
        print("Error: The width or height is too small.")  
        quit()  
    # Draw the top of the box  
    print("*" * width)  
    # Draw the sides of the box  
    for i in range(height - 2):  
        print("*" + " " * (width - 2) + "*")  
    # Draw the bottom of the box  
    print("*" * width)
```

```
drawBox(15, 4)
```

- Write a function `drawBox` that takes four arguments and draw a box.

```
def drawBox(width, height, outline="*", fill=" "):
```

```
drawBox(14, 5, "@", ".")
```

```
@@@@@@@@@@@@@@
@.....@
@.....@
@.....@
@@@@@@@@@@@@@
```

- Write a **function** `from_roman_number(roman_number)` that computes the corresponding decimal number from a textually valid Roman number.
- Write **function** `to_roman_number(value)` that converts a decimal number to a (valid) Roman number.

| Arabic | Roman |
|--------|-----------|
| 17 | "XVII" |
| 444 | "CDXLIV" |
| 1971 | "MCMLXXI" |
| 2020 | "MMXX" |

For syntactically invalid Roman numbers, such as **IXD**, an incorrect result, here 489, can be computed by applying subtraction rule twice in a row: $0 - 1 - 10 + 500$.

Write a **functions** that takes a word as an input argument and returns the number of letters in the word, the number of vowels in the word, and the percentage of vowels.

```
>>> letterStats('sequoia')  
>>> (7, 5, 71.42)
```