Variable-length arguments

In Python, we can pass a variable number of arguments to a function using special symbols. There are two special symbols:

- *args (Non-Keyword Arguments)
- **kwargs (Keyword Arguments)

Example 1: Variable length non-keywords argument

```
# Python program to illustrate
# *args for variable number of arguments


def myFun(*argv):
    for arg in argv:
        print(arg)


myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Output

```
Hello
Welcome
to
GeeksforGeeks
```

Example 2: Variable length keyword arguments

```
# Python program to illustrate
# *kwargs for variable number of keyword arguments


def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))


# Driver code
myFun(first='Geeks', mid='for', last='Geeks')
```

Output

```
first == Geeks
mid == for
last == Geeks
```

Docstring

The first string after the function is called the Document string or Docstring in short. This is used to describe the functionality of the function. The use of docstring in functions is optional but it is considered a good practice.

The below syntax can be used to print out the docstring of a function:

Syntax: `print(function_name.__doc__)`

Example: Adding Docstring to the function

```
# A simple Python function to check
# whether x is even or odd



def evenOdd(x):
    """Function to check if the number is even or odd"""


    if (x % 2 == 0):
        print("even")
    else:
        print("odd")



# Driver code to call the function
print(evenOdd.__doc__)
```

Output

```
Function to check if the number is even or odd
```

The return statement

The function return statement is used to exit from a function and go back to the function caller and return the specified value or data item to the caller.

Syntax: `return [expression_list]`

The return statement can consist of a variable, an expression, or a constant which is returned to the end of the function execution. If none of the above is present with the return statement a None object is returned.

Is Python Function Pass by Reference or pass by value?

One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function, a new reference to the object is created. Parameter passing in Python is the same as reference passing in Java.

Example:

```
# Here x is a new reference to same list lst
```

```
def myFun(x):
    x[0] = 20



# Driver Code (Note that lst is modified
# after function call.
lst = [10, 11, 12, 13, 14, 15]
myFun(lst)
print(lst)
```

Output

```
[20, 11, 12, 13, 14, 15]
```

When we pass a reference and change the received reference to something else, the connection between the passed and received parameter is broken. For example, consider the below program.

```
def myFun(x):

    # After below line link of x with previous
    # object gets broken. A new object is assigned
    # to x.
    x = [20, 30, 40]



# Driver Code (Note that lst is not modified
# after function call.
lst = [10, 11, 12, 13, 14, 15]
myFun(lst)
print(lst)
```

Output

```
[10, 11, 12, 13, 14, 15]
```

Another example to demonstrate that the reference link is broken if we assign a new value (inside the function).

```
def myFun(x):
```

```
    # After below line link of x with previous
    # object gets broken. A new object is assigned
    # to x.
    x = 20


# Driver Code (Note that lst is not modified
# after function call.
x = 10
myFun(x)
print(x)
```

Output

```
10
```

Exercise: Try to guess the output of the following code.

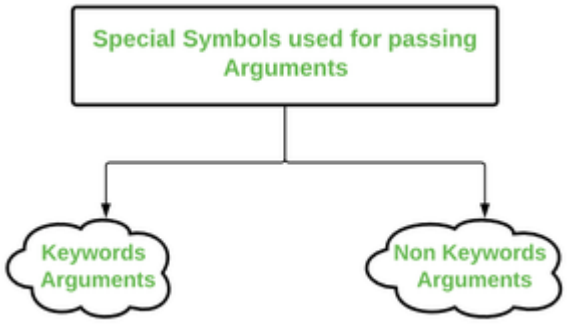
```
def swap(x, y):
    temp = x
    x = y
    y = temp


# Driver code
x = 2
y = 3
swap(x, y)
print(x)
print(y)
```

Output

```
2
3
```

*args and **kwargs in Python

In Python, we can pass a variable number of arguments to a function using special symbols. There are two special symbols:



Special Symbols Used for passing arguments:-

1.)*args (Non-Keyword Arguments)

2.)**kwargs (Keyword Arguments)

Note: "We use the "wildcard" or "*" notation like this – *args OR **kwargs – as our function's argument when we have doubts about the number of arguments we should pass in a function."

1.) *args

The special syntax **args* in function definitions in python is used to pass a variable number of arguments to a function. It is used to pass a non-key worded, variable-length argument list.

- The syntax is to use the symbol * to take in a variable number of arguments; by convention, it is often used with the word args.
- What **args* allows you to do is take in more arguments than the number of formal arguments that you previously defined. With **args*, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).
- For example : we want to make a multiply function that takes any number of arguments and able to multiply them all together. It can be done using *args.
- Using the *, the variable that we associate with the * becomes an iterable meaning you can do things like iterate over it, run some higher-order functions such as map and filter, etc.

```
# Python program to illustrate

# *args for variable number of arguments

def myFun(*argv):

    for arg in argv:
```

```
        print (arg)


myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Output:

```
Hello
Welcome
to
GeeksforGeeks
```

```
# Python program to illustrate

# *args with first extra argument

def myFun(arg1, *argv):

    print ("First argument :", arg1)

    for arg in argv:

        print("Next argument through *argv :", arg)


myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Output:

```
First argument : Hello
Next argument through *argv : Welcome
Next argument through *argv : to
Next argument through *argv : GeeksforGeeks
```

2.)**kwargs

The special syntax ***kwargs* in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name *kwargs* with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).

- A keyword argument is where you provide a name to the variable as you pass it into the function.
- One can think of the *kwargs* as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the *kwargs* there doesn't seem to be any order in which they were printed out.

Example for usage of **kwargs:

```
# Python program to illustrate
# *kwargs for variable number of keyword arguments

def myFun(**kwargs):
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))

# Driver code
myFun(first ='Geeks', mid ='for', last='Geeks')
```

Output:

```
last == Geeks
mid == for
first == Geeks
```

```
# Python program to illustrate  **kwargs for
# variable number of keyword arguments with
# one extra argument.

def myFun(arg1, **kwargs):
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))

# Driver code
myFun("Hi", first ='Geeks', mid ='for', last='Geeks')
```

Output:

```
last == Geeks
mid == for
first == Geeks
```

Using *args and **kwargs to call a function

Example:

```
def myFun(arg1, arg2, arg3):
    print("arg1:", arg1)
```

```
    print("arg2:", arg2)
    print("arg3:", arg3)


# Now we can use *args or **kwargs to
# pass arguments to this function :
args = ("Geeks", "for", "Geeks")
myFun(*args)


kwargs = {"arg1" : "Geeks", "arg2" : "for", "arg3" : "Geeks"}
myFun(**kwargs)
```

Output:

```
arg1: Geeks
arg2: for
arg3: Geeks
arg1: Geeks
arg2: for
arg3: Geeks
```

Using *args and **kwargs in same line to call a function

Example:

```
def myFun(*args,**kwargs):
    print("args: ", args)
    print("kwargs: ", kwargs)



# Now we can use both *args ,**kwargs
# to pass arguments to this function :
myFun('geeks','for','geeks',first="Geeks",mid="for",last="Geeks")
```

Output:

```
args: ('geeks', 'for', 'geeks')
kwargs {'first': 'Geeks', 'mid': 'for', 'last': 'Geeks'}
```

When to use yield instead of return in Python?

- Difficulty Level : Easy

- Last Updated : 21 Jul, 2021

The yield statement suspends function's execution and sends a value back to the caller, but retains enough state to enable function to resume where it is left off. When resumed, the function continues execution immediately after the last yield run. This allows its code to produce a series of values over time, rather than computing them at once and sending them back like a list.

Let's see with an example:

```
# A Simple Python program to demonstrate working
# of yield


# A generator function that yields 1 for the first time,
# 2 second time and 3 third time
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3


# Driver code to check above generator function
for value in simpleGeneratorFun():
    print(value)
```

Output:

```
1
2
3
```

Return sends a specified value back to its caller whereas **Yield** can produce a sequence of values. We should use yield when we want to iterate over a sequence, but don't want to store the entire sequence in memory.

Yield are used in Python **generators**. A generator function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function.

```
# A Python program to generate squares from 1
# to 100 using yield and therefore generator
```

```
# An infinite generator function that prints
# next square number. It starts with 1
def nextSquare():
    i = 1

    # An Infinite loop to generate squares
    while True:
        yield i*i
        i += 1 # Next execution resumes
                # from this point

# Driver code to test above generator
# function
for num in nextSquare():
    if num > 100:
         break
    print(num)
```

Output:

```
1
4
9
16
25
36
49
64
81
100
```

Generators in Python

- Difficulty Level : Easy
- Last Updated : 31 Mar, 2020

Prerequisites: Yield Keyword and Iterators

There are two terms involved when we discuss generators.

1. **Generator-Function :** A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function.

```
# A generator function that yields 1 for first time,
# 2 second time and 3 third time
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3


# Driver code to check above generator function
for value in simpleGeneratorFun():
    print(value)
```

2. Output :

3. 1
4. 2
5. 3

6. **Generator-Object :** Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a “for in” loop (as shown in the above program).

```
# A Python program to demonstrate use of
# generator object with next()


# A generator function
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3


# x is a generator object
x = simpleGeneratorFun()


# Iterating over the generator object using next
print(x.next()) # In Python 3, __next__()
print(x.next())
print(x.next())
```

7. Output :

```
1
2
3
```

So a generator function returns an generator object that is iterable, i.e., can be used as an Iterators .

As another example, below is a generator for Fibonacci Numbers.

```
# A simple generator for Fibonacci Numbers
def fib(limit):

    # Initialize first two Fibonacci Numbers
    a, b = 0, 1

    # One by one yield next Fibonacci Number
    while a < limit:
        yield a
        a, b = b, a + b

# Create a generator object
x = fib(5)

# Iterating over the generator object using next
print(x.next()) # In Python 3, __next__()
print(x.next())
print(x.next())
print(x.next())
print(x.next())

# Iterating over the generator object using for
# in loop.
print("\nUsing for in loop")
for i in fib(5):
    print(i)
```

Output :

```
0
1
1
2
3

Using for in loop
0
1
1
2
3
```

Applications : Suppose we to create a stream of Fibonacci numbers, adopting the generator approach makes it trivial; we just have to call next(x) to get the next Fibonacci number without bothering about where or when the stream of numbers ends.
A more practical type of stream processing is handling large data files such as log files. Generators provide a space efficient method for such data processing as only parts of the file are handled at one given point in time. We can also use Iterators for these purposes, but Generator provides a quick way (We don't need to write __next__ and __iter__ methods here).

Properties of first class functions:

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, …

Examples illustrating First Class functions in Python

1. Functions are objects: Python functions are first class objects. In the example below, we are assigning function to a variable. This assignment doesn't call the function. It takes the function object referenced by shout and creates a second name pointing to it, yell.

```
# Python program to illustrate functions
# can be treated as objects
def shout(text):
    return text.upper()

print (shout('Hello'))

yell = shout

print (yell('Hello'))
```

Output:

```
HELLO
HELLO
```

2. Functions can be passed as arguments to other functions: Because functions are objects we can pass them as arguments to other functions. Functions that can accept other functions as arguments are also called higher-order functions. In the example below, we have created a function **greet** which takes a function as an argument.

```
# Python program to illustrate functions
# can be passed as arguments to other functions
def shout(text):
    return text.upper()

def whisper(text):
    return text.lower()

def greet(func):
    # storing the function in a variable
    greeting = func("""Hi, I am created by a function
                    passed as an argument.""")
    print (greeting)

greet(shout)
greet(whisper)
```

Output

```
HI, I AM CREATED BY A FUNCTION PASSED AS AN ARGUMENT.
hi, i am created by a function passed as an argument.
```

3. Functions can return another function: Because functions are objects we can return a function from another function. In the below example, the create_adder function returns adder function.

```
# Python program to illustrate functions
# Functions can return another function

def create_adder(x):
    def adder(y):
        return x+y

    return adder

add_15 = create_adder(15)

print (add_15(10))
```

Output:

```
25
```

Python Closures

Before seeing what a closure is, we have to first understand what nested functions and non-local variables are.

Nested functions in Python

A function that is defined inside another function is known as a nested function. Nested functions are able to access variables of the enclosing scope.
In Python, these non-local variables can be accessed only within their scope and not outside their scope. This can be illustrated by the following example:

```
# Python program to illustrate

# nested functions

def outerFunction(text):

    text = text


    def innerFunction():

        print(text)


    innerFunction()


if __name__ == '__main__':

    outerFunction('Hey!')
```

As we can see innerFunction() can easily be accessed inside the outerFunction body but not outside of it's body. Hence, here, innerFunction() is treated as nested Function which uses **text** as non-local variable.

Python Closures

A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

- It is a record that stores a function together with an environment: a mapping associating each free variable of the function (variables that are used locally but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.
- A closure—unlike a plain function—allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

```
# Python program to illustrate
```

```
# closures

def outerFunction(text):

    text = text


    def innerFunction():

        print(text)


    # Note we are returning function

    # WITHOUT parenthesis

    return innerFunction


if __name__ == '__main__':

    myFunction = outerFunction('Hey!')

    myFunction()
```

```
Output:
omkarpathak@omkarpathak-Inspiron-3542:
~/Documents/Python-Programs/$ python Closures.py
Hey!
```

1. As observed from the above code, closures help to invoke functions outside their scope.
2. The function **innerFunction** has its scope only inside the outerFunction. But with the use of closures, we can easily extend its scope to invoke a function outside its scope.

```
# Python program to illustrate

# closures

import logging

logging.basicConfig(filename='example.log',

                    level=logging.INFO)



def logger(func):

    def log_func(*args):

        logging.info(

            'Running "{}" with arguments {}'.format(func.__name__,

                                                    args))

        print(func(*args))


    # Necessary for closure to
```

```
    # work (returning WITHOUT parenthesis)
    return log_func


def add(x, y):
    return x+y


def sub(x, y):
    return x-y


add_logger = logger(add)
sub_logger = logger(sub)


add_logger(3, 3)
add_logger(4, 5)


sub_logger(10, 5)
sub_logger(20, 10)
```

```
OUTPUT:
omkarpathak@omkarpathak-Inspiron-3542:
~/Documents/Python-Programs/$ python MoreOnClosures.py
6
9
5
10
```

When and why to use Closures:

1. As closures are used as callback functions, they provide some sort of data hiding. This helps us to reduce the use of global variables.

2. When we have few functions in our code, closures prove to be an efficient way. But if we need to have many functions, then go for class (OOP).

Decorators in Python

Decorators are a very powerful and useful tool in Python since it allows programmers to modify the behaviour of function or class. Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it. But before diving deep into decorators let us understand some concepts that will come in handy in learning the decorators.

First Class Objects

In Python, functions are first class objects that mean that functions in Python can be used or passed as arguments.

Properties of first class functions:

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, …

Consider the below examples for better understanding.

Example 1: Treating the functions as objects.

```
# Python program to illustrate functions

# can be treated as objects

def shout(text):

    return text.upper()


print(shout('Hello'))


yell = shout


print(yell('Hello'))
```

Output:

```
HELLO
HELLO
```

In the above example, we have assigned the function shout to a variable. This will not call the function instead it takes the function object referenced by a shout and creates a second name pointing to it, yell.

Example 2: Passing the function as an argument

```
# Python program to illustrate functions

# can be passed as arguments to other functions

def shout(text):

    return text.upper()


def whisper(text):
```

```
    return text.lower()


def greet(func):
    # storing the function in a variable
    greeting = func("""Hi, I am created by a function passed as an
argument.""")
    print (greeting)


greet(shout)
greet(whisper)
```

Output:

```
HI, I AM CREATED BY A FUNCTION PASSED AS AN ARGUMENT.
hi, i am created by a function passed as an argument.
```

In the above example, the greet function takes another function as a parameter (shout and whisper in this case). The function passed as an argument is then called inside the function greet.

Example 3: Returning functions from another function.

```
# Python program to illustrate functions
# Functions can return another function


def create_adder(x):
    def adder(y):
        return x+y


    return adder


add_15 = create_adder(15)


print(add_15(10))
```

Output:

```
25
```

In the above example, we have created a function inside of another function and then have returned the function created inside.

The above three examples depict the important concepts that are needed to understand decorators. After going through them let us now dive deep into decorators.

Decorators

As stated above the decorators are used to modify the behaviour of function or class. In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

Syntax for Decorator:

```
@gfg_decorator
def hello_decorator():
    print("Gfg")

'''Above code is equivalent to -

def hello_decorator():
    print("Gfg")

hello_decorator = gfg_decorator(hello_decorator)'''
```

In the above code, gfg_decorator is a callable function, will add some code on the top of some another callable function, hello_decorator function and return the wrapper function.

Decorator can modify the behaviour**:**

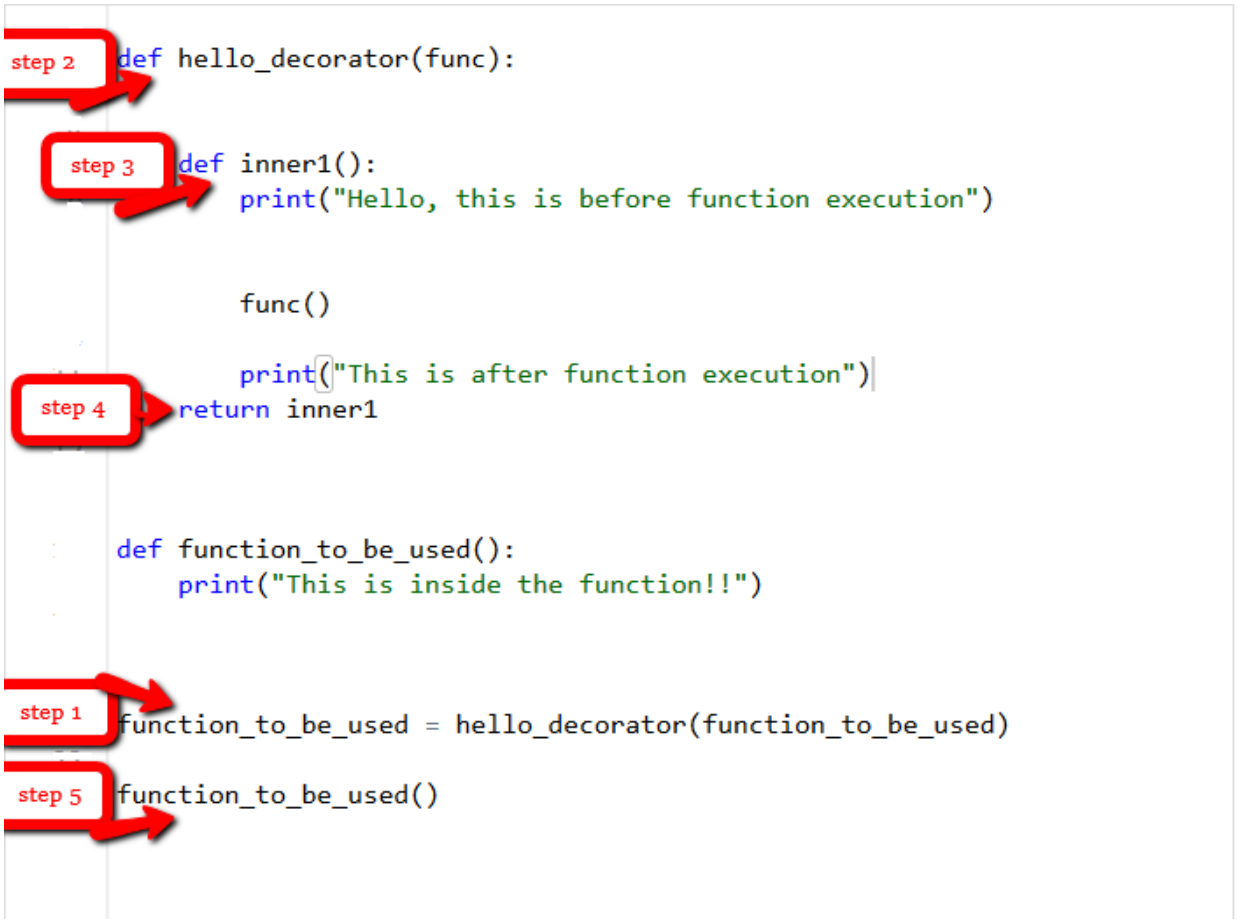
```
# defining a decorator

def hello_decorator(func):


    # inner1 is a Wrapper function in

    # which the argument is called


    # inner function can access the outer local

    # functions like in this case "func"

    def inner1():
        print("Hello, this is before function execution")


        # calling the actual function now

        # inside the wrapper function.

        func()


        print("This is after function execution")
```

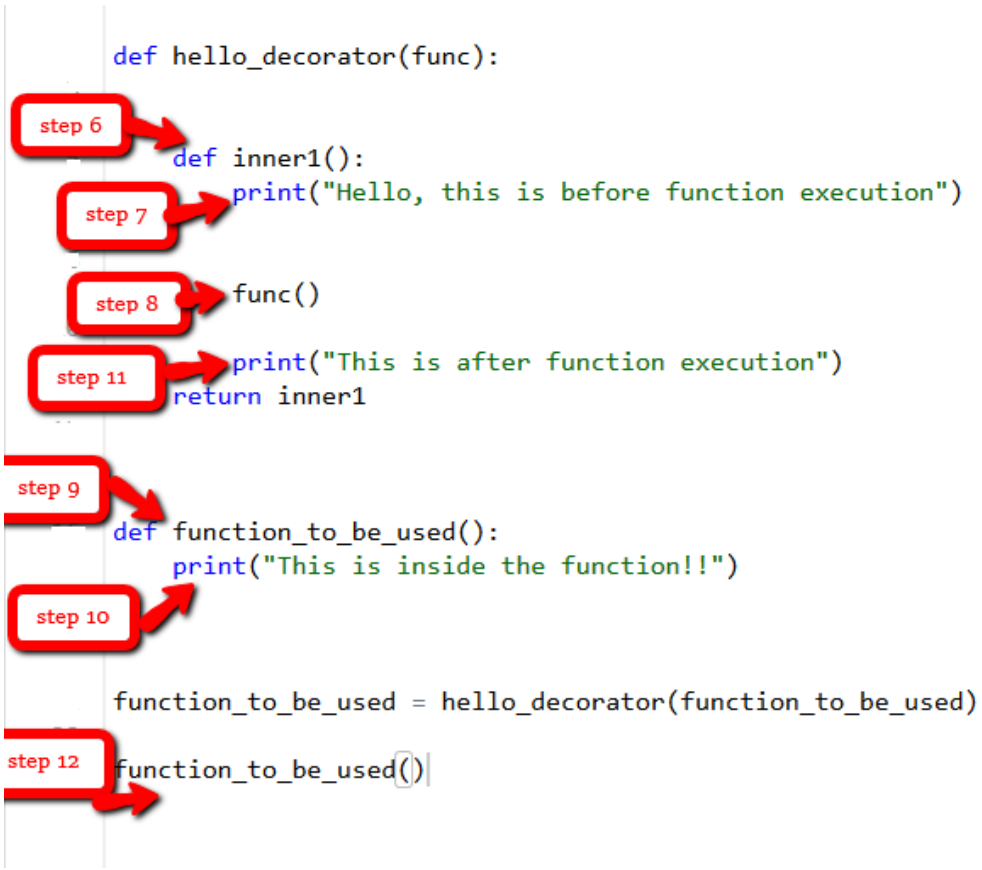
```
    return inner1


# defining a function, to be called inside wrapper
def function_to_be_used():
    print("This is inside the function !!")


# passing 'function_to_be_used' inside the
# decorator to control its behaviour
function_to_be_used = hello_decorator(function_to_be_used)


# calling the function
function_to_be_used()
```

Output:

```
Hello, this is before function execution
This is inside the function !!
This is after function execution
```

Let’s see the behaviour of the above code how it runs step by step when the “function_to_be_used” is called.


step 2
def hello_decorator(func):
step 3
def inner1():
print("Hello, this is before function execution")
func()
print("This is after function execution")
step 4
return inner1
def function_to_be_used():
print("This is inside the function!!")
step 1
function_to_be_used = hello_decorator(function_to_be_used)
step 5
function_to_be_used()


def hello_decorator(func):
step 6
def inner1():
step 7
print("Hello, this is before function execution")
step 8
func()
step 11
print("This is after function execution")
return inner1
step 9
def function_to_be_used():
print("This is inside the function!!")
step 10
function_to_be_used = hello_decorator(function_to_be_used)
step 12
function_to_be_used()

Let’s jump to another example where we can easily find out **the execution time of a function** using a decorator.

```
# importing libraries
import time
import math

# decorator to calculate duration
# taken by any function.
def calculate_time(func):

    # added arguments inside the inner1,
    # if function takes any arguments,
    # can be added like this.
    def inner1(*args, **kwargs):

        # storing time before function execution
        begin = time.time()

        func(*args, **kwargs)

        # storing time after function execution
        end = time.time()
        print("Total time taken in : ", func.__name__, end - begin)

    return inner1



# this can be added to any function present,
# in this case to calculate a factorial
@calculate_time
def factorial(num):

    # sleep 2 seconds because it takes very less time
```

```
    # so that you can see the actual difference
    time.sleep(2)
    print(math.factorial(num))


# calling the function.
factorial(10)
```

Output:

```
3628800
Total time taken in :  factorial 2.0061802864074707
```

What if a function returns something or an argument is passed to the function?

In all the above examples the functions didn't return anything so there wasn't any issue, but one may need the returned value.

```
def hello_decorator(func):
    def inner1(*args, **kwargs):

        print("before Execution")

        # getting the returned value
        returned_value = func(*args, **kwargs)
        print("after Execution")

        # returning the value to the original frame
        return returned_value

    return inner1


# adding decorator to the function
@hello_decorator
def sum_two_numbers(a, b):
    print("Inside the function")
    return a + b
```

```
a, b = 1, 2


# getting the value through return of the function
print("Sum =", sum_two_numbers(a, b))
```

Output:

```
before Execution
Inside the function
after Execution
Sum = 3
```

In the above example, you may notice a keen difference in the parameters of the inner function. The inner function takes the argument as *args and **kwargs which means that a tuple of positional arguments or a dictionary of keyword arguments can be passed of any length. This makes it a general decorator that can decorate a function having any number of arguments.

Chaining Decorators

In simpler terms chaining decorators means decorating a function with multiple decorators.

Example:

```
# code for testing decorator chaining
def decor1(func):
    def inner():
        x = func()
        return x * x
    return inner


def decor(func):
    def inner():
        x = func()
        return 2 * x
    return inner


@decor1
@decor
def num():
```

```
    return 10


print(num())
```

Output:

```
400
```

The above example is similar to calling the function as –

```
decor1(decor(num))
```

Decorators with parameters in Python

Prerequisite: Decorators in Python, Function Decorators

We know Decorators are a very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. In this article, we will learn about the ***Decorators with Parameters*** with help of multiple examples.
Python functions are First Class citizens which means that functions can be treated similarly to objects.

- Function can be assigned to a variable i.e they can be referenced.
- Function can be passed as an argument to another function.
- Function can be returned from a function.

Decorators with parameters is similar to normal decorators.

The syntax for decorators with parameters :

```
@decorator(params)
def func_name():
    ''' Function implementation'''
```

The above code is equivalent to

```
def func_name():
    ''' Function implementation'''

func_name = (decorator(params))(func_name)
"""
```

As the execution starts from left to right **decorator(params)** is called which returns a function object **fun_obj**. Using the fun_obj the call **fun_obj(fun_name)** is made. Inside the inner function, required operations are performed and the actual function reference is returned which will be assigned to **func_name**. Now, **func_name()** can be used to call the function with decorator applied on it.

How Decorator with parameters is implemented

```
def decorators(*args, **kwargs):
    def inner(func):
        '''
           do operations with func
        '''
        return func
    return inner #this is the fun_obj mentioned in the above content

@decorators(params)
def func():
    """
         function implementation
    """
```

Here **params** can also be empty.

Observe these first :

```
# Python code to illustrate
# Decorators basic in Python

def decorator_fun(func):
  print("Inside decorator")

  def inner(*args, **kwargs):
    print("Inside inner function")
    print("Decorated the function")
    # do operations with func

    func()

  return inner
```

```
@decorator_fun
def func_to():
    print("Inside actual function")

func_to()
```

Another Way:

```
# Python code to illustrate
# Decorators with parameters in Python

def decorator_fun(func):
  print("Inside decorator")

  def inner(*args, **kwargs):
    print("Inside inner function")
    print("Decorated the function")

    func()

  return inner



def func_to():
    print("Inside actual function")

# another way of using decorators
decorator_fun(func_to)()
```

Output:

```
Inside decorator
Inside inner function
Decorated the function
Inside actual function
```

Let’s move to another example:

Example #1:

```
# Python code to illustrate
# Decorators with parameters in Python

def decorator(*args, **kwargs):
    print("Inside decorator")

    def inner(func):

        # code functionality here
        print("Inside inner function")
        print("I like", kwargs['like'])

        func()

    # returning inner function
    return inner

@decorator(like = "geeksforgeeks")
def my_func():
    print("Inside actual function")
```

Output:

```
Inside decorator
Inside inner function
I like geeksforgeeks
Inside actual function
```

Example #2:

```
# Python code to illustrate
# Decorators with parameters in Python
```

```
def decorator_func(x, y):

    def Inner(func):

        def wrapper(*args, **kwargs):
            print("I like Geeksforgeeks")
            print("Summation of values - {}".format(x+y) )

            func(*args, **kwargs)

        return wrapper
    return Inner



# Not using decorator
def my_fun(*args):
    for ele in args:
        print(ele)

# another way of using decorators
decorator_func(12, 15)(my_fun)('Geeks', 'for', 'Geeks')
```

Output:

```
I like Geeksforgeeks
Summation of values - 27
Geeks
for
Geeks
```

This example also tells us that Outer function parameters can be accessed by the enclosed inner function.

Example #3:

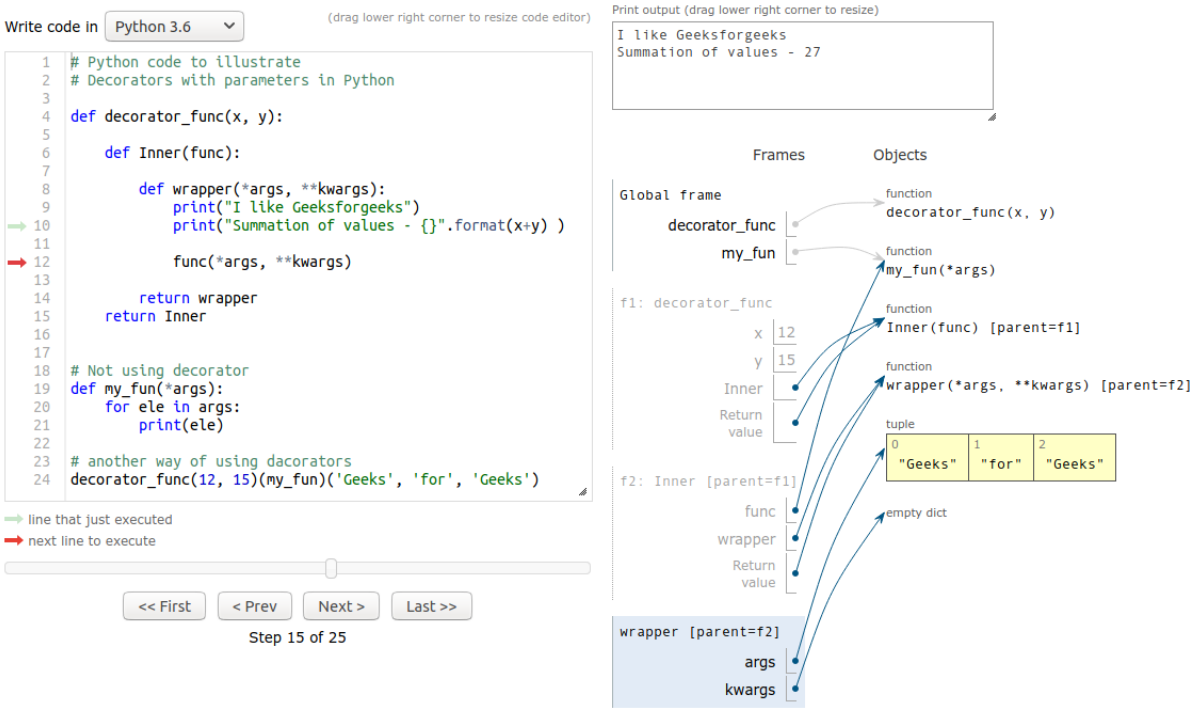
```
# Python code to illustrate
# Decorators with parameters in Python  (Multi-level Decorators)
```

```
def decodecorator(dataType, message1, message2):
    def decorator(fun):
        print(message1)
        def wrapper(*args, **kwargs):
            print(message2)
            if all([type(arg) == dataType for arg in args]):
                return fun(*args, **kwargs)
            return "Invalid Input"
        return wrapper
    return decorator


@decodecorator(str, "Decorator for 'stringJoin'", "stringJoin started ...")
def stringJoin(*args):
    st = ''
    for i in args:
        st += i
    return st


@decodecorator(int, "Decorator for 'summation'\n", "summation started ...")
def summation(*args):
    summ = 0
    for arg in args:
        summ += arg
    return summ


print(stringJoin("I ", 'like ', "Geeks", 'for', "geeks"))
print()
print(summation(19, 2, 8, 533, 67, 981, 119))
```

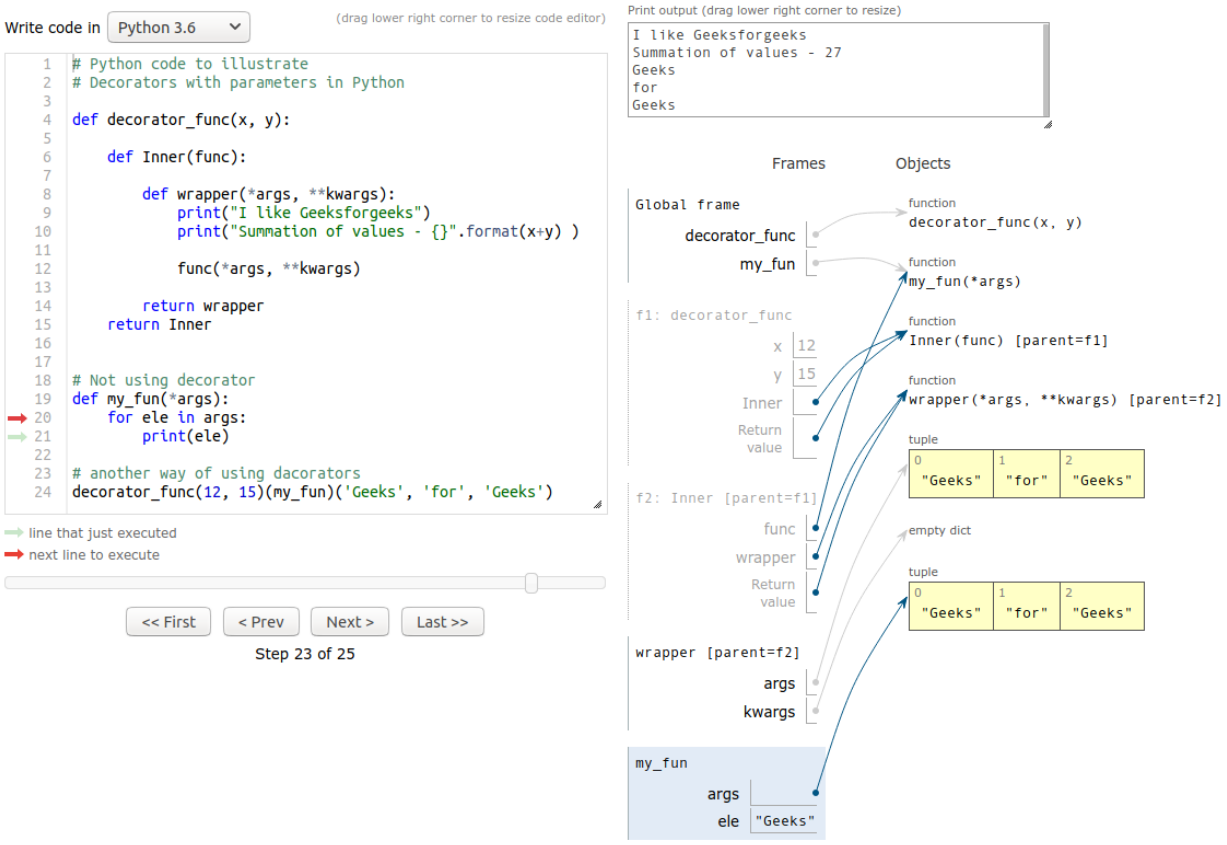
Output:

```
Decorator for 'stringJoin'
Decorator for 'summation'

stringJoin started ...
I like Geeksforgeeks

summation started ...
1729
```

1. Inside the Decorator



2. Inside the function



Note: Image snapshots are taken using PythonTutor.

Memoization using decorators in Python

- Difficulty Level : Medium
- Last Updated : 10 Nov, 2018

Recursion is a programming technique where a function calls itself repeatedly till a termination condition is met. Some of the examples where recursion is used are: calculation of fibonacci series, factorial etc. But the issue with them is that in the recursion tree, there can be chances that the sub-problem that is already solved is being solved again, which adds to an overhead.

Memoization is a technique of recording the intermediate results so that it can be used to avoid repeated calculations and speed up the programs. It can be used to optimize the programs that use recursion. In Python, memoization can be done with the help of function decorators.

Let us take the example of calculating the factorial of a number. The simple program below uses recursion to solve the problem:

```
# Simple recursive program to find factorial

def facto(num):
```

```
    if num == 1:
        return 1
    else:
        return num * facto(num-1)


print(facto(5))
```

The above program can be optimized by memoization using decorators.

```
# Factorial program with memoization using
# decorators.

# A decorator function for function 'f' passed
# as parameter
def memoize_factorial(f):
    memory = {}

    # This inner function has access to memory
    # and 'f'
    def inner(num):
        if num not in memory:
            memory[num] = f(num)
        return memory[num]

    return inner

@memoize_factorial
def facto(num):
    if num == 1:
        return 1
    else:
        return num * facto(num-1)

print(facto(5))
```

Explanation:
1. A function called *memoize_factoria*l has been defined. It's main purpose is to store the intermediate results in the variable called memory.
2. The second function called *facto* is the function to calculate the factorial. It has been annotated by a decorator(the function memoize_factorial). The *facto* has access to the *memory* variable as a result of the concept of closures.The annotation is equivalent to writing,

```
facto = memoize_factorial(facto)
```

3. When facto(5) is called, the recursive operations take place in addition to the storage of intermediate results. Every time a calculation needs to be done, it is checked if the result is available in *memory*. If yes, then it is used, else, the value is calculated and is stored in *memory*.
4. We can verify the fact that memoization actually works, please see output of this program.