Decorators

Recall that:

A function can define a new function inside itself as well as return the function.

```
def f(x):
    def sq(z):
        return z*z
    return sq(x)
```

```
>>>
>>> f(3)
9
>>>
```

A function is a first-class object and so can be assigned to a variable.

```
def f(x):
    def sq(z):
        return z*z
    return sq(x)

g=f
print(g(3))
```

We get

```
>>> g(3)
9
```

The name of a function is a pointer to the function object.

So, if we write sum=0, this breaks the connection to the built-in sum function.

These ideas allow us to define "decorator" functions in Python.

A decorator is a function that creates a “wrapper” around another function.

In Python, a decorator is a function that takes another function and extends or modifies its behavior without explicitly modifying its code.

The primary purpose of this wrapping is to alter or enhance the behavior of the original function without altering it in any way.

For example, you might want to print a message whenever a given function is entered and again when it is exited. You could accomplish this by changing the functions by adding the appropriate print functions. But … say you don’t have access allowing you to modify the function so this method wont work.

Or say you are debugging a system and you would like to have many different functions report on their entry and exit. It would be time consuming to modify all of them. Among other uses, decorators provide an answer to this problem.

Example:

The function

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

adds two numbers and returns the sum.

Say that you would like to print a message when add_numbers is entered and when it exits. You can do it with decorators like this:

1. Define a function (for this example call the function “log_function_call”) that takes the original function (call original function func) as an argument and makes a “sandwich” around f that
    a. Prints that func has entered
    b. Calls func to do the work
    c. Prints that func has exited.

    We call the sandwich a “wrapper”.

    The “log_function_call” function returns the wrapper functions. This is easier to understand by just looking at the code.

This is the decorator function

```
def log_function_call(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function {func.__name__}")
        result = func(*args, **kwargs)
        print(f"Finished calling function {func.__name__}")
        return result
    return wrapper
```

This is the function to be decorated

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

This says that the log_function_call function will “decorate” the add function

```
@log_function_call
def add_numbers(x, y):
    return x + y
```

Now you call add_number(2,3)

```
result = add_numbers(2, 3)
print(result)
```

This is equivalent to

```
add_numbers= log_function_call(add_numbers(2,3))
result = add_numbers(2, 3)
print(result)
```

In general:

Syntactically, decorators are denoted using the special `@` symbol as follows:

```
@decorate
def func(x):
    ...
```

The preceding code is shorthand for the following:

```
def func(x):
    ...
func = decorate(func) # decorate is the name of the decorator in this example
```

So the original function name is associated with the decorated function.

In the example, a function `func()` is defined.

- However, immediately after its definition, the function object itself is passed to the function **`decorate()`**,
- which returns an object that **replaces** the original `func`. (i.e. the new object is assigned to the original name func)

We can run this in the emulator to see exactly the control path during program execution.

There is a problem, however. In practice, functions also contain metadata such as the function name, doc string, and type hints. If you put a wrapper around a function, this information gets hidden. When writing a decorator, it's considered best practice to use the `@wraps()` decorator, for example:

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        """ wrapper"""
        print("Before the function is called.")
        result = func(*args, **kwargs)
        print("After the function is called.")
        return result
    return wrapper

@my_decorator
def my_function(x,y):
    """ my func"""
    return x+y

z=my_function(2,3)
print(z,my_function.__name__)
print(z,my_function.__doc__)
```

When I run this, I expect the function name to be "my_function" and the doc string to be """ my func""". **But** here is what I get:

```
Before the function is called.
After the function is called.
5 wrapper  Wrong function name
5  wrapper Wrong doc string
>>>
```

The solution

`from functools import wraps`

The `@wraps()` decorator copies various function metadata to the replacement function. In this case, metadata from the given function `func()` is copied to the returned wrapper function `call()`.

So now we have:

```
import functools


def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        """ wrapper"""
        print("Before the function is called.")
        result = func(*args, **kwargs)
        print("After the function is called.")
        return result
    return wrapper


@my_decorator
def my_function(x,y):
    """ my func"""
    return x+y
```

```
z=my_function(2,3)

print(z,my_function.__name__)  # Output: "my_function"

print(z,my_function.__doc__)
```

When decorators are applied, they must appear on their own line immediately prior to the function.

Now that we have the decorator we can apply it to any function we like. Say you have a function

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

I can decorate it like this:

```
import functools

def my_decorator(func):
   @functools.wraps(func)
   def wrapper(*args, **kwargs):
      """ wrapper"""
      print("Before the function is called.")
      result = func(*args, **kwargs)
      print("After the function is called.")
      return result
   return wrapper


@my_decorator
def mult(x,y):
   """ mult func"""
   return x*y

z=mult(2,3)
print(z,mult.__name__)  # Output: "my_function"
print(z,mult.__doc__)
```

and I get

```
Before the function is called.
After the function is called.
6 mult
6  mult func
```

>>> >>>

Problem:

Write a decorator that when applied to a function will keep track of how many times that function has been called. It will do this by keeping count of the calls to the decorated functions in a dictionary passed to the decorator.

```
calldict={} # calldict is the dictionary

@countcalls(calldict) #countcalls is the decorator
def add(x,y):
    return x+y

@countcalls(calldict)
def mult(x,y):
    return x*y

z=add(2,3)
print(calldict)
z=add(2,3)
print(calldict)
z=mult(2,3)
print(calldict)
```

When I run the above code, I get:

```
{'add': 1}
{'add': 2}
{'add': 2, 'mult': 1}
>>>
```

Write the countcalls decorator.

Solution:

Another example. A decorator to return the runtime of the decorated functions.

Some background.

Python has extensive libraries for dealing with dates and times.

The **time** library in Python provides various time-related functions. It allows you to measure time intervals in seconds, determine the current time, and perform conversions between different time formats.

Example:

```
import time

def my_function():
    time.sleep(2)  # simulate a 2 second delay
    return "Hello, world!"

start_time = time.time()
result = my_function()
end_time = time.time()

duration = end_time - start_time

print(f"Result: {result}")
print(f"Duration: {duration} seconds")
```

When I run this code I get:

```
Result: Hello, world!
Duration: 2.0110862255096436 seconds
```

In the above example, the **time.time()** function is used to get the current time in seconds since the epoch (January 1, 1970). The **my_function()** function is then executed, with a 2 second delay simulated using the **time.sleep()** function. The start and end times are recorded, and the duration of the function execution is calculated by subtracting the start time from the end time.

Another useful function in the time library is **timeit**, which provides a simple way to time the execution of a function with **higher precision**:

```
import timeit

def my_function():
    time.sleep(2)  # simulate a 2 second delay
    return "Hello, world!"

duration = timeit.timeit(my_function, number=1)

print(f"Duration: {duration} seconds")
```

The **timeit** module is specifically designed to measure the execution time of small code snippets with high accuracy. It provides a **timeit()** function that takes a Python statement as input and executes it a number of times to measure the average execution time.

For example:

```
import timeit

def my_function():
    for i in range(1000000):
        pass

# timeit can be used as a standalone function
time_taken = timeit.timeit(my_function, number=100)

print("Execution time:", f"{time_taken:.4f} seconds")
```

The result is
Execution time: 1.8632 seconds

Another example:

```
import timeit

def my_function():
    sum = 0
    for i in range(1000000):
```

```
        sum += i
    return sum

# Time the execution of my_function 1000 times
t = timeit.timeit(my_function, number=1000)

print(f"Execution time: {t:.6f} seconds")
```

Problem:

Lets compare the runtime of two sorting algorithms.

The first sort: bubble sort. Write it and sort 1,000 random integers.

```
def bubble_sort(arr):
```

The second sort: selection sort. Write it and sort 1,000 random integers.

Write a decorator time_it so that when it decorates a sort function, it sorts the list passed to the function and returns the runtime.

Here is the test:

```
arr = [random.randint(0, 100) for i in range(1000)]

sorted_arr1 = bubble_sort(arr.copy())

sorted_arr2 = selection_sort(arr.copy())

name=""
if sorted_arr1>sorted_arr2:
    name="Selection sort"
    diff=sorted_arr1-sorted_arr2
else:
    name="Bubble sort"
    diff = sorted_arr2-sorted_arr1

print(name+' is faster by '+str(diff)+' seconds.')
```

Problem:

Here is the code for merge sort. Time it and see how it compares to the above two sorts.

```
@time_it
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
```

```
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    while i < len(L):
        arr[k] = L[i]
        i += 1
        k += 1

    while j < len(R):
        arr[k] = R[j]
        j += 1
        k += 1
return arr
```

Problem:

What happened. Now what?