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.

Note: for a nice introduction see https://www.geeksforgeeks.org/decorators-in-python/ or https://www.programiz.com/python-programming/decorator

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 when you call the original function without changing the code of the original function 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: (run this)

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_whee():
    print("Whee!")

say_whee = my_decorator(say_whee)
```

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
```

Understanding the process

Functions in Python are decorated at the time they are defined, not when they are called. The decoration process is part of the function definition and occurs immediately after the function object is created, but before the function is actually called for the first time.

Here's how it works:

1. Function Definition: When Python encounters a function definition, it creates a function object. This includes parsing the function's code and setting up its name, parameters, and the code to execute.
2. Applying the Decorator: If the function definition is preceded by one or more decorator expressions, each decorator is applied as soon as the function object is created. A decorator is essentially a callable that takes a function object as an argument and returns a new function object. The original function object is replaced by the one returned by the decorator.
3. Final Function Object: The final function object, which might have been modified or completely replaced by the decorators, is then bound to the function's name in the current namespace.
4. Function Calls: When the decorated function is later called, it's the modified version of the function (as returned by the decorator) that is executed, not the original version.

So, for example

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

# At this point, say_hello is already decorated
say_hello()
```

in this example

- The say_hello function is defined.
- Immediately after its definition, my_decorator is applied to say_hello.

- The my_decorator function takes say_hello as an argument, wraps additional functionality around it, and returns the wrapper function.
- The name say_hello now refers to the wrapper function, not the original say_hello function.
- When say_hello() is called, it's actually the wrapper function that gets executed.

This illustrates that the decoration process is part of the function's creation and definition phase, not the call phase.

Note also that the **decorated function is its own object**, distinct from the original function object. When a function is decorated, the decorator takes the original function object as an input and typically returns a new function object. This new function object usually wraps or modifies the behavior of the original function. **As a result, the original and decorated functions are two different function objects.**

In general

1. Original Function Object:
   - When you define a function, Python creates a function object for it. This object represents the function with its original behavior.
2. Applying the Decorator:
   - A decorator is a callable (usually another function) that takes a function object as an argument and returns a new function object.
   - The decorator may add some functionality to the original function, modify it, or even completely replace it with another function.
3. New Function Object (Decorated Function):
   - The object returned by the decorator is now the decorated function. This new function object is what is accessible using the original function's name after decoration.
   - The new function object can retain a reference to the original function object, allowing it to invoke the original function's behavior within the new behavior.
4. Two Distinct Objects:
   - The original function object and the new (decorated) function object are separate objects in memory. *
   - If you retain a separate reference to the original function (before it is decorated), you can still access its undecorated behavior.

Here is one way to keep the original function (other than defining it twice):

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Decorator adds this before the function call")
        result = func(*args, **kwargs)
        print("Decorator adds this after the function call")
        return result
```

```
    # Provide access to the undecorated function via an attribute of the wrapper
    wrapper.original = func
    return wrapper

# Applying the decorator using the @ notation
@my_decorator
def original_function(x, y):
    print(f"Original function called with arguments: {x}, {y}")
    return x + y

# Using the decorated function
print("Calling decorated function:")
original_function(5, 10)

# Using the undecorated function via an attribute of the decorated function
print("\nCalling undecorated function:")
original_function.original(5, 10)
```

You're right; duplicating a large function to retain both its decorated and undecorated versions can be messy and not ideal in terms of maintainability. Fortunately, there are cleaner ways to achieve this without duplicating the entire function definition. One common approach is to pass the original function to the decorator and have the decorator itself provide a way to access the undecorated function.

Let's illustrate this with a modified example:

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Decorator adds this before the function call")
        result = func(*args, **kwargs)
        print("Decorator adds this after the function call")
        return result

    # Provide access to the undecorated function via an attribute of the wrapper
    wrapper.original = func
    return wrapper

# Applying the decorator using the @ notation
@my_decorator
def original_function(x, y):
    print(f"Original function called with arguments: {x}, {y}")
    return x + y

# Using the decorated function
print("Calling decorated function:")
```

original_function(5, 10)

# Using the undecorated function via an attribute of the decorated function
print("\nCalling undecorated function:")
original_function.original(5, 10)

here:
- The decorator `my_decorator` is modified to attach the original, undecorated function to the wrapper function as an attribute (`wrapper.original = func`).
- When `original_function` is decorated, you can still access its original, undecorated form through `original_function.original`.
- This approach avoids duplicating the function's code and provides a clean and maintainable way to access both versions of the function.

This method is more elegant than using two definitions of the function and is particularly useful for large functions or when you want to preserve both behaviors without cluttering the codebase.

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:

```
def countcalls(calldict):
    def decorator(func):
        def wrapper(*args, **kwargs):
            # Increment the count for this function in calldict
            calldict[func.__name__] = calldict.get(func.__name__, 0) + 1
            # Call the original function
            return func(*args, **kwargs)
        return wrapper
    return decorator
```

When we run the program:

```
calldict = {}

@countcalls(calldict)
def add(x, y):
    return x + y


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

# Testing the functions and printing calldict
z = add(2, 3)
print(calldict)  # Output: {'add': 1}
z = add(2, 3)
print(calldict)  # Output: {'add': 2}
z = mult(2, 3)
print(calldict)  # Output: {'add': 2, 'mult': 1}
```

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. There is also timeit.

Excursus

`time.time()` and `timeit.timeit()` are both used for measuring time in Python, but they serve different purposes and operate in slightly different ways:

1. `time.time()`:

Purpose: `time.time()` is a function in the `time` module. It returns the current time in seconds since the Epoch (Jan 1, 1970, at 00:00:00 UTC), commonly known as Unix time.

Usage: It's typically used for getting the current timestamp or measuring the duration of an event by calculating the difference in time before and after the event. For example, to measure how long a piece of code takes to execute, you would record the time before and after its execution and calculate the difference.

Precision: The precision of `time.time()` can vary based on the system and platform. It's generally suitable for most practical purposes but might not be the best choice for microbenchmarking or where very high precision is needed.

2. `timeit.timeit()`:

Purpose: `timeit.timeit()` is a function in the `timeit` module. It's specifically designed for measuring the execution time of small code snippets. It provides a more accurate and reliable way of timing code than `time.time()`, particularly for short code snippets where the execution time is very brief.

Usage: `timeit.timeit()` runs the code snippet multiple times (default is 1,000,000 times) in a controlled environment and returns the total time taken for all executions. This repeated execution helps to average out any fluctuations in time due to systemrelated anomalies.

Setup: `timeit.timeit()` also allows you to specify setup code that runs once before the timing runs start. This is useful for setting up the environment without including this setup time in the final timing.

Precision: `timeit.timeit()` is more precise for timing short code snippets, as it minimizes the impact of external factors on the timing and averages out the time over multiple runs.

Examples

Using `time.time()`:

```
import time

  start = time.time()
  # Your code to time
  end = time.time()
  print(f"Duration: {end  start} seconds")
```

Using `timeit.timeit()`:

```
 import timeit

  duration = timeit.timeit('code_to_time()', setup='from __main__ import code_to_time',
number=1000)
  print(f"Total duration for 1000 runs: {duration} seconds")
```

So, ingeneral, use `time.time()` for general-purpose timing and longer running processes, and `timeit.timeit()` for more accurate timing of short code snippets, especially when conducting performance tests or benchmarks.

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
```

```
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:

Let’s compare the runtime of four sorting algorithms: bubble sort, insertion sort, quicksort and merge sort.

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

Answer:

```
import time
import functools
import matplotlib.pyplot as plt
import random

# Timer decorator to measure execution time
def timer(func):
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.time()
        value = func(*args, **kwargs)
        end_time = time.time()
        runtime = end_time - start_time
        #print(f"Runtime of {func.__name__}: {runtime:.6f} seconds")
        return runtime, value  # Return both runtime and the sorted array
    return wrapper_timer

# Bubble Sort
@timer
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

# Selection Sort
@timer
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[min_idx] > arr[j]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
return arr
```

```
# Merge Sort
@timer
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

# Quick Sort
@timer
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]

        # Apply quick_sort and unpack the results to ignore the timing
        _, sorted_less = quick_sort(less)
        _, sorted_greater = quick_sort(greater)

        return sorted_less + [pivot] + sorted_greater
```

```
# Generate test data
data_size = 10000  # Increase the data size to get more accurate results
test_data = [random.randint(1, 10000) for _ in range(data_size)]

# Dictionary to hold the results
times = {}

# Run each sorting algorithm and store the results
times['Bubble Sort'], _ = bubble_sort(test_data.copy())
times['Selection Sort'], _ = selection_sort(test_data.copy())
times['Merge Sort'], _ = merge_sort(test_data.copy())
times['Quick Sort'], _ = quick_sort(test_data.copy())

print("Runtimes")
print('Bubble Sort:',times['Bubble Sort'])
print('Selection Sort:',times['Selection Sort'])
print('Merge Sort:',times['Merge Sort'])
print('Quick Sort:',times['Quick Sort'])

# Plotting the results
plt.bar(times.keys(), times.values(), color=['red', 'green', 'blue', 'cyan'])
plt.ylabel('Time (seconds)')
plt.xlabel('Sorting Algorithm')
plt.title('Sorting Algorithm Performance')
plt.show()

# Plotting the results with a logarithmic scale
plt.bar(times.keys(), times.values(), color=['red', 'green', 'blue', 'cyan'])
plt.yscale('log')  # Set the y-axis to a logarithmic scale
plt.ylabel('Time (seconds)')
plt.xlabel('Sorting Algorithm')
plt.title('Sorting Algorithm Performance')
plt.show()
```

Finally, let's make sure that we understand the flow of control when decorators are used.

The following example is from:

https://www.geeksforgeeks.org/decorators-in-python/#

```
# 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()
```

```
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()
```