Function generator and decorator

November 26, 2022

1 Generators & Decorators

1.1 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.

[1]:
```
# 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)
```

```
1
2
3
```

1.2 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).

[2]:
```
# 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(next(x)) # In Python 3, __next__()
print(next(x))
print(next(x))
```

```
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.

[3]:
```
# 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(next(x)) # In Python 3, __next__()
print(next(x))
print(next(x))
print(next(x))
print(next(x))

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

```
0
1
1
2
3

Using for in loop
0
```

```
1
1
2
3
```

1.3 Decorators:

Decorators are a very powerful and useful tool in Python since it allows programmers to modify the behaviour of a 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.

1.3.1 First Class Objects

In Python, functions are first class objects which means 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.

[7]:
```
# Python program to illustrate functions
# can be treated as objects
def shout(text):
    return text.upper()

print(shout('Hello'))

yell = shout

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

```
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

[8]:
```
# 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)
```

```
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.

[9]:
```
# 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))
```

```
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.

1.3.2 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.

[10]:
```
# 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()
```

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

[11]:
```
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))
```

```
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.