### **First Steps to Decorators**

We know from our various Python training classes that there are some points in the definitions of decorators, where many beginners get stuck.

Therefore, we will introduce decorators by repeating some important aspects of functions. First you have to know or remember that function names are references to functions and that we can assign multiple names to the same function:

**def** succ(x):

**return** x + 1

successor = succ

successor(10)

**Output:**

11

succ(10)

**Output:**

11

This means that we have two names, i.e. "succ" and "successor" for the same function. The next important fact is that we can delete either "succ" or "successor" without deleting the function itself.

**del** succ

successor(10)

**Output:**

11

### 2. Functions inside Functions

Python’s concept of having or defining functions inside of a function is completely new to C or C++ programmers:

**def** f():

**def** g():

print("Hi, it's me 'g'")

print("Thanks for calling me")

print("This is the function 'f'")

print("I am calling 'g' now:")

g()

f()

**Output:**

This is the function 'f'

I am calling 'g' now:

Hi, it's me 'g'

Thanks for calling me

Another example using "proper" return statements in the functions:

**def** temperature(t):

**def** celsius2fahrenheit(x):

**return** 9 \* x / 5 + 32

result = "It's " + str(celsius2fahrenheit(t)) + " degrees!"

**return** result

print(temperature(20))

**Output:**

It's 68.0 degrees!

### 3. Functions as Parameters

If you solely look at the previous examples, this doesn't seem to be very useful. It gets useful in combination with two further powerful possibilities of Python functions. Due to the fact that every parameter of a function is a reference to an object and functions are objects as well, we can pass functions - or better "references to functions" - as parameters to a function. We will demonstrate this in the next simple example:

**def** g():

print("Hi, it's me 'g'")

print("Thanks for calling me")

**def** f(func):

print("Hi, it's me 'f'")

print("I will call 'func' now")

func()

f(g)

**Output:**

Hi, it's me 'f'

I will call 'func' now

Hi, it's me 'g'

Thanks for calling me

You may not be satisfied with the output. 'f' should write that it calls 'g' and not 'func'. Of course, we need to know what the 'real' name of func is. For this purpose, we can use the attribute \_\_name\_\_, as it contains this name:

def g():

print("Hi, it's me 'g'")

print("Thanks for calling me")

def f(func):

print("Hi, it's me 'f'")

print("I will call 'func' now")

func()

print("func's real name is " + func.\_\_name\_\_)

f(g)

**Output:**

Hi, it's me 'f'

I will call 'func' now

Hi, it's me 'g'

Thanks for calling me

func's real name is g

The output explains what's going on once more. Another example:

**import** **math**

**def** foo(func):

print("The function " + func.\_\_name\_\_ + " was passed to foo")

res = 0

**for** x **in** [1, 2, 2.5]:

res += func(x)

**return** res

print(foo(math.sin))

print(foo(math.cos))

**Output:**

The function sin was passed to foo

2.3492405557375347

The function cos was passed to foo

-0.6769881462259364

### 4. Functions returning Functions

The output of a function is also a reference to an object. Therefore functions can return references to function objects.

**def** f(x):

**def** g(y):

**return** y + x + 3

**return** g

nf1 = f(1)

nf2 = f(3)

print(nf1(1))

print(nf2(1))

**Output:**

5

7

We will implement a polynomial "factory" function now. We will start with writing a version which can create polynomials of degree 2.



The Python implementation as a polynomial factory function can be written like this:

**def** polynomial\_creator(a, b, c):

**def** polynomial(x):

**return** a \* x\*\*2 + b \* x + c

**return** polynomial

p1 = polynomial\_creator(2, 3, -1)

p2 = polynomial\_creator(-1, 2, 1)

**for** x **in** range(-2, 2, 1):

print(x, p1(x), p2(x))

**Output:**

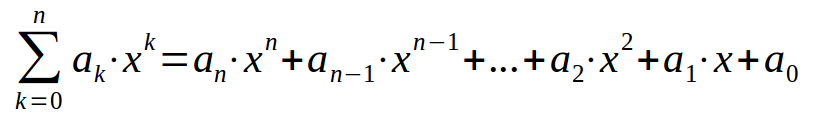
-2 1 -7

-1 -2 -2

0 -1 1

1 4 2

We can generalize our factory function so that it can work for polynomials of arbitrary degree:



**def** polynomial\_creator(\*coefficients):

*""" coefficients are in the form a\_n, ... a\_1, a\_0*

*"""*

**def** polynomial(x):

res = 0

**for** index, coeff **in** enumerate(coefficients[::-1]):

res += coeff \* x\*\* index

**return** res

**return** polynomial

p1 = polynomial\_creator(4)

p2 = polynomial\_creator(2, 4)

p3 = polynomial\_creator(1, 8, -1, 3, 2)

p4 = polynomial\_creator(-1, 2, 1)

**for** x **in** range(-2, 2, 1):

print(x, p1(x), p2(x), p3(x), p4(x))

**Output:**

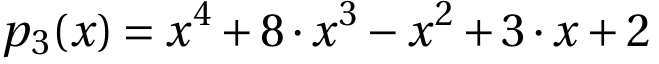
-2 4 0 -56 -7

-1 4 2 -9 -2

0 4 4 2 1

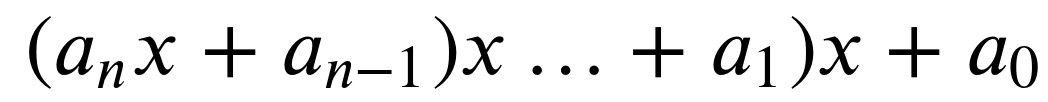
1 4 6 13 2

The function p3 implements, for example, the following polynomial:



The polynomial function inside of our decorator polynomial\_creator can be implemented more efficiently. We can factorize it in a way so that it doesn't need any exponentiation.

Factorized version of a general polynomial without exponentiation:



Implementation of our polynomial creator decorator avoiding exponentiation:

**def** polynomial\_creator(\*coeffs):

*""" coefficients are in the form a\_n, a\_n\_1, ... a\_1, a\_0*

*"""*

**def** polynomial(x):

res = coeffs[0]

**for** i **in** range(1, len(coeffs)):

res = res \* x + coeffs[i]

**return** res

**return** polynomial

p1 = polynomial\_creator(4)

p2 = polynomial\_creator(2, 4)

p3 = polynomial\_creator(1, 8, -1, 3, 2)

p4 = polynomial\_creator(-1, 2, 1)

**for** x **in** range(-2, 2, 1):

print(x, p1(x), p2(x), p3(x), p4(x))

**Output:**

-2 4 0 -56 -7

-1 4 2 -9 -2

0 4 4 2 1

1 4 6 13 2

If you want to learn more about polynomials and how to create a polynomial class, you can continue with our chapter on [Polynomials](https://www.python-course.eu/polynomial_class_in_python.php).

### 5. A Simple Decorator

Now we have everything ready to define our first simple decorator:

**def** our\_decorator(func):

**def** function\_wrapper(x):

print("Before calling " + func.\_\_name\_\_)

func(x)

print("After calling " + func.\_\_name\_\_)

**return** function\_wrapper

**def** foo(x):

print("Hi, foo has been called with " + str(x))

print("We call foo before decoration:")

foo("Hi")

print("We now decorate foo with f:")

foo = our\_decorator(foo)

print("We call foo after decoration:")

foo(42)

**Output:**

We call foo before decoration:

Hi, foo has been called with Hi

We now decorate foo with f:

We call foo after decoration:

Before calling foo

Hi, foo has been called with 42

After calling foo

If you look at the output of the previous program, you can see what's going on. After the decoration "foo = our\_decorator(foo)", foo is a reference to the 'function\_wrapper'. 'foo' will be called inside of 'function\_wrapper', but before and after the call some additional code will be executed, i.e. in our case two print functions.