Introduction

A **decorator** is a function that takes as input a function and returns a function. Using decorators is done in a generic manner: it can be applied quickly to existing functions to have them behave a bit differently.

Simple example

Complete syntax

Code

In [1]:

```
def custom_decorator(function):
    def new_function(*args, **kwargs):
        print(f"This message is printed because `{function.__name__}` was decorated u
        result = function(*args, **kwargs)
        return result
    return new_function

def basic_function(a, b):
    print(f"We are in `basic_function`: {a}, {b}")


new_function = custom_decorator(basic_function)
new_function(3, 4)
```

```
This message is printed because `basic_function` was decorated using `custom
_decorator`.
We are in `basic_function`: 3, 4
```

Loading [MathJax]/extensions/Safe.js

Explanation

A new function `new_function` is defined in the body of `custom_decorator`. It achieves some work (`print`) and then call the function passed as an argument (`basic_function`). This new funtion is returned and can be stored in a variable.

Notes

A decorator cannot easily modified what happens **inside** the function. Yet, it can modify the arguments it takes as input and the output it returns

Lighter syntax

The same can be achievd without using an intermediate `new_function` variable: the instruction `@decorator_name` is placed the line preceding the `def` keywork of a function to decorate.

In [2]:
```
@custom_decorator
def basic_function(a, b):
    print(f"We are in `basic_function`: {a}, {b}")

basic_function(3, 4)
```

```
This message is printed because `basic_function` was decorated using `custom
_decorator`.
We are in `basic_function`: 3, 4
```

Internally, Python replaces `basic_function` by its decorated version.

Loading [MathJax]/extensions/Safe.js

Fictive use case

In the example below, a `prod`, a `sum` and a `pow` functions are defined. They perform the sum/product/iterative power of elements of the iterable they receive.

In [3]:

```python
def sum_(var):
    return sum(var)

def prod_(var):
    p = 1
    for e in var:
        p *= e
    return p

def pow_(var):
    if var:
        p = var[0]
        for e in var[1:]:
            p = p ** e
        return p
    else:
        return 1
```

Now, we want these functions to return 0 whenever the result is negative. The common way is to modify these functions directly:

In [4]:
```python
def sum2_(var):
    s = sum(var)
    return max(s, 0)

def prod2_(var):
    p = 1
    for e in var:
        p *= e
    return max(p, 0)

def pow2_(var):
    if var:
        p = var[0]
        for e in var[1:]:
            p = p ** e
        return max(p, 0)
    else:
        return 1
```

But with these modifications, the body of the functions is modified and the functions do not do anymore what they were first intended to do. Instead, one could use an decorator:

In [5]:
```python
def only_positive(func):
    def new_func(*args, **kwargs):
        result = func(*args, **kwargs)
        return max(result, 0)
    return new_func
```

In [6]:

```
@only_positive
def sum_p(var):
    return sum_(var)

@only_positive
def prod_p(var):
    return prod_(var)

@only_positive
def pow_p(var):
    return pow_(var)
```

In [7]:

```
var = [-3, 5, 3]
print(sum_(var), prod_(var), pow_(var))
print(sum_p(var), prod_p(var), pow_p(var))
```

```
5 -45 -14348907
5 0 0
```

Advanced

This is handy, but the functions are still modified a bit and we have to comment the decorator line to remove the special behavior.

Instead, we will define a **decorator with argument** to decide whether we want this special behavior to apply. To this purpose, `set_only_positive` is a function that returns a decorator, depending on whether we want (`positive=True`) or do not want (`positive=False`) to apply the special behaviour on the to-be-decorated function.

In [8]:

```python
def set_only_positive(positive):
    if positive:
        return only_positive
    else:
        def no_decorator(func):
            return func
        return no_decorator

positive_results = True

@set_only_positive(positive_results)
def sum_p(var):
    return sum_(var)

@set_only_positive(positive_results)
def prod_p(var):
    return prod_(var)

@set_only_positive(positive_results)
def pow_p(var):
    return pow_(var)
```

In [9]:
```
var = [-3, 5, 3]
print(sum_(var), prod_(var), pow_(var))
print(sum_p(var), prod_p(var), pow_p(var))
```

```
5 -45 -14348907
5 0 0
```

Conclusion

Decorators are advanced features of the python language. In most cases, it can be replaced by (more ugly) simpler solutions. But they are very common in some common libraries, thus it is important to understand the meaning of the `@decorator` syntax.

In [ ]:

```
```

Loading [MathJax]/extensions/Safe.js

Loading [MathJax]/extensions/Safe.js