PYTHON NOTES

ON

DECORATORS

BY

0xLeo (github.com/0xleo)

JUNE 5, 2021

DRAFT X.YY
MISSING: ...

Contents

1 Higher-order functions, decorators, properties

1.1 Functions as first-class citizens

Functions in Python are `first-class citizens`. According to Popplestone:

> **Definition 1.1** (first-class citizen (FCC)). *A function is a first-class citizen if all of the below are true*
>
> 1. *It can be the actual parameters of functions.*
> 2. *It can be returned as results of functions.*
> 3. *It can be the subject of assignment statements.*
> 4. *It can be tested for equality.*

Therefore if we assign a FCC function to a variable, then the variable acquires the function's properties. The example below illustrates this and of course prints `<b>some text</b>`. Python allows functions to be assigned to variables since both functions and variables are objects.

```
def makebold(text: str):
    return '<b>' + text + '</b>'

bold = makebold # function gets assigned to a variable
print(bold('some text'))
```

The assigned variable (i.e. `bold`) holds a *ready-to-use* (*ready to call*) copy of the function object and it can call it on any arguments supported by the original function (i.e. `makebold`), acting as a proxy of it.

1.2 Higher-order functions

Another important concept is `higher-order functions` (HOF).

> **Definition 1.2** (higher-order function (HOF)). *A first-class citizen function that takes another function as parameter or returns a (callable) function is called higher-order function.*

Therefore HOFs operate on other functions by augmenting them. Inside a HOF, an inner function is typically defined which takes the same parameters as the input function. However this inner function can be augmented. An important thing to note is that the inner function can be embedded with variables local to the HOF. These local variables then become embedded with the inner function. Then the inner function may or may not be return (see the HOF definition).

The advantage of using HOFs is that if we want to call $g(f(x))$, instead of explicitly calling $g(f(x))$ every time, we can define $h(x) = g(f(x))$ and call the latter instead.

The best way to demonstrate this is with an example, where the aim is to augment the input function by wrapping its return in the HTML tags.

Listing 1: Wrapping in the return of function `get_text` (src/decorators/bold_text.py).

```
def make_bold(func):
    """HOF that accepts a function `func` and returns a new one.
    The new function calls `func` and modifies its return value.
    That new function is returned."""
    tag_beg, tag_end = '<b>', '</b>'
    def inner(text: str):
        return '{}{}{}'.format(tag_beg, func(text), tag_end)
    return inner

def get_text(text: str):
    """Dummy function that we want to augment, i.e. pass it in a HOF"""
    return text

bold = make_bold(get_text)
print(bold('A quick brown fox'))
print(bold('jumps over the'))
```

The following text gets printed:

```
<i><b>The quick brown fox</b></i>
<i><b>jumps over the</b></i>
```

An important thing to remember is that the arguments passed in the HOF (i.e. `get_text`) must be compatible with the arguments of the inner function (i.e. `inner`). Then supposing we have a new function to augment `join3` which takes 3 inputs, then `make_bold` wouldn't work on it as the latter expects a 2-argument input. Then a new wrapper `make_bold3` would be needed, which operates on a 3-argument function. This would lead to code duplication, as listed below:

Listing 2: Augmenting the functions `join2` and `join3` (src/decorators/bold_text_2_3_args.py) with one wrapper for each.

```
def make_bold2(func):
    """HOF - augments a function of 2 arguments"""
    tag_beg, tag_end = '<b>', '</b>'
    def inner(word1: str, word2: str):
        return '{}{}{}'.format(tag_beg, func(word1, word2), tag_end)
    return inner

def make_bold3(func):
    """HOF - augments a function of 3 arguments"""
    tag_beg, tag_end = '<b>', '</b>'
    def inner(word1: str, word2: str, word3: str):
        return '{}{}{}'.format(tag_beg, func(word1, word2, word3), tag_end)
    return inner

def join2(word1: str, word2: str):
    """Dummy function that we want to augment, i.e. pass in a HOF"""
    return word1 + word2

def join3(word1: str, word2: str, word3: str):
    """Dummy function that we want to augment, i.e. pass in a HOF"""
    return word1 + word2 + word3

bold = make_bold2(join2)
print(bold('A ', 'quick'))
bold = make_bold3(join3)
print(bold('brown ', 'fox ', 'jumps'))
```

It prints:

```
<b>A quick</b>
<b>brown fox jumps</b>
```

A neat way to make HOFs more flexible is by allowing them accept functions that take variable number of arguments. In this case, the augmented function `inner` accepts any number of arguments and unpacks them (`*args` operator) in the "augmentee" function `func`. Therefore if we wanted `make_bold` to operate on functions that take any number of arguments, we could rewrite the previous code as follows:

Listing 3: Augmenting the functions `join2` and `join3` (src/decorators/bold_text_args.py) with one wrapper for both.

```
def make_bold(func):
    """HOF - augments the `inner` function"""
    tag_beg, tag_end = '<b>', '</b>'
    def inner(*args):
        return '{}{}{}'.format(tag_beg, func(*args), tag_end)
    return inner

def join2(word1: str, word2: str):
    """Dummy function that we want to augment, i.e. pass in a HOF"""
    return word1 + word2

def join3(word1: str, word2: str, word3: str):
    """Dummy function that we want to augment, i.e. pass in a HOF"""
    return word1 + word2 + word3

bold = make_bold(join2)
print(bold('A ', 'quick'))
```

```
bold = make_bold(join3)
print(bold('brown ', 'fox ', 'jumps'))
```

That is the idea of decorators in the next section – to create a wrapper function which accepts some function inputs, augment or combine (i.e. "decorate") the inputs without modifying them, and return a new decorated function.

1.3 Decorators

Decorators are a superset (with a grain of salt, I'm not 100% sure if that's the official definition) to HOFs, in the sense that they can also accept a class as input.

> **DEFINITION 1.3** (Decorator). *A decorator is a function that takes another class / function as input or returns a function.*

Python offers syntactic sugar for decorators (therefore HOFs as well). Decorators operate on a the function they aim to wrap, e.g. referring to the last example `make_bold` operates on `join2` and `join3`. However when a decorator is applied, the input function's behaviour changes permanently. The syntax for decorators is as follows:

```
@decorator
def function_to_decorate():
    # do some stuff
```

This is equivalent to:

```
function_to_decorate = decorator(function_to_decorate)
```

Therefore instead of calling `function_to_decorate()`, `decorator(function_to_decorate())` is called. Applying this syntax to Listing 3 with variable number of arguments (`*args`), the code is rewritten as:

Listing 4: Applying the `make_bold` to `join2` and `join3` (src/decorators/dec_bold.py).

```
def make_bold(func):
    """HOF - augments the `inner` function"""
    tag_beg, tag_end = '<b>', '</b>'
    def inner(*args):
        return '{}{}{}'.format(tag_beg, func(*args), tag_end)
    return inner

@make_bold
def join3(word1: str, word2: str, word3: str):
    """Function to decorate"""
    return word1 + word2 + word3

@make_bold
def join2(word1: str, word2: str):
    """Function to decorate"""
    return word1 + word2 + word3

print(join3('A ', 'quick '))
print(join3('brown ', 'fox ', 'jumps'))
```

Furthermore, multiple decorators can be chained on the same function. Referring to Listing 4, if we wanted to make the text bold and then italic, the decorators would be chained as follows:

```
@make_italic
@make_bold
def foo():
    # return some text
```

Therefore they are called from the innermost to the outermost decorator. Hence to decorate the return of `join3` with `make_bold` and then `make_italic` we could chain them as follows:

Listing 5: Applying the `make_bold` to `join2` and `join3` (src/decorators/dec_bold_it.py).

```
def make_bold(func):
```

```
    """HOF - augments the `inner` function"""
    tag_beg, tag_end = '<b>', '</b>'
    def inner(*args):
        return '{}{}{}'.format(tag_beg, func(*args), tag_end)
    return inner

@make_bold
def join3(word1: str, word2: str, word3: str):
    """Function to decorate"""
    return word1 + word2 + word3

print(join3('brown ', 'fox ', 'jumps'))
```

It prints:

```
<i><b>brown fox jumps</b></i>
```

2 Caching

A Appendices

A.1 Appendix Example

Listing 6: A code listing (src/foo_bar.py).

```
class Bar:
    """
    Bar's doc
    """
    def __init__(self, i):
        Bar._i = i

def foo():
    """
    foo's doc
    """
    i = 0
    return "foo"
```