PYTHON NOTES

ON

DECORATORS

BY

0xLeo (github.com/0xleo)

JUNE 10, 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 and closures

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).

There is a definition for enclosed functions combined with local variables of the wrapper function – that is a `closure`. An example of closure will be provided in Listing 1.

> **Definition 1.3** (closure). *The concept of functions enclosed in other functions paired with an environment provided by the outer function is known as closure.*

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. This is done by defining an `inner` function that returns a string, which is formed by referencing local variables of the wrapper. Therefore lines 5-7 define a closure that is returned.

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

```
1  def make_bold(func):
2      """HOF that accepts a function `func` and returns a new one.
3      The new function calls `func` and modifies its return value.
4      That new function is returned."""
5      tag_beg, tag_end = '<b>', '</b>'
6      def inner(text: str):
7          return '{}{}{}'.format(tag_beg, func(text), tag_end)
8      return inner
9
10 def get_text(text: str):
11     """Dummy function that we want to augment, i.e. pass it in a HOF"""
12     return text
13
```

```
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

1.3.1 Basics

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 [1].

> **DEFINITION 1.4** (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, 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:

[1] To be more precise, Python decorators can accept any callable object, i.e. any object that implements the `self.__call__` method. We will focus on functions though.

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

Therefore they are called from bottom to top. 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>
```

1.3.2 Passing parameters to decorators

Because a decorator takes as input the function it decorates, additional arguments cannot be passed directly to it. The trick is to create another HOF that wraps the decorator and takes arguments that the decorator will use. Inside the wrapper, the decorator is defined and returned. Hence, when the decorator is applied (e.g. `@decorator(arg1 = 'val1', arg2 = 'val2')`), what is actually applied to the function is `decorator`'s "inner" function with `arg1` and `arg2` as its internal variables. In the example below the goal is to be able to pass the tag as a decorator parameter, so the user can use the same decorator for any tag.

Listing 6: Modifying the `make_bold` decorator to accept any tag (src/decorators/dec_any_tag.py).

```
def make_tag(tag = 'b'):
    def make_tag_inner(func):
        tag_beg, tag_end = '<{}>'.format(tag), '</{}>'.format(tag)
        def inner(*args):
            return '{}{}{}'.format(tag_beg, func(*args), tag_end)
        return inner
    return make_tag_inner

@make_tag('div')
def join2(word1: str, word2: str):
    """Function to decorate"""
    return word1 + word2

print(join2('brown ', 'fox '))
```

It prints:

```
<div>brown fox </div>
```

1.3.3 Decorator application 1: logging to file

Often when we run unit tests a large number of debugging messages are printed on the screen. Some of them are valuable because they might help trace back a crash. Therefore it may be necessary log them to redirect the standard output from stdout (screen) to a file. In Python this is achieved by adding the following before doing `print`'s:

```
sys.stdout = sys.stderr = open('logfile.txt', 'a') # 'a' to append
```

We want to create a decorator that

- redirects all standard output from `stdout` and `stderr` to a file,
- be able to select the file where the output will be redirected to.

A parameter-based decorator for this task can be applied to function `print_table` as follows:

Listing 7: A decorator which redirects the outputs from `prints` to a file (src/decorators/dec_logger.py).

```
import sys


def log_to_file(fname = '/tmp/log.txt'):
    def log_to_file_inner(func):
        sys.stdout = sys.stderr = open(fname, 'a')
        def inner(*args):
            func(*args)
        return inner
    return log_to_file_inner


@log_to_file(fname = '/tmp/log_test1.txt')
def print_table(text: str):
    n = len(text)
    print('-'*(n+4))
    print('| ' + text +  ' |')
    print('-'*(n+4))

print_table('Test 1 results')
```

Then, the following text would be appended to the output file (in this case `/tmp/log_test1.txt`):

```
------------------
| Test 1 results |
------------------
```

As a final thought, to make a more advanced logger more information could be appended to the log file via `log_to_file`, such as the name of the called function (`func.__name__`), when it was called, and when it exited. `datetime` module provides this functionality. However, making such a logger is not in the score of this article.

1.3.4 Decorator application 2: memoisation

Complicated functions, such as recursive ones, prime number finders, etc. often take longer and longer time the larger the input is. Specifically for recursive functions, the current input may depend on previous, e.g. the Fibonacci numbers. The idea of memoisation is to cache previously computer returns in a $\{x, f(x)\}$ dictionary so that when $f(x)$ is about to be called, if x_i is in the dictionary, then return the pre-computed $f(x_i)$. If not, then add $\{x_i, f(x_i)\}$ in the cache. Decorators make it elegant to implement this.

Before implementing a memoization, it's important to remember that closures encapsulate not just the returned function but also the local variables in the wrapper. Therefore the wrapper local variables don't die after the closure is called, but they keep living inside the wrapper (decorator).

Using these specifications, the `memoize` decorator looks as follows:

Listing 8: A "memoiser" which caches the returns of the function it wraps (src/decorators/dec_memoisation.py).

```
def memoize(func):
    cache = {}
    def inner(*args):
        if args not in cache:
            # register it in cache
            cache[args] = func(*args)
        return cache[args]
    return inner
```

To apply the decorator, a function is created which simulates some heavy work by sleeping for $0.1 \times n$ seconds, where n is its input. Adding this to the code in Listing 8, we end up with the following complete program:

Listing 9: A "memoiser" applied to a function that takes a long time to process its input (src/decorators/dec_memoisation.py).

```
import time

def memoize(func):
    cache = {}
    def inner(*args):
        if args not in cache:
            # register it in cache
            cache[args] = func(*args)
        return cache[args]
    return inner

@memoize
def long_function(n):
    dur = n*0.1
    time.sleep(dur)
    return dur

if __name__ == '__main__':
    before = time.time()
    for _ in range(1000):
        long_function(5)
    print("Took {:.4f} sec".format(time.time() - before))
```

We can confirm that the memoiser does its job by examining that running `long_function` 1000 times for the same input of $n = 5$ takes around 0.5 seconds, which is roughly the time for one run. The program prints:

```
Took 0.5008 sec
```