Closure in Python

First Lesson. (Most Important)

Nonlocal variable in a nested function

- Before getting into what a closure is, we have to first understand what a nested function and nonlocal variable is.
- A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope.
- In Python, these non-local variables are read only by default and we must declare them explicitly as non-local (using nonlocal keyword) in order to modify them.

Example

```
def print_msg(msg):

    def printer():
        print(msg)
    printer()

print_msg("Hello")
```

Defining a Closure Function

```
def print_msg(msg):
    def printer():
        print(msg)
    return printer  # this got changed
another = print_msg("Hello")
another()
```

Explanation

- The print_msg() function was called with the string "Hello" and the returned function was bound to the name another. On calling another(), the message was still remembered although we had already finished executing the print_msg() function.
- This technique by which some data ("Hello") gets attached to the code is called closure in Python.
- This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

Example

```
>>> del print_msg
>>> another()
Hello
>>> print_msg("Hello")
Traceback (most recent call last):
...
NameError: name 'print_msg' is not defined
```

When do we have a closure?

The criteria that must be met to create closure in Python are summarized in the following points.

- We must have a nested function (function inside a function).
- The nested function must refer to a value defined in the enclosing function.
- The enclosing function must return the nested function.

Short Class

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier

times3 = make_multiplier_of(3)

times5 = make_multiplier_of(5)

print(times3(9))

print(times5(3))

print(times5(times3(2)))
```

Make a nested loop and a python closure to make functions to get multiple multiplication functions using closures.

Assignment

Make a nested loop and a python closure to make functions to get multiple addition functions using closures.

Assignment

Make a nested loop and a python closure to make functions to get Non Divisible by 0 number using closures.

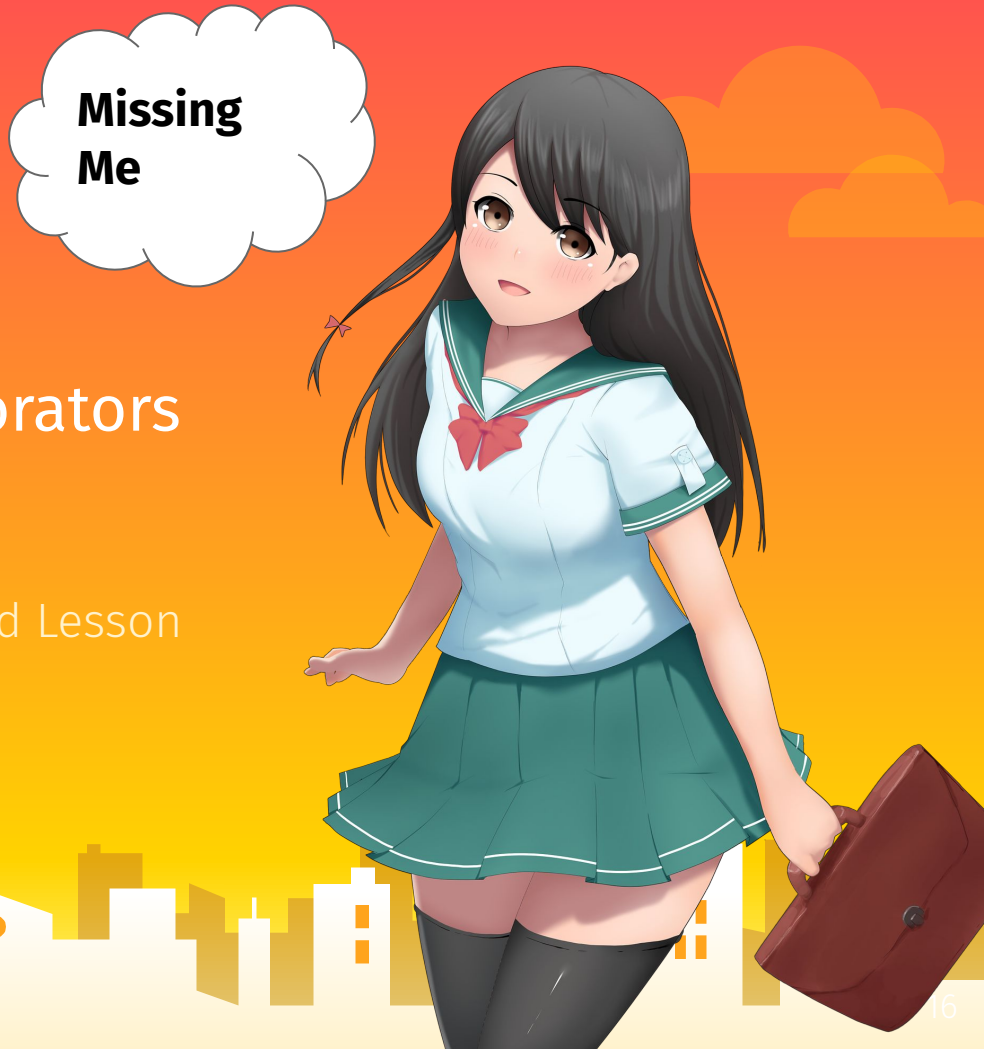
Assignment

Make a nested loop and a python closure to make functions to get multiple substration using closures.

Assignment

Make a nested loop and a python closure to make functions to get multiple division using closures.

Assignment



Decorators

Second Lesson

What are decorators in Python?

- Python has an interesting feature called decorators to add functionality to an existing code.
- This is also called metaprogramming as a part of the program tries to modify another part of the program at compile time.

higher order functions.

- Functions can be passed as arguments to another function.
- If you have used functions like map and filter in Python, then you already know about this.
- Such function that take other functions as arguments are also called higher order functions. Here is an example of such a function.

Example

```
def inc(x):
    return x + 1

def dec(x):
    return x - 1

def operate(func, x):
    result = func(x)
    return result
```

Execution

```
>>> operate(inc,3)
4
>>> operate(dec,3)
2
```

function can return another function

```
def is_called():
    def is_returned():
        print("Hello")
    return is_returned

new = is_called()
new()
```

Getting back to Decorators

- Functions and methods are called callable as they can be called.
- In fact, any object which implements the special method __call__() is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.
- Basically, a decorator takes in a function, adds some functionality and returns it.

Example

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner

def ordinary():
    print("I am ordinary")
```

Execution

```
>>> ordinary()
I am ordinary

>>> # let's decorate this ordinary function
>>> pretty = make_pretty(ordinary)
>>> pretty()
I got decorated
I am ordinary
```

@ sign for decorators

We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

```
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

Decorating Functions with Parameters

```
def smart_divide(func):
    def inner(a,b):
        print("I am going to divide",a,"and",b)
        if b == 0:
            print("Whoops! cannot divide")
            return
        return func(a,b)
    return inner
@smart_divide
def divide(a,b):
    return a/b
```

Write a Python program to make a chain of function decorators (bold, italic, underline etc.).

Assignment

Make a decorator factory which returns a decorator that decorates functions with one argument.

Assignment

Thank you

Miss You...