*ACT I - DECORATOR*

**Teacher:** Suppose I have a function:

def foo( ):

print(“Hello World”)

This function is rather bland by itself. I would like to make sure that whenever this function is called, we get a message “Greetings” before it prints “Hello World”, and a message “Goodbye” after it. How can we go about doing this?

**C:** Well, that wouldn't be any less bland than the original function!

**Teacher:** Come on, what else would you want?

**B:** Our friend wants to decorate it and probably wrap it in garlands and what not. That alone will satisfy him.

**Teacher:** Well, you don't realize a hidden meaning in your statement. Anyway, alright. Let's make it more interesting then. Let's say after the function prints “Hello World”, it should print out its own name.

**D:** Wow! Can a function know its own name?

**A:** Of course not! That's a trick question the teacher is asking. But it's straightforward. All we need to do is write: print(“I am foo”) after the first print statement. And print(“Greetings”) before the first print statement.

**C:** My goodness. You are so good at decorating stuff, aren't you?

**A:** I am the person who decorates the college during all grand events. So naturally, when I can decorate a college, what is a mere function? That can as well be easily done.

**Teacher:** Sadly, both I and the Python world would highly recommend you to stick to decorating the college if this is your approach!

**A:** Huh? Isn't my solution elegant?

**B:** Yeah, it definitely is! You yourself said the function is supposed to print something before and after. And this does the job pretty neatly.

**Teacher:** Ok, suppose I also wanted the function to print out “Goodbye” after printing out “I am foo”.

**A:** I don't see what you are trying to get at. It is so blatantly obvious that a half-blind bat could tell you the solution!

**D:** Hey, aren't bats blind totally by default? So this bat would have to be far more intelligent than the others if it was half-blind.

**A:** You and your lame jokes. Anyway, just add print(“Goodbye”) after print(“I am foo”)

**Teacher:** The problem with this approach is that you are modifying the function body itself! And a trivial point – you are printing “I am foo”, when you are not foo, but you are A!

**C:** Oh God! I thought only A made lame jokes. Not even the teacher now!

**Teacher:** But I am serious. We want the function to tell us who it is.

**D:** Surely, there must be some special function which when called will give us the name of the function.

**B:** That must indeed be such a stupid thing. I can't imagine writing getName(foo) only so as to get back foo, when all along we might as well have written foo, without all the fancy stuff. What kind of drama is this?

**C:** Our lives are embroiled in drama. All our life is a stage, and all men and women merely players, as Shakespeare said.

**Teacher:** We seem to be digressing towards the trivia. Sigh. I had something else in mind, and we are getting sidetracked looking at the minor issue of printing a function's name. To answer B, when you say getName(foo), you won't get back foo, but you would get back 'foo'!

**B:** What on earth do you mean? You are contradicting yourself! You said you won't get back foo, but we will get back foo. Is this some sort of “paradocs”?

**Teacher:** Oops, I forgot that I there is no quotation mark in speech!

**C:** Our teacher has gone crazy.

**Teacher:** Well, what I meant was that, when you call such a function you might feel what is the need of it, but then it does not just return back foo, but it returns the *string* foo!

**D:** Ah! Now I get it. So we call print(“I am ” + getName(foo)) after the other print.

**Teacher:** The only problem is that there is no such thing as getName!

**A:** Aaaah! You led us down the garden path! You should have told us before.

**Teacher:** The way to get a function name is rather interesting. There is an attribute called \_\_name\_\_ associated with every function. Given any function x, x.\_\_name\_\_ will give us a string containing the name of the function.

**C:** How elegant! We will use that then.

**Teacher:** We still haven't handled the main problem! We are changing the function manually!

**B:** The function is there, isn't it?

**Teacher:** In a general case, we may not even have access to the function's implementation. For example, look at random.random function. Just imagine if we had access to its internal mechanisms. Any one could change it willy nilly, and the random function would behave truly randomly!!! :P

**C:** Oh god! That was a calculated joke wasn't it?

**Teacher:** Well come on! How do you know that random function wasn't written something like: return getpid( ) ? That would appear to be random enough! That reminds me of a joke I had once heard. There was a random number generator that went on printing 9, 9, 9 … Someone asked if it really was a random number generator. To which he was answered, “Yes of course! You never know what it is going to generate next!”

**B:** I think I get the point. We only have the interface to some function, and the teacher wants the function to print something else before and after without us tinkering with it.

**Teacher:** Finally at long last! We have arrived at an understanding of the problem.

**A:** Ok. Then can't we just say:

print(“Greetings”)

foo()

print(“I am ” + foo.\_\_name\_\_)

**Teacher:** But there is a problem with that approach. Each time I want to call foo, I want this behaviour. Which means that each time I want to call foo, I need to remember to put one print statement before calling the function, and one print statement after calling it. That's rather cumbersome, isn't it?

**D:** Oh I know! Why don't we pack all of these into some other function, say bar. Then the user calls bar whenever he wants this functionality. This way, bar will execute the print before, then the function and then the print after.

**Teacher:** The user wants to call foo. That name foo is the only interface to him. Looking at the name foo will tell him what this function is doing (Ok, I must admit, looking at the name foo won’t tell him anything at all, but let’s assume it does). It is terrible to change the interface so that the user has to call some other function each time he wants to call what he originally wanted to! I mean, come on, suppose I told you to call function SQUARE\_ROOT whenever you wanted to call math.sqrt. You would do it in some places, and you might forget to do it in other places. It is so inconvenient for you to have to remember each time consciously that you have this new function name that you better be calling each time you intended to call the original!

**C:** Well then why can't we say foo = bar; We know that a function name acts like an expression, and so after this statement whenever we call foo, since both foo and bar refer to the same thing, we get what we want. Thus we are saved from having to remember to call bar each time.

**Teacher:** We are making some progress. There is a problem with this however. When we call foo after your suggested change, this will go and call bar instead. Then inside bar, after printing out “Greetings”, it will make a call to foo. But by this time, foo is already changed to refer to bar itself. Which means this will end up calling bar only once again. Thus, it will print out “Greetings” once again, and once again attempt to call foo, but hopelessly it is stuck in a quagmire, since foo and bar are one and the same at this point!

**B:** Oh god! This is such a paradoxical state.

**A:** Yeah, it sort of reminds me of these Harry Potter or other science fiction stuff!

**Teacher:** Huh? How so?

**A:** Because the first time we call foo, it should go and actually call bar. But within bar, when we call foo, the function foo should go back in time, and change itself to refer to what it was originally referring to! This looks like witchcraft!

**D:** Ooooh! I think I see a light in the dark. Why don't we make sure that we have some other variable called temp that originally refers to foo. This way, both foo and temp refer to that original block of code that foo is referring to. So that later when we say foo = bar, the original block of code does not become garbage since at least some other variable temp is referring to it. Then within bar, just before calling foo, we can say foo = temp. This way, just before foo gets called within bar, it changes to refer to what it was originally referring to. In this way, we won't have any infinite recursion!

def foo():

print(“Hello”)

temp = foo

def bar():

print(“Greetings”)

foo = temp

foo()

print(“I am ”, foo.\_\_name\_\_)

foo = bar

foo()

**A:** Wow! This looks so elegant!

**Teacher:** Very good. However, I wanted to lead us to another solution, and we have landed up with this one. Not bad.

**C:** Do you mean to say there is another way to achieve this?

**Teacher:** Yes indeed! Let us try to lead ourselves to such a solution. In this example that we have taken, it was rather simple. Now, suppose I have a function foo which goes like this:

def foo(x):

print(“<h1>” + x + “</h1>”)

This is some sort of HTML kind of snippet. We want this function to execute the print statement only if x is a string. How can we do it?

**D:** Wait a minute! I thought Python was supposed to be a “dynamically” typed language.

**B:** Yes, but that does not mean that variables have no types.

**Teacher:** Yeah, it means that the type of a variable is determined at run time.

**A:** Ok, so that means that a variable does have a type associated. Oh yeah, of course. We should all remember that we have been using type(something) so often. Surely this means that each variable is associated with a type.

**B:** Ok, so basically we need to say something like: if type(x) == 'str': then execute the print statement.

**C:** But wait. If you remember, whenever we ran the program with a type function, it displayed in a rather fancy way, with angle brackets, and the word class and all that. So we cannot just compare type(x) with 'str'; we need to compare with “<class 'str'>”

**D:** Yeah, that makes sense.

**Teacher:** That is too cumbersome. Besides, there is a nice little function to help us with this. It is called isinstance. It takes two parameters, the first being some value or variable, and the second being the name of a type. It returns True or False depending on whether the value/variable is of the type specified by the second parameter. So, isinstance(10, int) will return True; Notice that the int is not specified in quotes – it is not a string. We specify the type name as is.

**C:** Ok, that considerably saves effort. So we need to say, if isinstance(x, str) : then execute the print statement.

**Teacher:** But where do we put this if statement?

**A:** Just before the print statement of course.

**D:** Oh no! Haven't we been through this argument before? We are not supposed to touch the function.

**B:** Oh yeah. So we can have a function bar that checks the if statement, and if x is of type str, then it calls foo.

def bar(x):

if isinstance(x, str):

foo(x)

**Teacher:** Now we again face the same problem as before. We don't want to change the interface to the user. He should not be forced to call bar, when what he wants is to call foo. So like before, you would suggest that we can assign foo = bar;

**C:** Yeah. But now we have the problem as mentioned before; when we call foo, it is actually calling bar, which will cause a problem of infinite recursion since it calls foo, which is actually not foo, but has changed to bar.

**D:** Yeah, so we will use the trick we used before. Store foo in temp. Then change foo to temp inside bar just before we call foo.

def foo(x):

print(“<h1>” + x + “</h1>”)

temp = foo

def bar(x):

if isinstance(x, str):

foo = temp

foo(x)

foo = bar

foo(“Hello”)

foo(10)

**C:** This works!

**Teacher:** Ok. Now, suppose we have one more function say foo2, like this:

def foo2(x):

print(“<p>” + x + “</p>”)

Now we again wish to do the same thing like before. That is, if x is a string, only then should we execute the print statement. So how would you proceed?

**A:** This is becoming boring by now. It's the same trick.

temp = foo2

foo2 = bar

**B:** Yeah, as a result of the above two statements, foo2 will refer to the modified function. So when the user calls foo2, it internally checks whether the parameter passed is a string, and if so, it calls foo, which by this time has been changed to refer to temp, i.e the original foo2.

**Teacher:** Now, with all the above done as you have told, tell me what happens here:

foo2(“Hello”)

**D:** It prints <p>Hello</p>

**Teacher:** foo2(10)

**C:** Nothing is printed.

**Teacher:** foo(“Hello”)

**A:** It prints <h1>Hello</h1>

**Teacher:** Whoa!!!! Hold it right there!!!! Can we please trace this particular line in detail?

**B:** Oh my god! How many times do we have to play this charade? We have done it so many times by now.

**Teacher:** Trace the execution of only this line: foo(“Hello”); given the following code that we have elaborately written so far:

def foo(x):

print(“<h1>” + x + “</h1>”)

temp = foo

def bar(x):

if isinstance(x, str)

foo = temp

foo(x)

foo = bar

def foo2(x):

print(“<p>” + x + “</p>”)

temp = foo2

foo2 = bar

foo(“Hello”)

**C:** Ok. When we call foo, since we have earlier said foo = bar, this will call bar. Now, inside bar, it checks if “Hello” is an instance of str, which it is, so it assigns foo to temp...

**Teacher:** Ok, pause a moment. Look at the code. Tell me what is temp at this point?

**A:** Oh my God! Just two lines before calling foo, we have said temp = foo2. So when we are inside bar, temp is referring to foo2, which gets assigned to foo, and so when we call foo(x) inside bar, we are actually ending up calling foo2(x), when we wanted to call foo(x). So it will print: <p>Hello</p>

**Teacher:** Now do you all see the gaping hole in your approach?

**C:** Goodness! All these that we have elaborately built has gone down the drain.

**Teacher:** Don't be disheartened. You people were on the right track. Why don't we perform a post mortem on this to figure out why it went wrong?

**B:** Sounds like a good idea except that I don't like post mortems.

**D:** Oh come on! I would rather look at the insides of code than a human.

**C:** Funnily enough, we are performing a post mortem on Python!

**A:** Ok guys. Enough post mortem on the word post mortem.

**D:** Haha. Ok. Well, the reason why the above code didn't work was because when bar was called temp was referring to foo2, when it should have been referring to foo.

**B:** Ok. So what we want is that inside bar we should be able to call foo, and this should work regardless of how we call bar, whether through foo1, or foo2 or foo3, or anything else.

**Teacher:** In other words, bar should somehow know which function is calling it. If foo2 is using bar, then inside bar, when we call foo, this should in turn call foo2. If foo3 is using bar, then when we call foo inside bar, this should in turn call foo3's code. To achieve this, do you remember the idea of closure that we discussed about?

**C:** Yeah. When we have a nested function, that nested function will remember the variables of the enclosing function.

**Teacher:** Yeah, that is a crude way to put it. Can we use this idea of closure to our aid here? Let us look at this code:

def decorate(fn):

def bar(x):

if isinstance(x, str):

fn(x)

return bar

**A:** My head is spinning!

**D:** How are we supposed to use this to achieve what we wanted to?

**Teacher:** Let's think of this. Decorate is a function that takes a function as its parameter. It creates a nested function which makes use of this fn parameter. Now, remember the idea of closure. Once we are outside of decorate function, which has happily returned bar, if we call this bar through that outside reference, then bar will get executed, and inside bar, fn will get executed. This fn is “remembered” to be what was passed as the argument to decorate. So if fn happened to be foo, then bar will call foo. If fn happened to be foo3 when we called decorate, then bar will end up calling foo3. In other words, decorate(foo2)(“Hello”) will call foo2, before which it checks that the parameter “Hello” is a string. decorate(foo2) is a function call which returns a callable bar. This in turn can be called, and hence cascaded as shown above.

**C:** Are you trying to say that each time the user wants to call foo2, he needs to have this elaborate double call? You had told that the user should only use the interface given, which is foo2, so why this complexity?

**Teacher:** This leads us to the elegance:

foo2 = decorate(foo2)

foo2(“Hello”)

**D:** That makes sense. Just before the user calls foo2 the way he normally does, he calls this other function and assigns it back to foo2. What this function returns is our bar and hence foo2 now refers to bar. However, this bar is not just any bar! It is a clever nested bar, which means that when we call foo2, we in turn call the nested bar, which by virtue of being nested, when calling fn(x), knows that fn refers to foo2 because of closure!

**Teacher:** Yes. Now, we can even say foo = decorate(foo), and use it. This will not cause the problem we had earlier when temp had got corrupted to refer to foo2 when it should have been referring to foo. This won't happen in this case, because each nested bar that is returned remembers the parameter fn name that decorate was called with.

**A:** Wow! This is brilliant.

**Teacher:** This complex stuff can also be written this way:

def decorate(fn):

def inner(x):

if isinstance(x, str):

fn(x)

return inner

@decorate

def foo(x):

print(“<h1>” + x + “</h1>”)

What the above notation means is that whenever we have something like: @alpha followed by a def beta(x), this does: beta = alpha(beta) automatically without us having to explicitly do so. So in the above, it does, foo = decorate(foo).

And this, my dear students is the concept of decorator.