**Closures and Decorators in Python**

You will finally understand what closures and decorators are



Closures are an important tool in functional programming, and some important concepts like *currying* and *partial application* can be implemented using them. Decorators are also a powerful tool in Python which are implemented using closures and allow the programmers to modify the behavior of a function without permanently modifying it. In this article, I will first explain the closures and some of their applications and then introduce the decorators.

**Scope of variables**

To better understand the closures first we need to learn about the scope of variables in Python. The *scope* of a variable refers to the area in which you can see or access a variable. A variable cannot be accessed outside its scope. The scope of a variable is determined by where it is assigned in the source code. Generally, variables can be assigned in three different places, corresponding to three different scopes:

*Global scope*: When a variable is defined outside all functions. A global variable can be accessed by all the functions in the file.

*Local scope*: When a variable is defined inside a function, it is local to that function. A local variable can be only accessed inside the function in which it was defined.

*Nonlocal scope*: When a variable is assigned in an enclosing function, it is nonlocal to its nested functions. A nonlocal variable can be accessed by the function in which it was defined and all its nested functions.

If you reassign a global variable inside a function, a new local variable with the same name will be created, and the global variable is said to be *shadowed* by this local variable. Any changes to this local variable inside the function, will not affect the global variable. So if you want to change a global value inside a function, you have to use the global keyword in Python for this. For example in Listing 1, initially, we define two global variables x and y. When we reassign x inside the function f(), a new local variable with the same name is defined, and it does not affect the global variable x. However, by using the global keyword we can access the global variable y inside f(). z is a local variable of f() and cannot be accessed outside it. After exiting f(), this variable is not present in memory, so it cannot be accessed anymore.

# Listing 1

x = 1 # x is a global variable

y = 5 # y is a global variable

def f():

    global y

    x = 2   # x is a local variable

    y += 1  # Reassigning the global variable y

    z = 10   # z is a local variable

    print("Local variable x =", x)

    print("Global variable y =", y)

    print("Local variable z =", z)

f()

print("Global variable  x =", x)

print("Global variable y =", y)

Output:

Local variable x = 2

Global variable y = 6

Local variable z = 10

Global variable  x = 1

Global variable y = 6

In Python, everything is an object, and the variables are references to these objects. When you pass a variable to a function, Python passes a copy of the reference to the object to which the variable refers. It does not send the object or the original reference to the function. So both the original reference and the copied reference that the function receives as its argument are referring to the same object. Now if we pass a global object that is immutable (like an integer or a string), the function cannot modify it using its argument. However, if the object is mutable (like a list), the function can modify it. Here is an example:

Listing 2

a = [1, 2, 3]

b = 5

def func(*x*, *y*):

*x*.append(4)

*y* = *y* + 1

func(a, b)

print("a=", a)  #  Output is a=[1, 2, 3, 4]

print("b=", b)  #  Output is b=5

We re passing two variables a and b to func. a is a reference to a list which is a mutable object, and b is a reference to an integer which is immutable. func receives a copy of a as x and a copy of b as y. Appending a new element to x will change the original object that a refers to. But adding 1 to y inside func does not affect the object that b refers to. It only creates a new integer object that contains 6 and y will refer to it now (Figure 1).

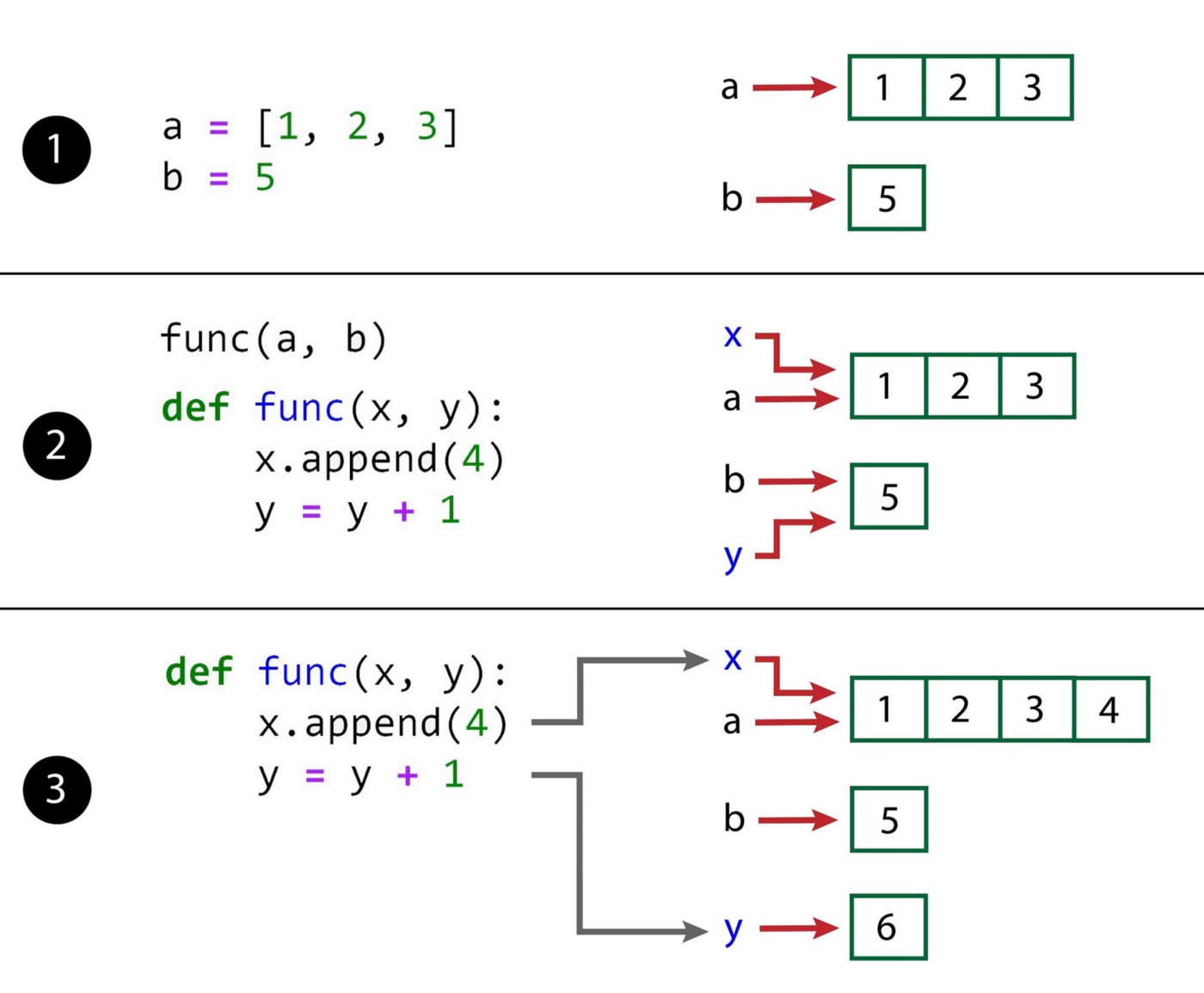


Figure 1

**Inner functions**

An *inner function* (or *nested function*) is a function that is defined inside another function (the outer function). The local variables of the outer function are said to be nonlocal to its inner function. The inner function can access the nonlocal variables but cannot change them. Reassigning them simply creates a new local variable with the same name in the inner function, and does not affect the nonlocal variable. So if you want to make a change to a nonlocal variable in a nested function, you must use the nonlocal keyword.

In this article, we may simply call a variable of the outer function a nonlocal variable, but it means that it is nonlocal with respect to an inner function. If you remove the inner functions, it will be only a local variable of the outer function. In addition, the inner function should somehow access this variable or declare it as nonlocal to call it a nonlocal variable. If the inner function does not touch a local variable of the outer function, it will not be a nonlocal variable to that inner function.

In this article, we never try to change a function parameter. So it suffices to know that the outer function parameters behave like nonlocal variables and can be also accessed (or read) by the inner function.

In Listing 3, the function parameter x, and the variables y, z and t are the local variables of f().

# Listing 3

def f(*x*):

    y = 5

    z = 10

    t = 10

    def g():

        nonlocal y

        y += 1

        z = 20

        print("Nonlocal variable x =", x)

        print("Local variable z =", z)

    print("Local variable t =", t)

    g()

    print("Nonlocal variable x =", x)

    print("Nonlocal variable y =", y)

    print("Local variable z =", z)

f(5)

# This does not work:

# g()

Output:

Local variable t = 10

Nonlocal variable x = 5

Local variable z = 20

Nonlocal variable x = 5

Nonlocal variable y = 6

Local variable z = 10

x, and y are also the nonlocal variables of its inner function g(). They are accessed inside g(). When you reassign z, a new local variable with the same name will be created inside g() which does not affect the local variable z in f(), so z is not a nonlocal variable (to g). However, by using the nonlocal keyword you can change y and it is still a nonlocal variable. The variable t is not accessed by g at all, so it is not a nonlocal variable.

The variables y and z belong to the local scope of f() and cannot be accessed outside f(). The inner function g() is also defined inside f() and can be only called there. In fact, If you try to run Listing 3 and call g() outside f(), Python gives you an error.

But what if we could find a way to call it outside g(). Then we would have a second problem. After exiting the outer function, its local variables (which are nonlocal to g()) are not present in the memory anymore. In that case, the inner function can not access them anymore. Closures make it possible to call an inner function outside the outer function and still access its nonlocal variables.

**Closures**

First of all, we need to find a way to call an inner function outside the outer function. Remember that a function can always return a value. In Listing 4, x and y are local variables of f():

# Listing 4

def f():

    x = 5

    y = 10

    return x

h=f()

So after running h=f(), all the local variables of f() are gone, and you cannot access x and y anymore. But we still have the value of x that it is returned and stored in h. So maybe we can make the outer function to return the inner function. This is possible in Python since Python functions are *first class*. It means that Python treats functions as values, so you can assign a function to a variable, pass it as a function argument or return it by another function. In Listing 5, the outer function f(x) return its inner function g.

# Listing 5

def f(*x*):

    def g(*y*):

        return y

    return g

a = 5

b = 1

h=f(a)

h(b)  # Output is 1

Now f(x) returns the function g. So when we write h=f(a), we are assigning g to h and now h can be treated like g and accept the g’s argument. As a result h(b) is like calling g(b).

The \_\_name\_\_ attribute of a function stores the name with which the function was defined. If we write:

h.\_\_name\_\_

it returns

'g'

Which means that h refers to function g now.

You should note that what f(x) is returning is the function g not a specific value of that. For example. If we write something like:

# Listing 6

def f(*x*):

    def g(*y*):

        return *y*

    return g(y)

a = 5

b = 1

h=f(a)

# This does not work:

# h(b)

then we get an error. The reason is that we are returning the value of g(y) by f(x) (which is 1), so h=1, and we cannot use it to call a function.

We do not need to store f(a) in h. Instead, we can directly call f(a)(b).

# Listing 7

def f(*x*):

    def g(*y*):

        return *y*

    return g

a = 5

b = 1

f(a)(b)  # Output is 1

It is important to note that we should distinguish between f(a, b) and f(a)(b). The function f(a, b) takes two arguments. However, f(a)(b) is a sequence of nested functions that each takes one argument. A function that only takes one argument is called a *unary* function. so f(a)(b) is a sequence of two nested unary functions. As Figure 2 shows Python evaluates these nested functions from left to right. So first f(a) is evaluated. To be able to chain these functions, f(a) should return another function. Here it returns the inner function g. So f(a)(b)=g(b)

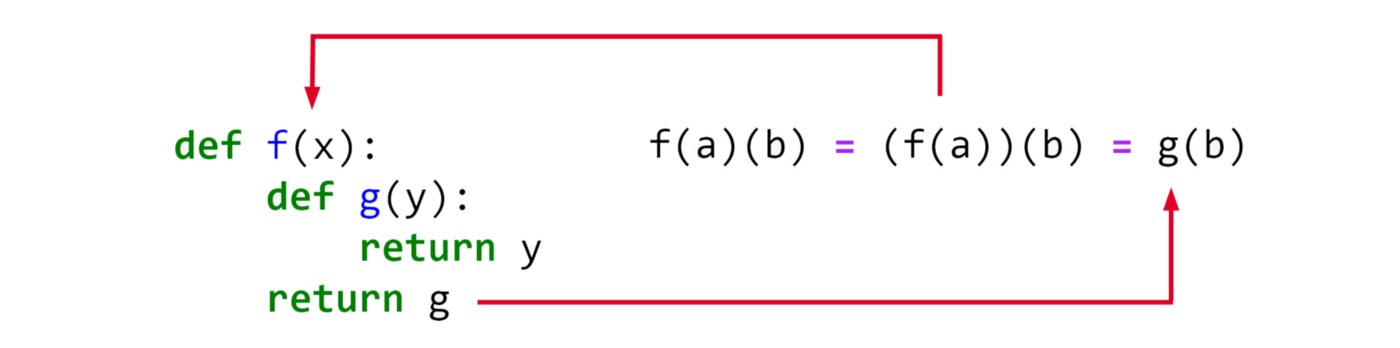


Figure 2

We can easily extend this method to have a longer sequence of unary functions. Listing 8 shows how we can have three nested unary functions. The first function f(x) has an inner function g(y) and g(y) has an inner function h(z). Each outer function returns its inner function.

# Listing 8

def f(*x*):

    def g(*y*):

        def h(*z*):

            return *z*

        return h

    return g

a = 5

b = 2

c = 1

f(a)(b)(c)  # Output is 1

but what happens if the outer function has some nonlocal variables? Here is an example:

# Listing 9

def f(*x*):

    z = 2

    def g(*y*):

        return z\**x* + *y*

    return g

a = 5

b = 1

h = f(a)

h(b)  # Output is 11

If we run it, we can see that it works without error and g(y) can access both the variables x and z. But how is that possible? After running f(x), we are not in the scope of f(x) anymore and the variables x and z should not be accessible. Why g(y) can still access them? That is because the inner function g(y) is now a *closure*.

A closure is an inner function with an extended scope that encompasses nonlocal variables of the outer function. So it remembers the nonlocal variables in the enclosing scopes even if they are not present in memory. A variable like y which is in the local scope of the inner function g(y) is called a *bound variable*. A function that only has bound variables is called a *close term*. On the other hand, a nonlocal variable like z is called a *free variable*since it is free to be defined outside g(y), and a function that contains free variables is called an *open term*.



Figure 3

The name “closure” comes from the fact that it captures the bindings of its free (nonlocal) variables and is the result of *closing* an open term.

So a closure is an open term which is closed by capturing the bindings of its free (nonlocal) variables. In Listing 9, the inner function g(y) is not a closure as long as it has a free variable which is not bound yet (x and z). Once we evaluate h=f(a), the enclosing function f(x) is evaluated, and the free variables x and z become bound to 5 and 2 respectively. So g(y) returned by f(a) becomes a closure, and h is now referring to a closure (Figure 4).

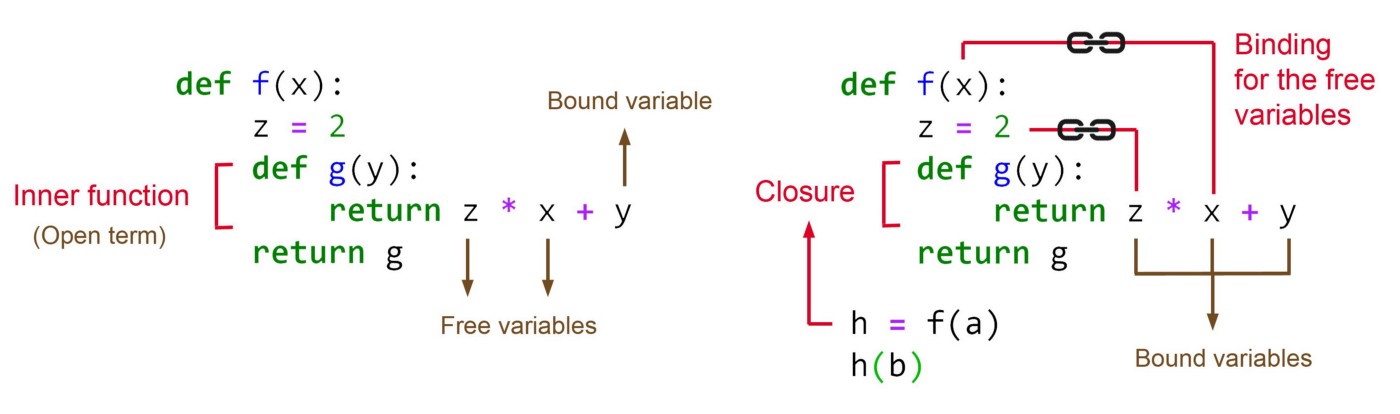


Figure 4

Python can keep track of the free variables for each function. We can simply use \_\_code\_\_.co\_freevars attribute to see which free variables are captured by the inner function. For example for the closure defined in Listing 9 we can write:

h.\_\_code\_\_.co\_freevars

and the output will be:

('x', 'z')

You can also get the value of these free variables using the closure attribute:

print(h.\_\_code\_\_.co\_freevars[0], "=",

      h.\_\_closure\_\_[0].cell\_contents)

print(h.\_\_code\_\_.co\_freevars[1], "=",

      h.\_\_closure\_\_[1].cell\_contents)

the output will be

x = 5

z = 2

It is important to note that to have a closure, the inner function should *access* the nonlocal variables of the outer function. When no free variable is accessed inside the inner function, it does not capture them since it is already a closed term and does not need to be closed. For example, if we change the inner function in Listing 9 to:

# Listing 10

def f(*x*):

    z = 2

    def g(*y*):

        return *y*

    return g

a = 5

b = 1

h = f(a)

h(b)  # Output is 1

Then it cannot be considered a closure anymore. The reason is that the nonlocal variables x and z are not accessed inside g(y) and there is no need for g(y) to capture them. We can easily check it by writing:

h.\_\_code\_\_.co\_freevars

and the output will be:

()

In addition, h.\_\_closure returns None which means h is not a closure anymore. If you do not access a nonlocal variable but define it as nonlocal inside the inner function, it is still captured by the closure. So in Listing 11 g(y) is a closure since it captures t.

# Listing 11

def f(*x*):

    z = 2

    t = 3

    def g(*y*):

        nonlocal t

        return *y*

    return g

a = 5

b = 1

h = f(a)

h(b)

h.\_\_code\_\_.co\_freevars  # Output is ('t',)

Another example which is a little trickier:

# Listing 12

def f(*x*):

    def g(*y* = *x*):

        return *y*

    return g

a = 5

b = 1

h = f(a)

h()  # Output is 5

Again g(y) is not a closure here since the value of x is just used to initialize y and g does not need to capture x.

When you have multiple nested functions, each closure is able to capture all the nonlocal variables which are at higher levels. For example, if we have three nested functions:

# Listing 13

def f(*x*):

    def g(*y*):

        def h(*z*):

            return *x* \* *y* \* *z*

        return h

    return g

a = 5

b = 2

c = 1

f(a)(b)(c)  # Output is 10

Then h(z) is accessing the nonlocal variables of f and g, so it will capture both of them, so the output of

# f(a)(b) refers to h

f(a)(b).\_\_code\_\_.co\_freevars

will be:

('x', 'y')

In addition, g(y) is also a closure and it captures x as a nonlocal variable. We can easily check it (remember that f(a) refers to g(y)):

f(a).\_\_code\_\_.co\_freevars  # Output is ('x',)

But g(y) does not access x. Why g(y) should become a closure and capture x? The reason is that its inner function is accessing it. h(z) needs to capture x, but a closure can only capture the free variables of its outer function which is g(y) in this example. So first g(y) should extend its scope and capture x, and then h(z) can extend its scope and capture x as a free variable of g(y). If we replace Listing 13 with:

# Listing 14

def f(*x*):

    def g(*y*):

        def h(*z*):

            return *y* \* *z*

        return h

    return g

a = 5

b = 2

c = 1

f(a).\_\_code\_\_.co\_freevars  # Output is ()

We can see that g(y) is not a closure anymore. That is because h(z) does not need to capture x. As a result, g(y) doesn't capture it either, and doesn't become a closure.

As you may have noticed, to call a closure, we do not use the name of the inner function. We only use the name of the outer function. So we can also replace the inner function with a lambda function if we only have one expression. For example, we can replace the inner function g in Listing 10 with an anonymous function:

# Listing 15def f(x):  
 z = 2  
 return lambda y: z\*x+y  
a = 5  
b = 1  
f(a)(b) # Output is 11

At this point, I want to summarize what we learned about closures so far. To define a closure we need an inner function that:

1-It should be returned by the outer function.

2- it should capture some of the nonlocal variables of the outer function. This can be done by accessing those variables, or defining them as a nonlocal variable or having a nested closure that needs to capture them.

After defining the closure, to initialize it, you have to call the outer function to return the closure.

In functional programming, closures make it possible to bind data to a function without actually passing them as parameters. This is similar to what a class does in object-oriented programming. In Listing 16, we compare these paradigms. We first create a class to calculate the nth root of a number.

# Listing 16class NthRoot:  
 def \_\_init\_\_(self, n=2):  
 self.n = n  
 def set\_root(n):  
 self.n = n  
 def calc(self, x):  
 return x \*\* (1/self.n)  
   
thirdRoot = NthRoot(3)  
print(thirdRoot.calc(27)) # Output is 3def nth\_root(n=2):  
 def calc(x):  
 return x \*\* (1/n)  
 return calcthird\_root = nth\_root(3)  
print(third\_root(27)) # Output is 3

As you see the outer function can play the role of a constructor for us here. It initializes the nonlocal variables that will be used by the inner function. However, there are also some differences. The NthRoot class can have more methods that can be invoked by the object thirdRoot . However, what is returned by nth\_root is a function itself. So this method is more limited compared to what classes can do. Now that we are familiar with closure, we can see some of their applications.

**Composition**

If we have two functions *f* and *g*, we can combine them in such a way so that the output of *f* becomes the input of *g*. In mathematics, this operation is called a *composition*. So the composition operation takes two functions *f* and *g* and produces a function *h* such that *h*(*x*) = *g*(*f*(*x*)). We can easily implement it using a closure:

# Listing 17def compose(g, f):  
 def h(\*args, \*\*kwargs):  
 return g(f(\*args, \*\*kwargs))  
 return h

Here h is a closure since it captures the nonlocal variables f and g. These nonlocal variables are functions themselves. What this closure returns is the composition of f and g which is g(f(\*args, \*\*kwargs)). We have used \*args and \*\*kwargs to be able to pass multiple arguments or keyword arguments to h,

Now here is a simple application. Imagine that we have some functions to convert the units. For example, suppose that we have two functions that convert *Inch* to *Foot* and *Foot* to *Meter* respectively. Now we can use our compose() function to combine them into one function that converts *Inch* to *Meter*:

# Listing 18inch\_to\_foot= lambda x: x/12  
foot\_meter= lambda x: x \* 0.3048inch\_to\_meter = compose(foot\_meter, inch\_to\_foot)  
inch\_to\_meter(12) # Output 0.3048

**Partial application**

In mathematics, the number of arguments that a function takes is called the *arity* of that function. Partial application is an operation that reduces the arity of a function. It means it allows you to fix the values of some of the arguments and freeze them to get a function with fewer parameters. So it somehow simplifies the function.

For example, the arity of *f*(*x, y, z)*is 3.We can fix the value of argument *x* at *a* to have *f(x=a, y ,z) = g(y,z)*.Now the arity of *g(y,z)*is 2, and it is the result of partial application of *f(x, y, z)*. So partial(*f*) => *g*. We can implement the partial application using closures:

# Listing 19def partial(f, \*f\_args, \*\*f\_keywords):  
 def g(\*args, \*\*keywords):  
 new\_keywords = f\_keywords.copy()  
 new\_keywords.update(keywords)  
 return f(\*(f\_args + args), \*\*new\_keywords)  
 return g

Here the outer function receives f and the positional and keyword argument of f that need to be fixed. The inner function g adds these fixed arguments to the remaining arguments of f that it receives later as the partial function. Finally, it calls f with all the gathered arguments. Listing 20 gives an example of using the partial function:

# Listing 20func = lambda x,y,z: x\*\*2 + 2\*y + zpfunc = partial(func, 1)  
pfunc(2, 3) # Output is 8

Here we fix the value of x at 1 and convert func to pfunc which now has two arguments y and z.

**Currying**

In mathematics, currying means transforming a function with multiple parameters into a sequence of nested unary functions. As mentioned before a unary function is a function that only takes one argument. So for example, if we have a function *f(x, y, z)*. Currying transforms it to *g(x)(y)(z)* = *((g(x))(y))(z)*. Listing 21 shows how we can implement it in Python:

# Listing 21def curry(f):  
 argc = f.\_\_code\_\_.co\_argcount  
 f\_args = []  
 f\_kwargs = {}  
 def g(\*args, \*\*kwargs):  
 nonlocal f\_args, f\_kwargs  
 f\_args += args  
 f\_kwargs.update(kwargs)  
 if len(f\_args)+len(f\_kwargs) == argc:  
 return f(\*f\_args, \*\*f\_kwargs)  
 else:  
 return g   
 return g

Now we can apply it to the function defined in Listing 20:

cfunc = curry(func)  
cfunc(1)(2)# Output:  
# <function \_\_main\_\_.curry.<locals>.g(\*args, \*\*kwargs)>

and

cfunc(3) # Output is 8

Again we are using a closure. But this closure g is a recursive function. Figure 5 shows how this function works. First, the outer function curry receives f as an argument. f is the function that should undergo the currying. We get the number of argument that f takes (the arity of f) using the co\_argcount attribute and stores it in argc. In this example args=3. We have two nonlocal variables f\_args and f\_kwargs which are used to store the arguments that g takes. curry returns the closure g and assign it to cfunc. So when we call cfunc(1)(2) we are calling g(1)(2). First g(1) will be evaluated.

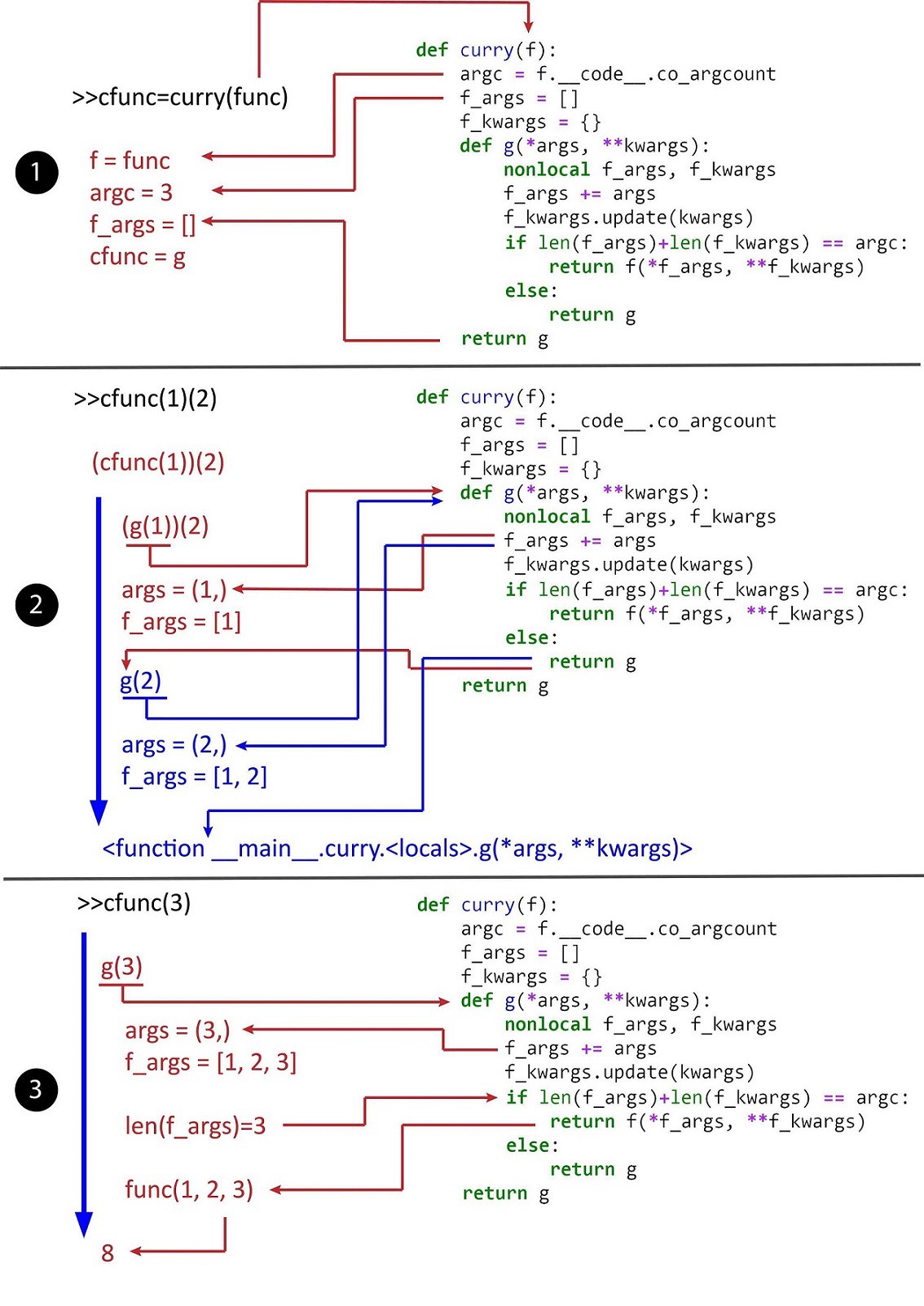


Figure 5

g adds its argument to f\_args list and since it hasn’t captured all the arguments of func, it returns itself recursively. So the result of g(1) is g and g(1)(2) turns into g(2). now g(2) is evaluated. Again g adds its argument to f\_args list, and f\_args=[1,2]. g returns itself again, but it has no arguments to be evaluated at the moment. As a result, the final output of cfunc(1)(2) is g which is shown as <function \_\_main\_\_.curry.<locals>.g(\*args, \*\*kwargs)> in Python. Then we run cfunc(3). This time g(3) is evaluated and 3 is added to f\_args, and it will be equal to [1,2,3]. Now len(f\_args) is equal to argc, and all the original arguments of func have been captured, so finally func(1,2,3) will be evaluated and returned by g.

You should know that the curry function in Listing 21 can do more than a mathematical curry operation. The output of this function is not limited to nested unary functions, and we can have nested functions with a higher arity. For example we could also write:

cfunc = curry(func)  
cfunc(1, 2)  
cfunc(3) # Output is 8

and get the same result. This is like converting *f(x, y, z)* to *g(x, y)(z)*.

**Decoration and decorators**

Before talking about decorators, I need to mention something about the functions in Python. When you define a function in Python, the name of that function is simply a reference to the body of the function (function definition). So by assigning a new value to it, you can force it to refer to another function definition. Listing 22 gives a simple example:

# Listing 22def f():  
 return("f definition")  
def g():  
 return("g definition")print("f is referring to ", f())  
print("g is referring to ", g())print("Swapping f and g")  
temp = f  
f = g  
g = tempprint("f is referring to ", f())  
print("g is referring to ", g())

Output:

f is referring to f definition  
g is referring to g definition  
Swapping f and g  
f is referring to g definition  
g is referring to f definition

So far we have created a closure by assigning the result of the outer function to a new variable:

h=f(a)

Now suppose that we have an outer function called deco(f) which defined in Listing 23:

# Listing 23def deco(f):  
 def g(\*args, \*\*kwargs):  
 return f(\*args, \*\*kwargs)  
 return gdef func(x):  
 return 2\*xfunc = deco(func)  
func(2) # Output is 4

Here the outer function is deco(f) which takes a function f as an argument, and the inner function g is defined to be a closure. We define another function func and apply deco to it. But to initialize the closure we assign the result of deco to func. So deco takes func as an argument and assigns its closure to func again. In this case, we say that func is*decorated*bydeco and deco is a *decorator* for func.

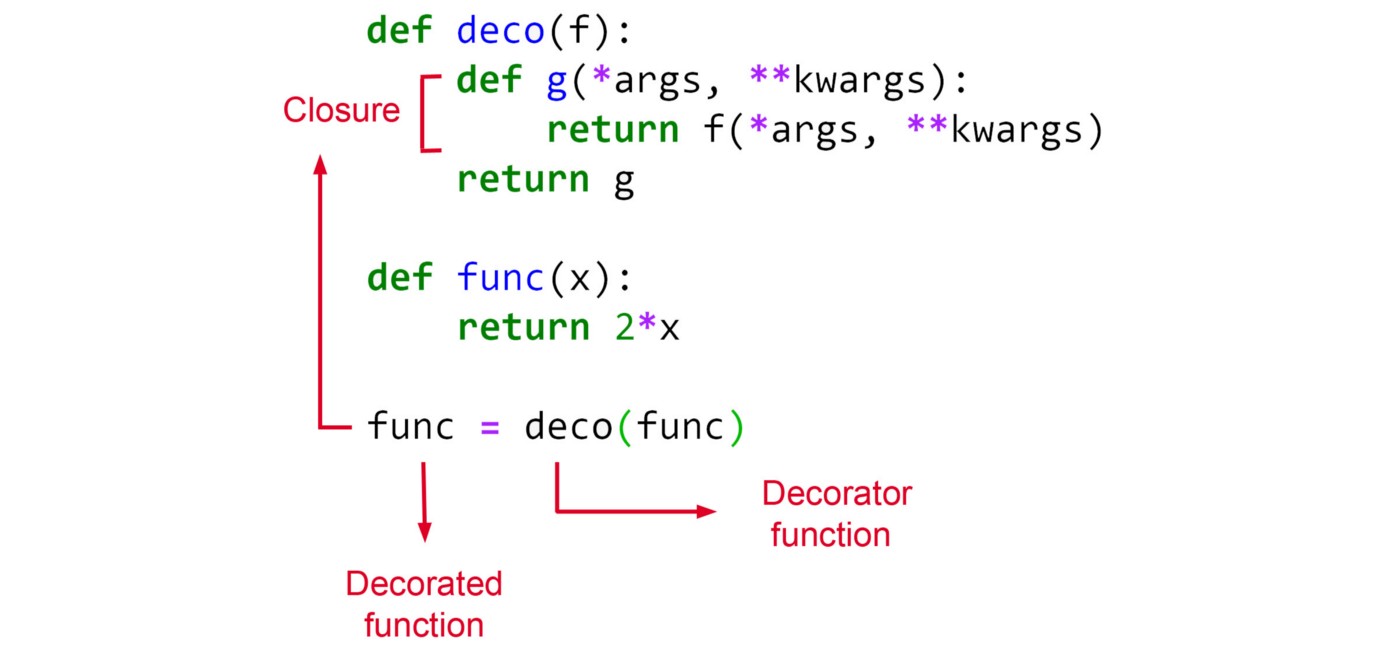


Figure 6

It is important to note the after assigning the result of the decorator to func, func refers to the closure g. So calling func(a) is like calling g(a). If we write:

func.\_\_name\_\_

the output will be 'g'. In fact, the variable func is simply a reference to the function definition. Initially, it refers to the func(x) definition, but after decoration, it refers to g(\*args, \*\*kwargs).

But what happens for the original function func(x)? Are we going to lose it? Remember that the decorator receives func as its argument. So it has a copy of the reference to func as its local variable f. If the original reference changes, it does not affect this local variable, so inside g, f still refers to func(x) definition (by function definition, I mean the actual body of the function func(x) as shown in Fig 7).

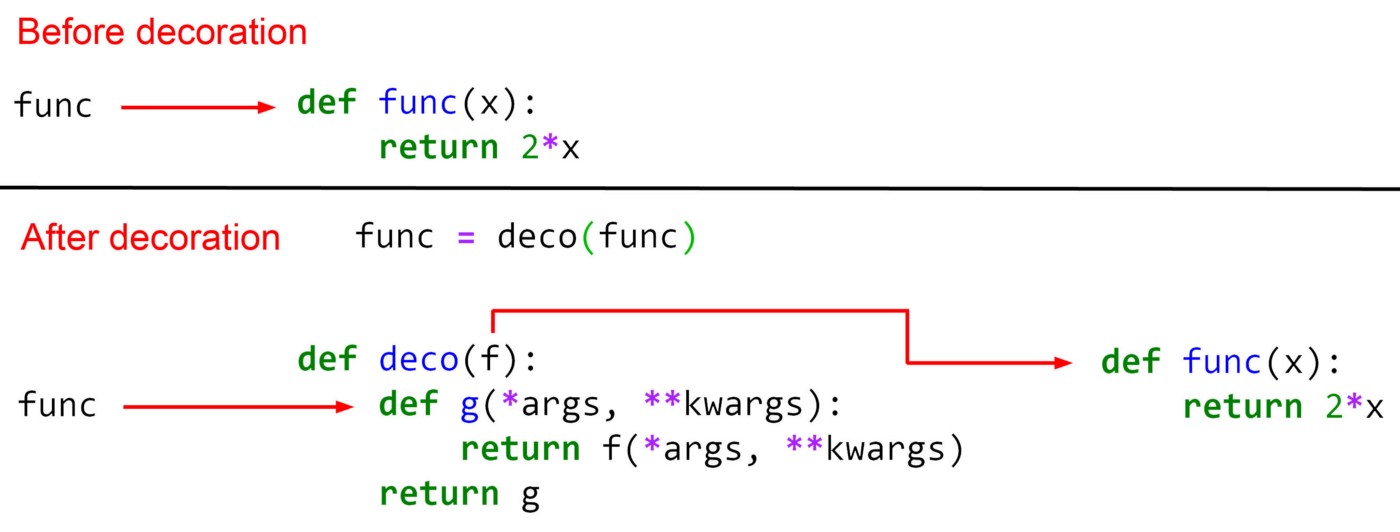


Figure 7

So to summarize it, after decoration, the variable func refers to the closure g, and inside g, f refers to the func(x) definition. In fact g is now acting as an interface for the original function func(x) which was decorated. We cannot directly call func(x)outside g. Instead, we first call func to call g, and then inside g we can call f to call the original function func(x). So we are calling the original function func(x) using the closure g.

Now using this closure, we can add some more code to be run before or after calling func(x). Let me give you a simple example. Suppose that we want to debug our code in Listing 23. We want to know when func(x) was called. We could simply place a print statement inside func(x) to let us know when it is running:

def func(x):  
 print("func is called")   
 return 2\*x

But it changes the function, and we have to remember to remove it later. A better solution is to define a decorator to wrap the function and add the print statement to that closure.

# Listing 24def deco(f):  
 def g(\*args, \*\*kwargs):  
 print("Calling ", f.\_\_name\_\_)  
 return f(\*args, \*\*kwargs)  
 return gdef func(x):  
 return 2\*xfunc = deco(func)  
func(2)

The output is:

Calling func  
4

This example shows how useful a decorator can be. Now we can see some more applications of desecrates:

**Memoization**

Memoisation is a programming technique which is used to speed up the programs. It was derived from the Latin word *memorandum*which means “to be remembered”. As the name suggests, it is based on memorizing or caching the results of expensive function calls. If the same input or a function call with the same parameters is used, the previously cached results will be used to avoid unnecessary calculations. In Python, we can automatically memoize functions using closures and decorators.

Listing 25 shows a function that calculates the Fibonacci numbers. The Fibonacci numbers are defined recursively. Each number is the sum of the two preceding ones, starting from 0 and 1:

# Listing 25def fib(n):  
 if n == 0:  
 return 0  
 elif n == 1:  
 return 1  
 else:  
 return fib(n-1) + fib(n-2)  
for i in range(6):  
 print(fib(i), end=" ")# Output  
# 0 1 1 2 3 5

Now we define a new function which can memoize another function:

# Listing 26def memoize(f):  
 memo = {}  
 def memoized\_func(n):  
 if n not in memo:   
 memo[n] = f(n)  
 return memo[n]  
 return memoized\_func

To use the memoize function, we can use it as a decorator for fib :

fib = memoize(fib)  
fib(30) # Output is 832040

So the function fib is now decorated by the memoize() . Now fib refers to the closure memoized\_func, so when we call fib(30), it is like calling memoized\_func(30). The decorator receives the original fib as its argument f, so f refers to fib(n) definition inside memoized\_func. The closure memoized\_func first checks if the n is in the memo dictionary or not. If it is in it, it simply returns memo[n], and does not call the original fib(n). if n is not in the memo dictionary, it first calls f(n) which refers to the original fib(n). It then stores the result in the memo dictionary, and finally returns it as the final result.

**Tracing recursive functions**

Suppose that we want to debug the recursive Fibonacci function that we defined in Listing 25. We want to see how this function calls itself to calculate the final result. We can define a decorator to track the function calls:

# Listing 27def trace(f):  
 level = 1  
 def helper(\*arg):  
 nonlocal level  
 print((level-1)\*" │", " ┌", f.\_\_name\_\_,  
 "(", ",".join(map(str, arg)), ")", sep="")  
 level += 1  
 result = f(\*arg)  
 level -= 1  
 print((level-1)\*" │", " └", result, sep="")  
 return result  
 return helper

Now we can write:

def fib(n):  
 if n == 0:  
 return 0  
 elif n == 1:  
 return 1  
 else:  
 return fib(n-1) + fib(n-2)  
fib = trace(fib)  
fib(4)

And the output is:

┌fib(4)  
 │ ┌fib(3)  
 │ │ ┌fib(2)  
 │ │ │ ┌fib(1)  
 │ │ │ └1  
 │ │ │ ┌fib(0)  
 │ │ │ └0  
 │ │ └1  
 │ │ ┌fib(1)  
 │ │ └1  
 │ └2  
 │ ┌fib(2)  
 │ │ ┌fib(1)  
 │ │ └1  
 │ │ ┌fib(0)  
 │ │ └0  
 │ └1  
 └33

The decorator function trace receives fib as its argument. After calling the decorator, fib refers to the closure helper. trace has a nonlocal variable called level which shows the recursion depth and is initially set to 1. helper first prints a few padding characters including │, ┌, and to show the recursion tree structure. The number of │ characters are proportional to the value of level.

As the recursion depth increases, more │ are printed. For the first call | is not printed since level-1 is equal to zero. Then the name of the function and the current values of its parameters are printed. After that level is increased by one to show the next level of recursion. Then function f (which refers to the original function passed to the decorator) is evaluated. So the fib(n) definition is evaluated. Since fib is defined recursively, it will call itself at some point. Remember that after decoration, fib refers to the closure helper. So the recursive call will call the helper again, and the next level of recursion will be printed.

The recursion continues until we reach the recursion base case and the function definition of fib(n) returns 1. Then level is decreased by one to connect the returned result with the name of function that returned it. The returned values are printed one by one until we get back to the first call.

**Syntactic sugar**

Syntactic sugar is syntax within a programming language that makes things easier to read or to express. Using syntactic sugar you can write the code more clearly or more concisely. For example, in Python, the += operator is syntactic sugar. So instead of writing a=a+1 you can simply write a+=1.

In Listing 24, we wrote func = deco(func) to decorate the function func. Actually, we can do it in a simpler way, and the end result is the same:

# Listing 28def deco(f):  
 def g(\*args, \*\*kwargs):  
 print("Calling ", f.\_\_name\_\_)  
 return f(\*args, \*\*kwargs)  
 return g@deco  
def func(x):  
 return 2\*xfunc(2)

Here instead of writing func = deco(func) we simply write @deco on top of the function that should be decorated by deco(f). This is syntactic sugar to decorate a function and sometimes is called the *pie* *syntax*.

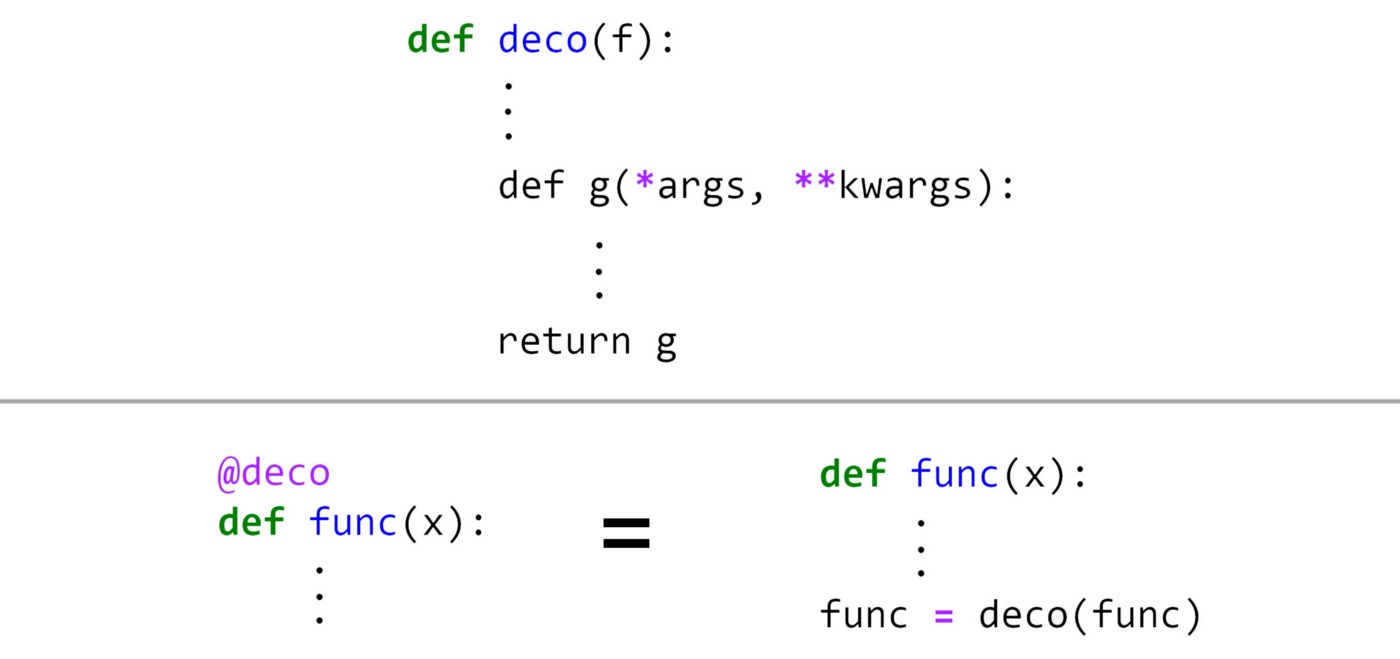


Figure 8

A few notes about this syntax. @deco cannot be anywhere else and no more statements are allowed between @deco and the definition of func(x). You should not write the parameter of deco after @deco. The syntax assumes that deco only takes one parameter which is the function that should be decorated. Later I will explain what to do if the decorator function has more parameters.

In Listing 24, the decoration starts once you write func = deco(func). But when does it happen in Listing 28? A key feature of decorators defined with the pie syntax is that they run right after the decorated function is defined. That is usually at import time when the module is loaded by Python.

**Stacked decorators**

In Listing 29 we have defined two decorators:

# Listing 29def deco1(f):  
 def g1(\*args, \*\*kwargs):  
 print("Calling ", f.\_\_name\_\_, "using deco1")  
 return f(\*args, \*\*kwargs)  
 return g1def deco2(f):  
 def g2(\*args, \*\*kwargs):  
 print("Calling ", f.\_\_name\_\_, "using deco2")  
 return f(\*args, \*\*kwargs)  
 return g2def func(x):  
 return 2\*xfunc = deco2(deco1(func))  
func(2)

Here we first apply deco1 to func and then apply deco2 to the closure returned by deco1. So we have two stacked decorators. We first decorate func with deco1 and then with deco2. The output is:

Calling g1 using deco2  
Calling func using deco1  
4

In fact g2 and g1 are now acting as interfaces for the original function func(x) which was decorated. To reach the original function func(x), we first call func which refers to g2. Inside g2, the parameter f refers to g1. So by calling f, we are calling g1, and then inside g1 we can call f to call the original function func(x).

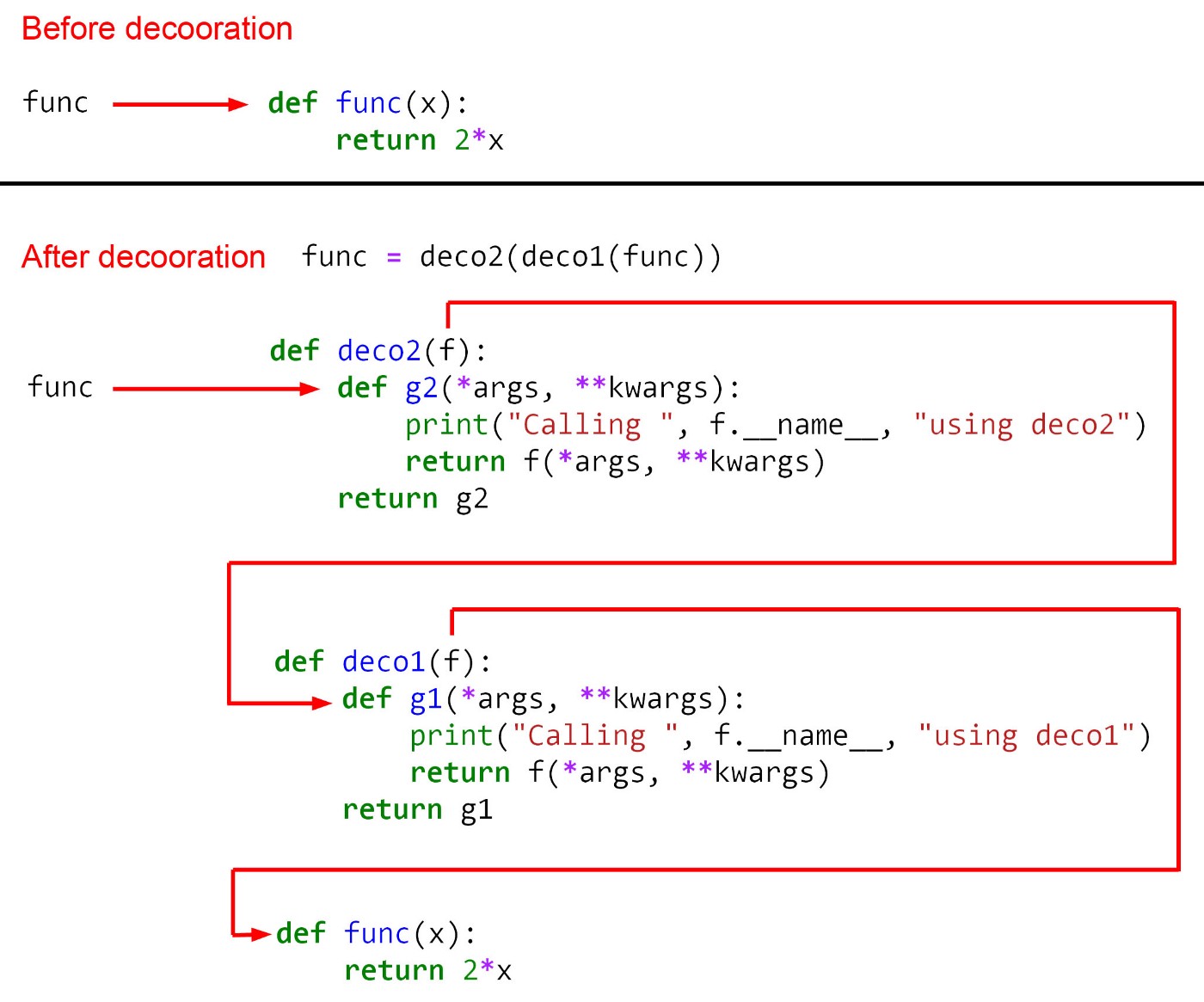


Figure 9

We can also use the pie syntax to apply stacked decorators to a function. To do that we can write:

# Listing 30def deco1(f):  
 def g1(\*args, \*\*kwargs):  
 print("Calling ", f.\_\_name\_\_, "using deco1")  
 return f(\*args, \*\*kwargs)  
 return g1def deco2(f):  
 def g2(\*args, \*\*kwargs):  
 print("Calling ", f.\_\_name\_\_, "using deco2")  
 return f(\*args, \*\*kwargs)  
 return g2@deco2  
@deco1  
def func(x):  
 return 2\*xfunc(2)

Here the decorators are stacked on the top of the function definition in the order they wrap the function.

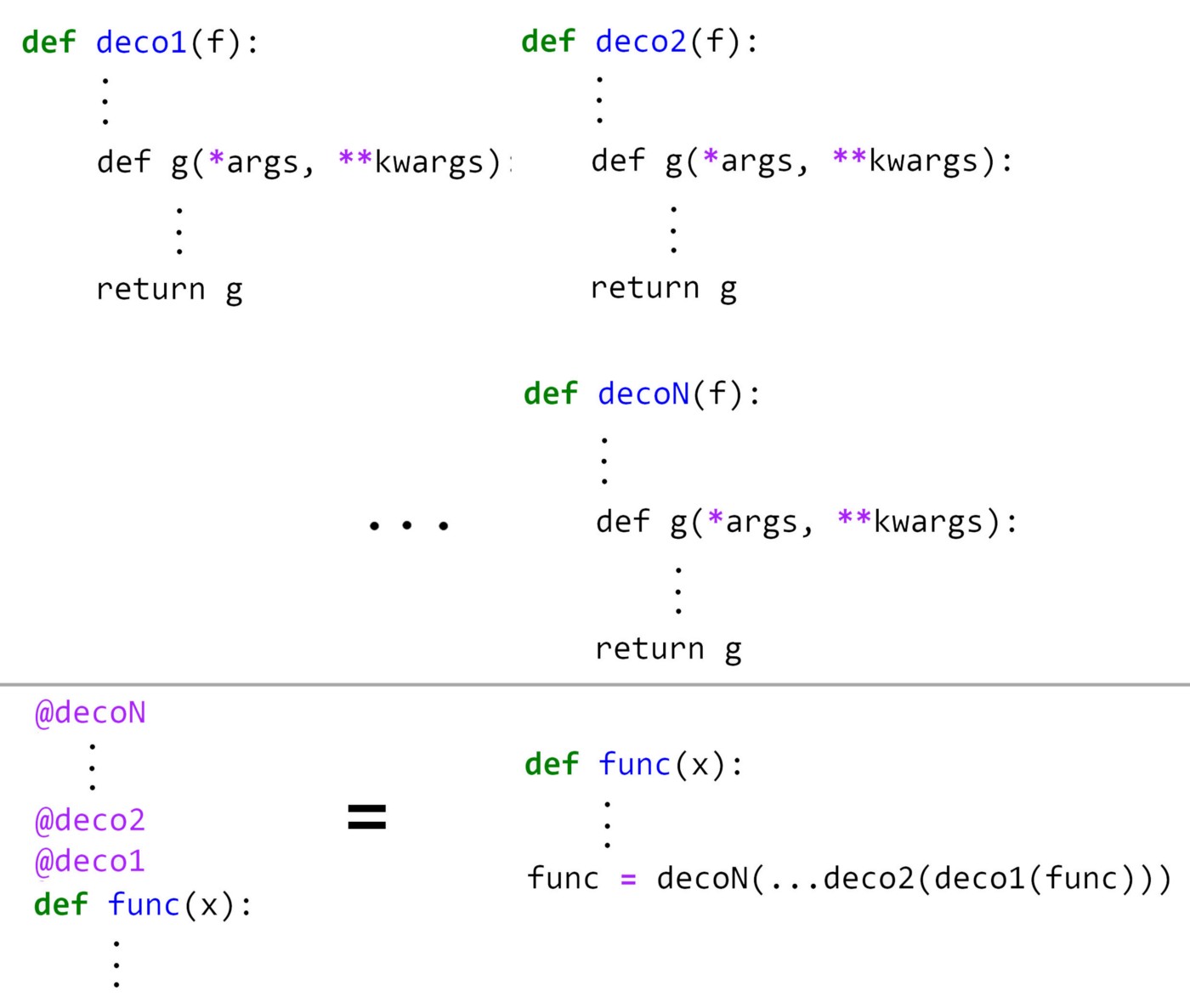


Figure 10

We could also use the function compose that was introduced in Listing 17 to combine these two decorators before applying them to the target function. The composition operation can take two functions deco1(f) and deco2(f) and produce a function deco such that deco(f) = deco2(deco1(f)). This is shown in Listing 31.

# Listing 31def deco1(f):  
 def g1(\*args, \*\*kwargs):  
 print("Calling ", f.\_\_name\_\_, "using deco1")  
 return f(\*args, \*\*kwargs)  
 return g1  
def deco2(f):  
 def g2(\*args, \*\*kwargs):  
 print("Calling ", f.\_\_name\_\_, "using deco2")  
 return f(\*args, \*\*kwargs)  
 return g2deco = compose(deco2, deco1)   
@deco  
def func(x):  
 return 2\*x  
func(2)

Now we can see an example of stacked decorators. Remember that we defined a decorator for memoization. We also defined a decorator to trace recursive functions. Now we can combine them to trace a recursive function with memoization. We first decorate the Fibonacci function with the memoize decorator given in Listing 26 and then with the trace decorator given in Listing 27. We first try to calculate fib(5).

# Listing 32@trace  
@memoize  
def fib(n):  
 if n == 0:  
 return 0  
 elif n == 1:  
 return 1  
 else:  
 return fib(n-1) + fib(n-2)fib(5)

The output is:

┌memoized\_func(5)  
 │ ┌memoized\_func(4)  
 │ │ ┌memoized\_func(3)  
 │ │ │ ┌memoized\_func(2)  
 │ │ │ │ ┌memoized\_func(1)  
 │ │ │ │ └1  
 │ │ │ │ ┌memoized\_func(0)  
 │ │ │ │ └0  
 │ │ │ └1  
 │ │ │ ┌memoized\_func(1)  
 │ │ │ └1  
 │ │ └2  
 │ │ ┌memoized\_func(2)  
 │ │ └1  
 │ └3  
 │ ┌memoized\_func(3)  
 │ └2  
 └55

Since the memo dictionary is initially empty, it will do all the recursive calls the calculate the result. Now if we try fib(6) we get:

┌memoized\_func(6)  
 │ ┌memoized\_func(5)  
 │ └5  
 │ ┌memoized\_func(4)  
 │ └3  
 └8  
8

This time the result of fib(5) and fib(4) are already stored in the memo dictionary. So it only needs to retrieve them to calculate the result, and no more recursion is necessary. You may have noticed that the name of function fib is not printed in the recursion tree. Instead, we see the name of closure memoized\_func. The reason is that we have two stacked decorators and the trace decorator receives the memoized\_func as its parameter, not the function fib. So inside trace now f refers to memoized\_func, and f.\_\_name\_\_ returns memoized\_func. In the next section, I will show you how we can solve this problem.

**Decorators with additional parameters**

The decorators discussed so far only take one parameter which is the decorated function, however, we can have additional parameters. Remember the decorator that we saw in Listing 24. It prints a message before calling plus the name of the decorated function. Now we want to create a decorator that takes the message as an additional parameter. In fact, we want to pass two additional parameters to the decorator. One is a message that will be printed before calling the decorated function, and the other is a message that will be printed after calling it.

# Listing 33def deco(msg\_before, msg\_after):  
 def original\_deco(f):  
 def g(\*args, \*\*kwargs):  
 print(msg\_before + " " + f.\_\_name\_\_)  
 result = f(\*args, \*\*kwargs)  
 print(msg\_after + " " + f.\_\_name\_\_)  
 return result  
 return g  
 return original\_deco  
def func(x):  
 return 2\*x  
func = deco("Starting", "Finished")(func)  
func(2)

The output is:

Starting func  
Finished func  
4

As you see to pass the additional parameters to the decorator, we need to wrap the function which was the original decorator with another function which takes these additional parameters. Now, this function will be the decorator and the original decorator will be a nested closure inside it. To call the decorator we need to first pass the additional parameters to deco and then the decorated function f will be passed to the closure that it returns (which is the original decorator). We can also use the pie syntax as shown in Listing 34 and Figure 11.

# Listing 34def deco(msg\_before, msg\_after):  
 def original\_deco(f):  
 def g(\*args, \*\*kwargs):  
 print(msg\_before + " " + f.\_\_name\_\_)  
 result = f(\*args, \*\*kwargs)  
 print(msg\_after + " " + f.\_\_name\_\_)  
 return result  
 return g  
 return original\_deco@deco("Starting", "Finished")  
def func(x):  
 return 2\*xfunc(2)

Then we only need to pass the additional parameters to @deco.

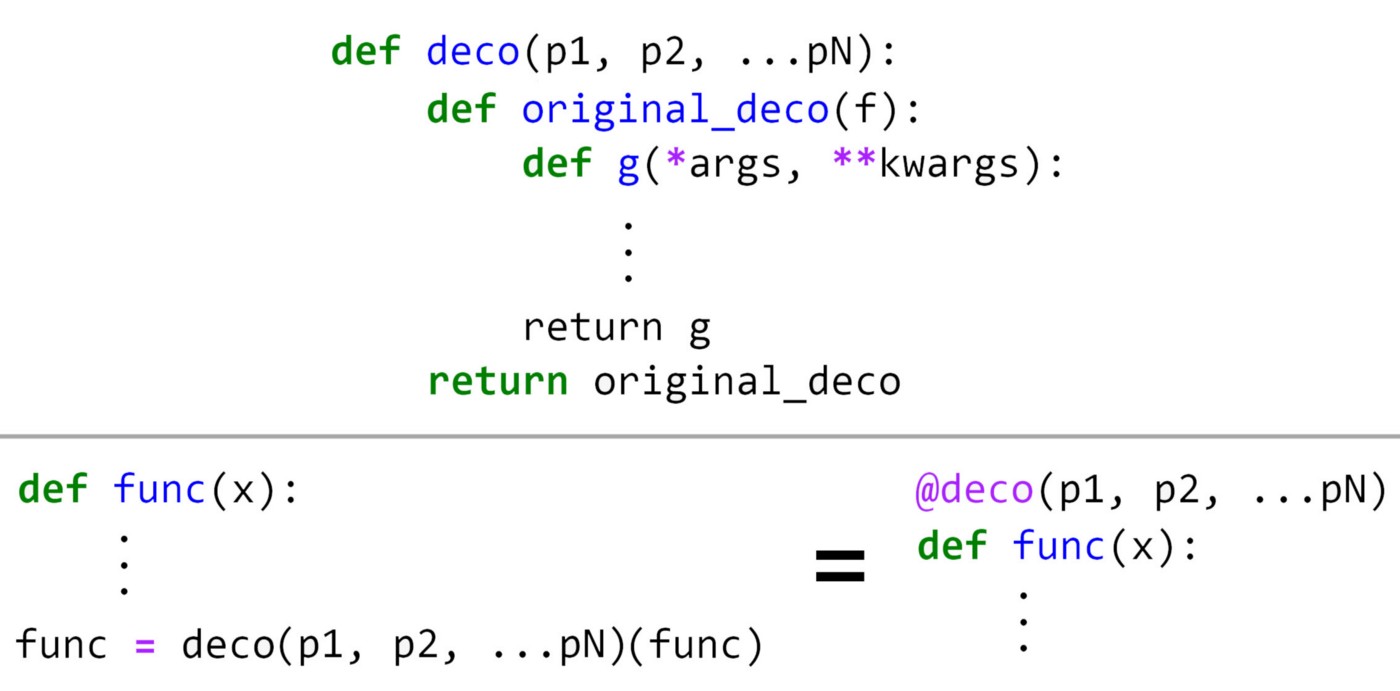


Figure 11

You may ask why we cannot pass the additional parameters to the original decorator? Actually we can:

# Listing 35def deco(msg\_before, msg\_after, f):  
 def g(\*args, \*\*kwargs):  
 print(msg\_before + " " + f.\_\_name\_\_)  
 result = f(\*args, \*\*kwargs)  
 print(msg\_after + " " + f.\_\_name\_\_)  
 return result  
 return g  
   
def func(x):  
 return 2\*x  
func = deco("Starting", "Finished", func)  
func(2)  
sds

Here we simply pass the additional parameters along with the function name. The problem is that you cannot use pie syntax if you define it this way. So if you write:

# This does not work:  
@deco("Starting", "Finished")  
def func(x):  
 return 2\*x  
#func = deco("Starting", "Finished")  
func(2)

Python gives an error. It is also possible to use the function curry that we defined in Listing 21 to simplify the decorator function. The curry function takes another function as its parameter and returns a closure, so it can be used to decorate our decorator. So we can replace Listing 34 with:

# Listing 36@curry  
def deco(msg\_before, msg\_after, f):  
 def g(\*args, \*\*kwargs):  
 print(msg\_before + " " + f.\_\_name\_\_)  
 result = f(\*args, \*\*kwargs)  
 print(msg\_after + " " + f.\_\_name\_\_)  
 return result  
 return g  
   
@deco("Starting", "Finished")  
def func(x):  
 return 2\*xfunc(2)

After curring the function deco can take its parameters separately. So by writing @deco(“Starting”, “Finished”) we provide the first two parameters of deco, and now it only takes one parameter which is the decorated function. So we can use the pie syntax, and we don't need to define another function to take the additional parameters.

We can now go back to the problem that we had in the previous section. We want to keep the signature of the decorated function after decoration. We can define a new decorator for this purpose:

# Listing 37def wraps(f):  
 def decorator(g):  
 def helper(\*args, \*\*kwargs):  
 return g(\*args, \*\*kwargs)  
 attributes = ('\_\_module\_\_', '\_\_name\_\_', '\_\_qualname\_\_',  
 '\_\_doc\_\_', '\_\_annotations\_\_')   
 for attr in attributes:  
 try:  
 value = getattr(f, attr)  
 except AttributeError:  
 pass  
 else:  
 setattr(helper, attr, value)  
 return helper  
 return decorator

To understand this decorator, suppose that we have a function named f and we want to decorate it with another function g. But we also want the closure returned by g (which is helper) to have the same signature of f so that we can keep the signature of f after decoration. wraps is a decorator that receives f as an additional parameter, and decorates the function g passed to its nested closure decorate. It then gets all the attributes of f using getattr() and assigns them to the closure helper which is finally returned as the result of decoration. Now helper has the signature of f.

We add this decorator to the memoize function defined in Listing 26. Here wraps keeps the signature of the function f that should be memoized.

# Listing 38def memoize(f):  
 memo = {}  
 @wraps(f)  
 def memoized\_func(n):  
 if n not in memo:   
 memo[n] = f(n)  
 return memo[n]  
 return memoized\_func

The function trace remains the same, and we can use it with memoize to trace fib properly:

# Listing 39@trace  
@memoize  
def fib(n):  
 if n == 0:  
 return 0  
 elif n == 1:  
 return 1  
 else:  
 return fib(n-1) + fib(n-2)fib(5)

Now the output shows the correct function name:

┌fib(5)  
 │ ┌fib(4)  
 │ │ ┌fib(3)  
 │ │ │ ┌fib(2)  
 │ │ │ │ ┌fib(1)  
 │ │ │ │ └1  
 │ │ │ │ ┌fib(0)  
 │ │ │ │ └0  
 │ │ │ └1  
 │ │ │ ┌fib(1)  
 │ │ │ └1  
 │ │ └2  
 │ │ ┌fib(2)  
 │ │ └1  
 │ └3  
 │ ┌fib(3)  
 │ └2  
 └55

And we can check the memoization by evaluating fib(6). The output is:

┌fib(6)  
 │ ┌fib(5)  
 │ └5  
 │ ┌fib(4)  
 │ └3  
 └8  
8

This example shows how different types of decorators can combine together to produce the desired output.

Decorators are a useful tool for metaprogramming in Python. Metaprogramming is about creating functions whose main goal is to manipulate code. The examples in this article showed how you can change the behavior of function by wrapping them with different decorators. I hope that you enjoyed reading this article. Please let me know if you have any questions or suggestions. All Code Listings in this article are available for download as a Jupyter notebook from GitHub at: <https://github.com/reza-bagheri/Closures-and-decorators>