Table of Contents

Global & Local Scopes

Scopes & Namespaces

When an object is assigned to a variable

```
a = 10
```

the variable points to some object and we say that the variable name is bound to that object.

This object can be accessed using that name in various parts of our code.

But not just anywhere!

That variable name and it's binding (name and object) only exist in specific parts of our code.

The portion of code where the name/binding is defined is called the lexical scope of the variable

These bindings are stored in namespaces
(each scope has its own namespace)

The Global Scope

The global scope is essentially the module scope

It spans a single only

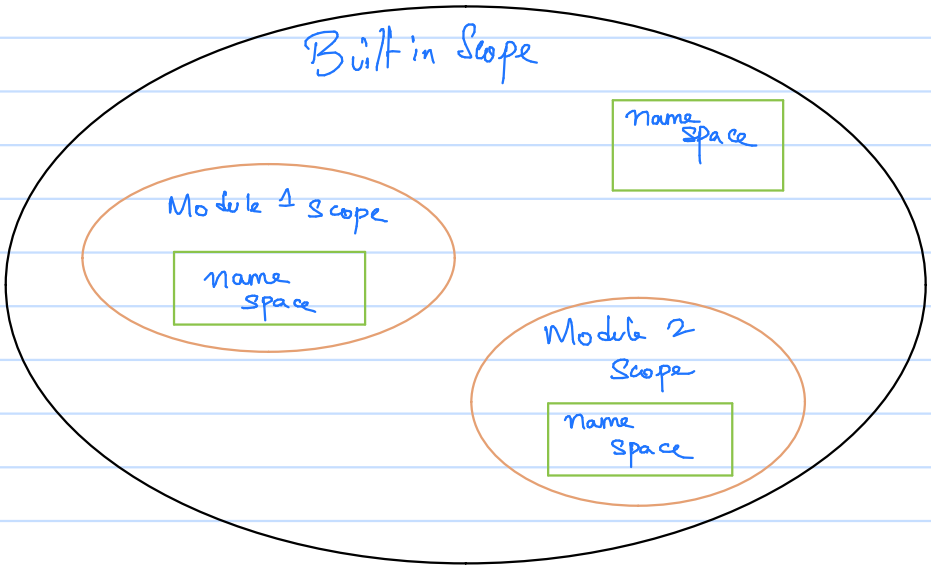
There is no concept of a truly global (across all modules in our entire app) scope in Python.

The only exception to this are some of the built-in globally available objects such as

True False None dict print

The built-in and global variables can be used anywhere inside our module.

Global scope are nested inside built in scope



Name Space	
var1	0xA3FE3
func1	0xFF34A

If you reference a variable name inside a scope and Python doesn't find it in that scope's namespace it will look for it in an enclosing scope's namespace.

Examples:

module1.py
```
print(True)
```
Python doesn't find any object 'print' in module scope so it looks in it's enclosing scope (built-in)

module2.py
```
print(a)
```
Python doesn't find 'a' in module scope, even doesn't find in built-in scope → throws run-time error.

module3.py
```
print = lambda x: 'hello {0}'.format(x)
print('world')
```
Python finds print in module scope itself so will use that only & print "hello world".

The Local Scope

When we create functions, we can create variables inside those functions (using arguments or declaring inside function body)

eg
```
def fn(a):
    b = 10
```

The variables defined inside a function is not created until the function is called.

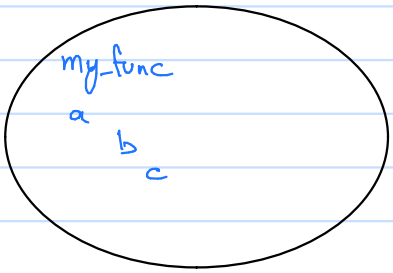
Every time the function is called, new scope is created.

Variables defined inside that function are assigned to that scope.

The actual objects the variable references could be different each time function is called
(this is why Recursion works)

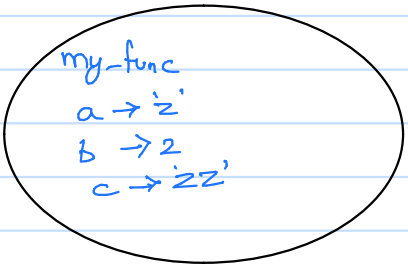
Examples

```
def my_func(a, b):
    c = a*b
    return c
```

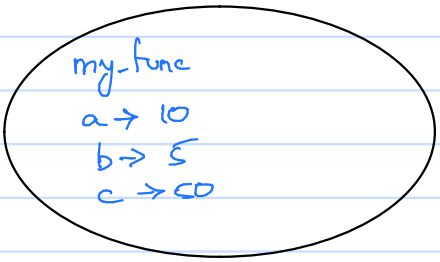


a, b, c are considered local to 'my_func'

my_func('z', 2)



my_func(10, 5)



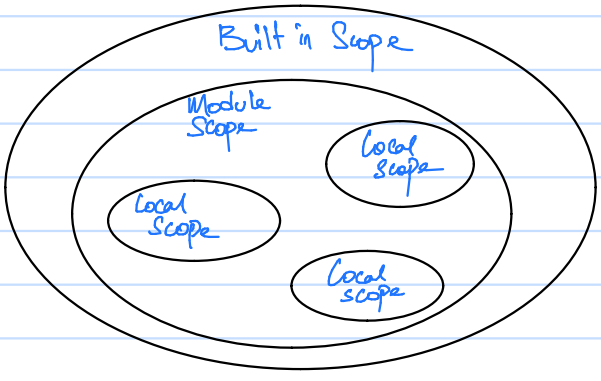
same names, different local scopes

Nested Scopes:

Scopes are often nested

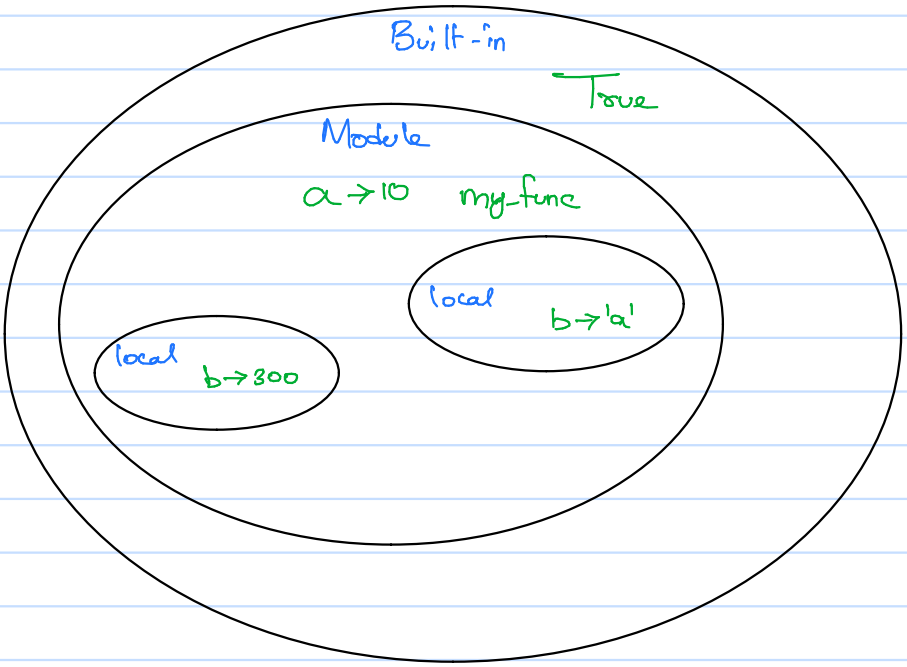
Namespace lookup:

When requesting object having bound to variable name, Python will try to find object in current local scope & works up chain of enclosing scopes.



Example:

```
a = 10
def my_func(b):
    print(True)
    print(a)
    print(b)

my_func(300)
my_func('a')
```



Remember reference counting ?

When my_func(var) finishes running after calling, the scope is gone too.

The reference count of the object 'var' was bound to, is decremented.

We also say that var goes out of scope.

Accessing global variable from local Scope:

```
a = 0
def func():
    a = 100
    print(a)
```

Here Python interprets this as a local variable (at compile time)
Any assignment creates a local variable

The local variable 'a' masks the global variable 'a'.

```
func()    // Print  100
print(a)  // Print  0
```

The global keyword

```
a = 0
def func():
    global a
    a = 100
    print(a)
```

We can ask Python to use global a inside local scope by using the keyword global

```
func()    // Print  100
print(a)  // Print  100
```

Global and Local Scoping

When Python encounters a function defined at compile-time.

It will scan for any labels (variables) that have values assigned to them (anywhere in the func), if the label has not been specified as global, then it will be local.

Variables that are referenced but not assigned a value anywhere in the function will not be local, and Python will, at run-time look for them in enclosing scopes.

```
a = 10

def func1():
    print(a)
```

a is only referenced in entire function at compile time
a → non local

```
def func2():
    a = 100
```

assignment at compile time
a → local

```
def func3():
    global a
    a = 100
```

assignment at compile time. But 'global' being used
a → global

```
def func4():
    print(a)
    a = 100
```

assignment at compile time
a → local.

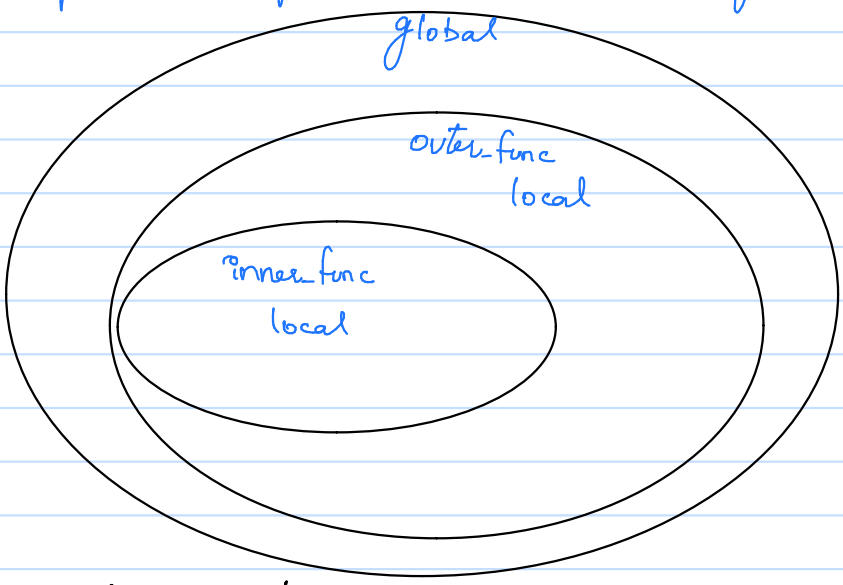
(But this leads to run-time error. Since at run-time 'a' is local variable, but print(a) is befor assignment, so error 'a' not found)

Non Local Scopes

Inner Function

We can define functions from inside another function.

```
def outer_func():
    # some code

    def inner_func():
        # some code

    inner_func()

outer_func()
```



Nested local scopes

Both the functions have access to the global & built-in scopes as well as their respective local scopes.

But the inner function also has access to its enclosing scope - the scope of the outer function.

That scope is neither local (to inner_func) nor global is called non-local scope.

Referencing variables from enclosing scope

module1.py

```
a = 10
def func():
    print(a)

func()
```

When we call func, Python sees the references to 'a'.
Since 'a' is not in the local scope, it looks in it's enclosing scope

module2.py

```
def outer()
    a = 10
    def inner()
        print(a)
    inner()
outer()
```

When we call outer, 'inner' is created and called

When 'inner' is called, Python doesn't find 'a' in local scope & search in 'its enclosing scope, in this case outer's scope which is 'its non-local scope

Modifying non-local variables

Just as with global variables, we to explicitly tell Python we are modifying a non local variable

We can do using nonlocal keyword

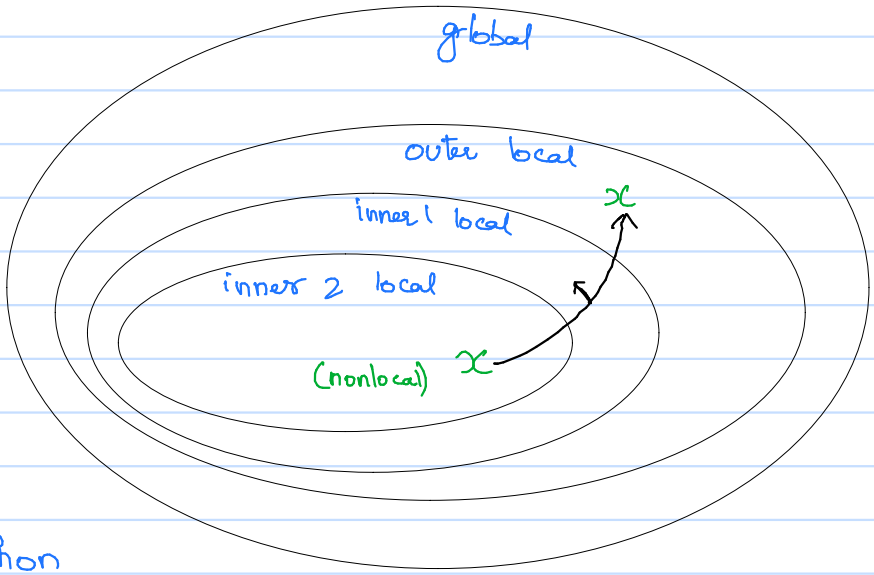
```
def outer_func():
    a = 10
    def inner_func():
        nonlocal a
        a = 15
        print(a)    // a = 15
    inner_func()
    print(a)        // a = 15
```

Whenever Python is told that a variable is nonlocal

'it will look for 'it 'in the enclosing local scopes chain until it first encounters the specified variable

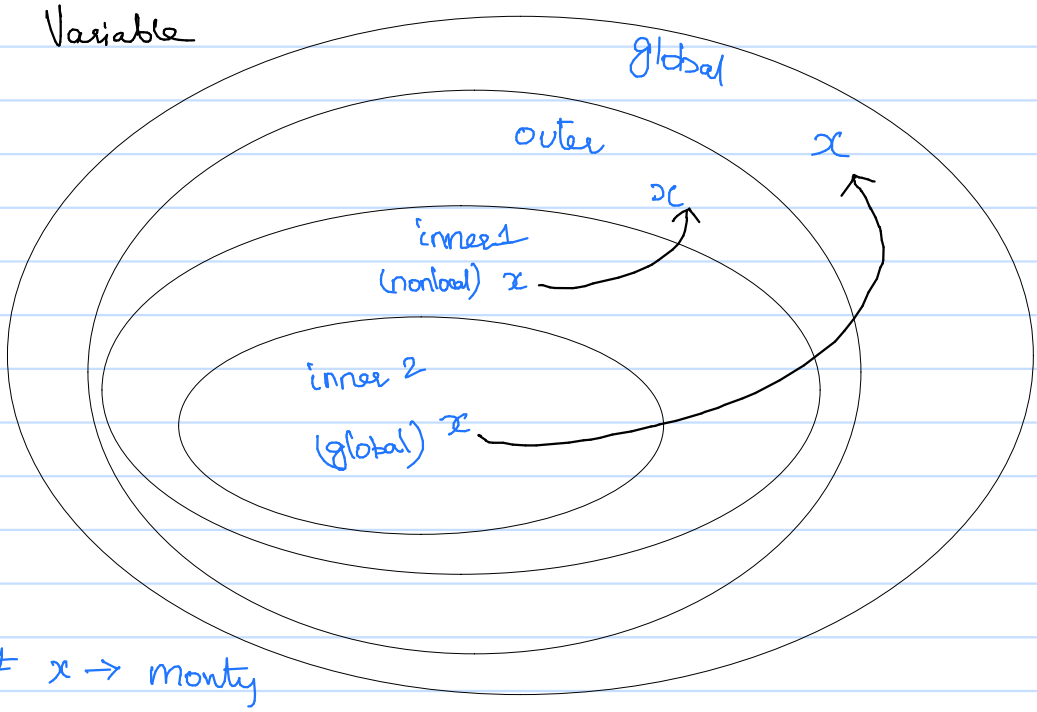
Beware: 'it will only look in local scopes, 'it will NOT look in the global scope

```
def outer():
    x = "hello"

    def inner1():
        def inner2():
            nonlocal x
            x = 'python'
        inner2()
    inner1()
    print(x)    // x -> python
```



Non local & Global Variable

```
x = 10
def outer():
    x = 'python'

    def inner1():
        nonlocal x
        x = 'monty'
        def inner2():
            global x
            x = 'hello'
        print(x)    # x -> monty
        inner2()
        print(x)    # x -> monty

    inner1()
    print(x)    # x -> monty

outer()
print(x)    # x -> hello
```



Closures 😎

Suppose we need to count the no. of times a user clicked a button on a webpage (tracking OnClick)

Approach 1:

```
count = 0
def updateOnClick()
    count += 1
```

But the pitfall is that any script on the page can change the counter without calling updateOnClick

Approach 2:

```
def updateOnClick()
    count = 0
    count += 1
```

Oops! Everytime we call updateOnClick, count becomes 0 & then 1.

Approach 3:

```
def countWrapper():
    count = 0
    def updateOnClick()
        count += 1
    updateOnClick()
    return count
```

This approach could have worked, ONLY if we could reach updateOnClick function from the outside & we also need to find a way to execute count = 0 only once & not every time.

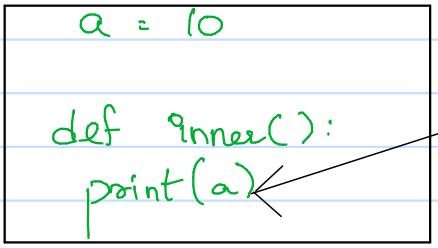
😑😑😑

Comes to the rescue, Closures 😎😎😎

Free Variables & Closures

Remember: Functions defined inside another function can access the outer (nonlocal) variables.

```
def outer():
    a = 10

    def inner():
        print(a)
```



This 'a' refers to one in outer's scope.

This nonlocal variable 'a' is called free variable.

inner()

When we consider inner, we really are looking at:
→ the function inner
→ the free variable 'a' (with value 10)

```
outer() # python
```

This is called Closure.

Returning the inner Function

What happens if, instead of calling (running) inner from inside the outer we return it?

```
def outer():
    x = 'python'

    def inner():
        print(x)

    inner()
    return inner
```

We can assign that return value to a variable name

```
fn = outer()
fn()  # python
```

When we called fn at that time Python determines the value of 'x' in extended scope. But notice the outer had finished running before we call fn - its scope was "gone"

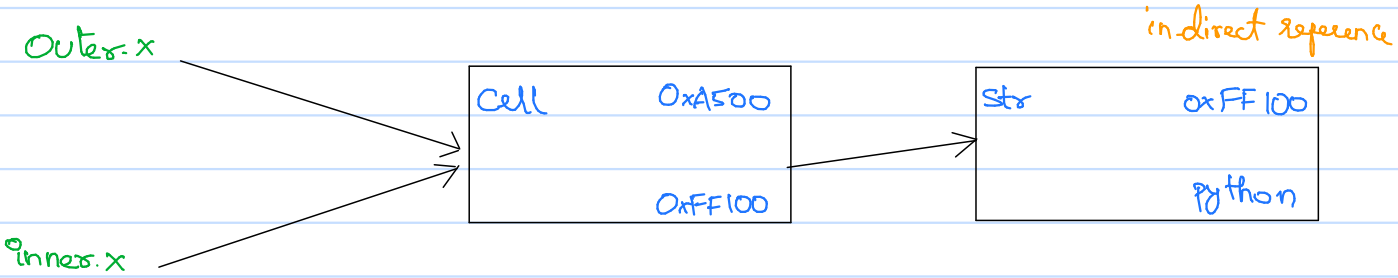
Python cells & Multi-Scope Variables:

```
def outer():
    x = 'python'
    def inner():
        print(x)
    return inner
```

Here the variable x is shared between 2 scopes
- outer
- closure

The label x is in two different scopes but always refer to same "value"

Python does this by creating a cell as an "intermediary object.



In effect, both variables x (in outer & inner) point to the same cell.

When requesting the value of the variable, Python will "double-hop" to get the final value.

Closures:

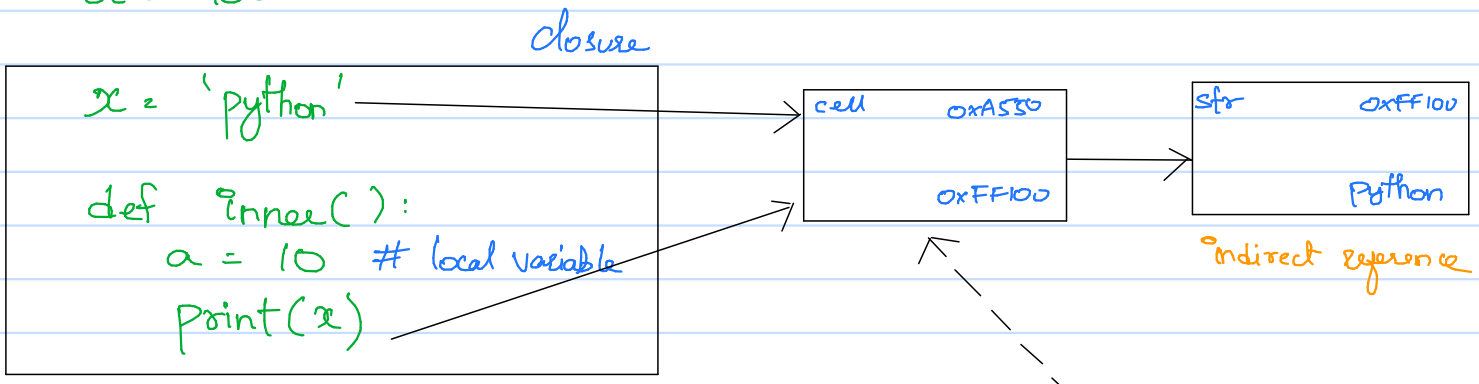
You can think of closures as a function plus an extended scope that contains the free variables.

The free variable's value is the object the cell points to - so that could change over time!

Every time the function in the closure is called and the free variable is referenced:

Python looks up the cell object, and then whatever the cell is pointing to.

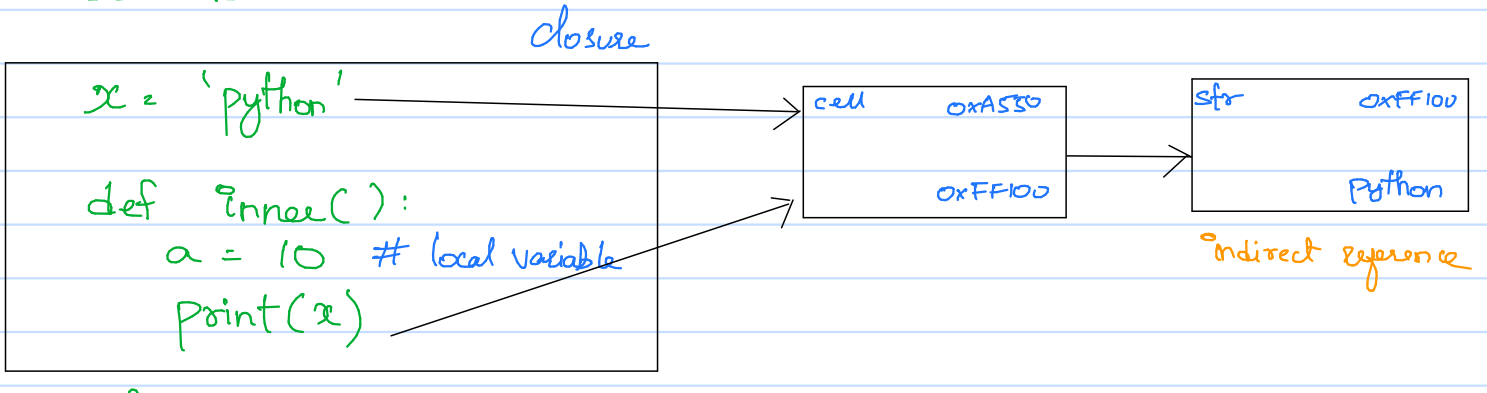
```
def outer():

    a = 100
```




```
fn = outer()    # fn → inner + extended scope.
```

Introspection:

```
def outer():

    a = 100
```




```
fn = outer()    # fn → inner + extended scope.

fn.__code__.co_freevars    # ('x',)  (a is NOT a free var)

fn.__closure__    # (<cell at 0xA550: str object at 0xFF100>)
```

Multiple Instances of Closure

Every time we run a function, a new scope is created.

If that function generates a closure, a new closure is created every time as well.

```
def counter():
    count = 0

    def inc():
        nonlocal count
        count += 1
        return count

    return inc

f1 = counter()
f2 = counter()
```

```
f1()   # → 1
f1()   # → 2
f1()   # → 3
f2()   # → 1
```

f1 & f2 doesn't have the same extended scope.

they are different instances of the closure.

the cells are different

Shared Extended Scope:

```
def outer():
    count = 0

    def inc1():
        nonlocal count
        count += 1
        return count

    def inc2():
        nonlocal count
        count += 1
        return count

    return inc1, inc2
```

count is a free variable bound to extended scope

This count is also a free variable bounded to same extended scope count

returns a tuple containing both closure.

```
f1, f2 = outer
f1()   # → 1
f2()   # → 2.
```

It may feel that, this shared scope is highly unusual..... but it's not.

```
def adder(n):
    def inner(x)
        return x+n
    return inner

add_1 = adder(1)
add_2 = adder(2)
add_3 = adder(3)
```

Three different closures
no shared scope

```
add_1(10)     # 11
add_2(100)    # 102
add_3(1000)   # 1003
```

But suppose we tried doing it this way

```
adders = []
for n in range(1,4):
    adders.append(lambda x: x+n)
```

n = 1 : free variable in the lambda is n, and it is bound to n we created in the loop.
n = 2 : free variable in the lambda is n, and it is bound to (same) n we created in the loop.
n = 3 : free variable in the lambda is n, and it is bound to (same) n we created in the loop.

Now we can call adders in this way

```
adders[0](10)     # → 13     (expected 11)
adders[1](100)    # → 103    (expected 12)
adders[2](1000)   # → 1003   (expected 1003)
```

Rember, Python doesn't evaluate the free variable n until the adders[i] function is called.
Since all the function in adders are bound to same n

by the time we call address[0], the value of n is 3
Since calling is after loop, the last assigned value in loop is 3.

Nested Closures:

```
def increment(n):
    # inner + n -> closure
    def inner(start)
        current = start
        # inc + current+n -> closure
        def inc( )
          nonlocal current
          current += n
          return current
    return inc

return inner
```

```
fn = increment(2)   # fn <- inner      fn.__code__.co_freevars -> 'n'
inc_2 = fn(100)     # inc_2 <- inc     inc_2.__code__.co_freevars -> 'current','n'

inc_2   # -> 102
inc_2   # -> 104
```

DECORATORS

Recall that simple closure example we did which allowed us to maintain a count of how many times a function was called?

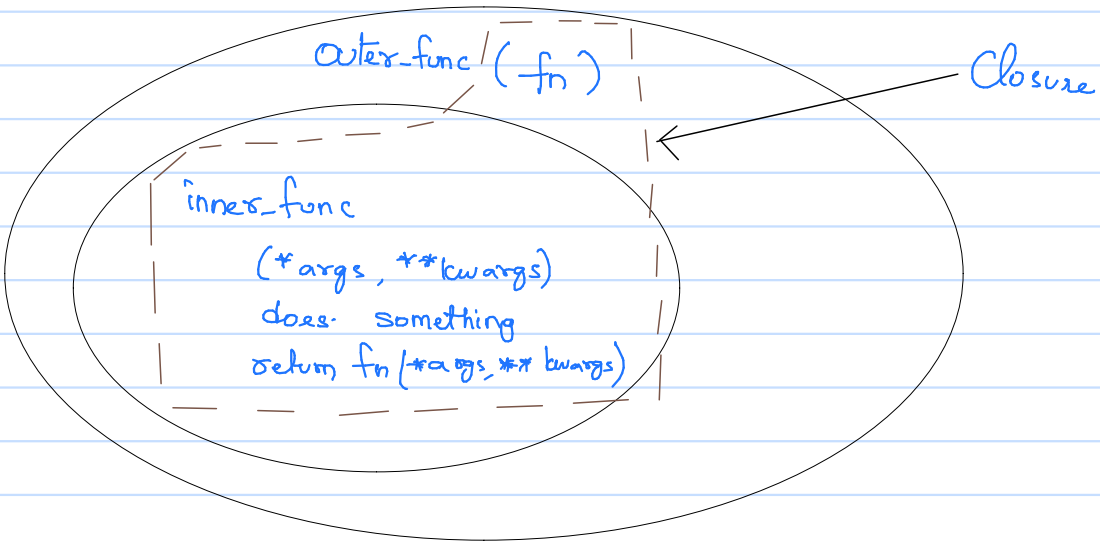
```
def count(fn):
    count = 0
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print(f'{fn.__name__} executed {count} times')
        return fn(*args, **kwargs)
    return inner

def add(a, b)
    return a+b

add = count(add)
result = add(1, 2)
```

We essentially modified our add function by wrapping it inside another function that added some functionality to it.

We also say that we decorated our function add with the function count, and we call count a decorator function

In general a decorator function:

- takes a function as an argument.
- returns a closure.
- the closure usually accepts any combination of parameters
- runs some code in the inner function (closure)
- the closure function calls the original function using the arguments passed to the closure.
  (original function → fn → free variable)
- returns whatever is returned by the function call.



Decorators and the @ Symbol:

In our previous example, we saw that count was a decorator and we could decorate our add function using:

```
add = count(add)
```

In general, if func is a decorator function, we decorate another function (my_func) using

```
my_func = func(my_func)
```

But we can also do

```
@count
def add(a, b):
    return a+b
```

which is exactly same as

```
def add(a, b):
    return a+b

add = count(add)
```

Introspecting Decorated functions:

```
def count(fn):
    count = 0
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print(f'{fn.__name__} executed {count} times')
        return fn(*args, **kwargs)
    return inner

@counter
def mul(a,b,c):
    '''
    multipty 3 numbers.
    '''
    return a*b*c
```

```
mult.__name__   # → 'inner  not mul.
```

mul name has been "changed" when decorated.

```
help(mult)  # Help on function inner in module __main__:
            inner(*args, **kwargs)
```

We have also "lost" docstrings & even original fn signature. Even using 'inspect' module's signature doesn't yield better results.

The functools.wrap function

The functools module has a wraps function that we can use to fix the metadata of our inner function in our decorator.

```
from functools import wraps
```

Infact, the wrap function itself is a decorator. but it needs to know, what was our "original" function - in this case fn

```
def count(fn):
    count = 0
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print(f'{fn.__name__} executed {count} times')
        return fn(*args, **kwargs)
    inner = wraps(fn)(inner)
    return inner
```

Same.

```
def count(fn):
    count = 0
    @wraps(fn)
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print(f'{fn.__name__} executed {count} times')
        return fn(*args, **kwargs)
    return inner
```

Now

```
@counter
def mul(a, b, c)
    '''
    multiplies 3 numbers
    '''
    return a * b * c

help(mul)  #  mul(a, b, c)
              multiplies 3 numbers
```

And introspecting using the inspect module works as expected

```
inspect.signature(mul)  #  <Signature (a, b, c)>
```

Parameterized Decorators

Decorator Parameters:

Function Call

```
@wraps(fn)
def inner():
    ....
```

```
@lru_cache(maxsize = 256)
def factorial():
    ....
```

This should look quite different from the decorators we have been creating & using.

```
@timed
def fibonacci():
    .....
```

← no function call

The timed decorator

```
def timed(fn):
    from time import perf_counter

    def inner(*args, **kwargs):
        time_elapsed = 0
        for i in range(10):
            start = perf_counter()
            result = fn(*args, **kwargs)
            end = perf_counter()
```

hard coded value

```
@timed
def my_func():
```

OR

```
my_func = timed(my_func)
```

How can we pass the value instead of hard coding.

One Approach:

extra parameter

```
def timed(fn, reps)
    from time import perf_counter
    def inner(*args, **kwargs):
        for i in range(reps):
            ....
        ....
    return inner
```

free variable

my_func = timed(my_func, 10) ✓

@timed(10) ✗ This will throw an error.
def my_func():
...

Need Closure to Rescue:

```
def timed(reps):
    def dec(fn):
        from time import perf_counter

        @wraps(fn)
        def inner(*args, **kwargs):
            total_elapsed = 0
            for i in range(reps):
                start = perf_counter()
                result = fn(*args, **kwargs)
                end = perf_counter()
            print((end-start)/reps)
            return result
        return inner
    return dec
```

free variable bound to reps in timed

calling timed(n) returns our original decorator with reps set to n (free var)

timed(n) is not a decorator, it is something we call DECORATOR FACTORY

```
@timed(10)
def my_func():
...
```
✓

my_func = outer(10)(my_func) ✓

We call timed as decorator factory, since it is a function that creates a new decorator each time it is called

We can also created decorator factories using Classes by implementing __call__ method which returns decorators.
(Check Session_6's assignment for sample implementation)

Extra Points

1. Closure outer when called defined the free var and 'inner.
   The value is assigned only when the 'inner is called.

```
def outer():
    n = 0
    def inner():
        return 10/n
    return inner

fn = outer()      # No Error
res = fn()        # Error. (can't divide by Zero)
```

2. Memoization

```
def memoize(fn):
    cache = dict()

    @wraps(fn)
    def inner(*args):
        if args not in cache:
            cache[args] = fn(*args)
        return cache[args]

    return inner
```

This is so much uses that Python has 'inbuilt memoization function → lru_cache() (last recently used cache)

```
from functools import lru_cache
```

The lru_cache when decorates a function, it remembers what it returned for a set of arguments & if same is provided, it doesn't run function & pass the return value.

```
@lru_cache()
def print_my_name(n):
    print("I was Executed")
    if n == "rohan":
        return 1
    else:
        return 0
```

```
print_my_name("rohan")  # I was Executed. 1
print_my_name("Mohan")  # I was Executed. 0
print_my_name("Mohan")  # 0
print_my_name("rohan")  # 1
```

{ Since "Mohan" & "rohan" was already passed, it didn't execute & just returned the values which was previously returned }

3. Single Dispatch

```
def singledispatch(fn):

    registry = dict()
    registry[object] = fn
    registry[int] = lambda arg: '{0}'.format(arg)
    registy[float] = lambda arg: '{0}'.format(round(arg,2))

    def inner(arg):
        fn = registry.get(type(arg), registry[object])
        return fn(arg)

    return inner
```

```
@singledispatch
def htmlize(a):
    return escape(str(a))
```

Single dispatch is used as Switch Case in this case.