CLOSURES

Free Variables and Closures

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

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

    def inner():
        print("{0} rocks!".format(x))

    inner()


outer()        → python rocks!
```

this x refers to the one in outer's scope

this nonlocal variable x is called a free variable

when we consider inner, we really are looking at:

- the function inner
- the free variable x (with current value python)

This is called a closure

Returning the inner function

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

```
def outer():

    x = 'python'

    def inner():
        print("{0} rocks!".format(x))

    ~~inner()~~
    return inner
```

`x` is a free variable in `inner`

it is bound to the variable `x` in `outer`

this happens when `outer` runs

(i.e. when `inner` is created)

this the closure

when we return `inner`, we are actually "returning" the closure

We can assign that return value to a variable name: `fn = outer()`

`fn()` → `python rocks!`

When we called `fn`

at that time Python determined the value of `x` in the extended scope

But notice that `outer` had finished running before we called `fn` – it's scope was "gone"

Python Cells and Multi-Scoped Variables

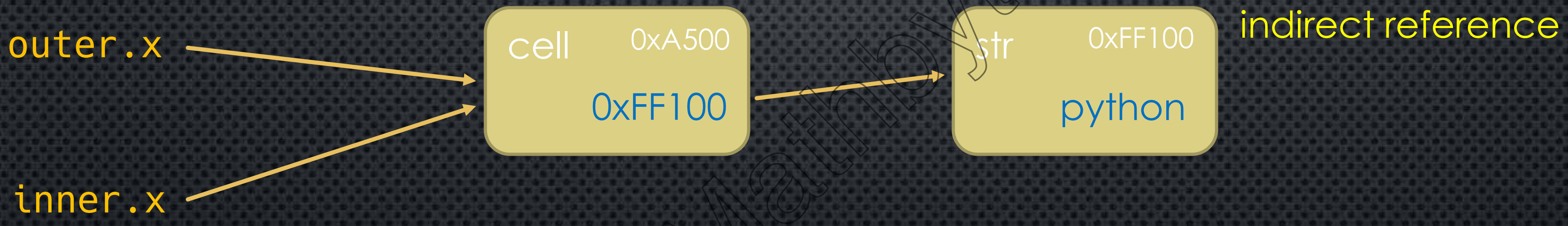
```
def outer():
    x = 'python'
    def inner():
        print(x)
    return inner
```

Here the value of x is shared between two scopes:

- outer
- closure

The label x is in two different scopes but always reference the same "value"

Python does this by creating a cell as an intermediary object



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

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

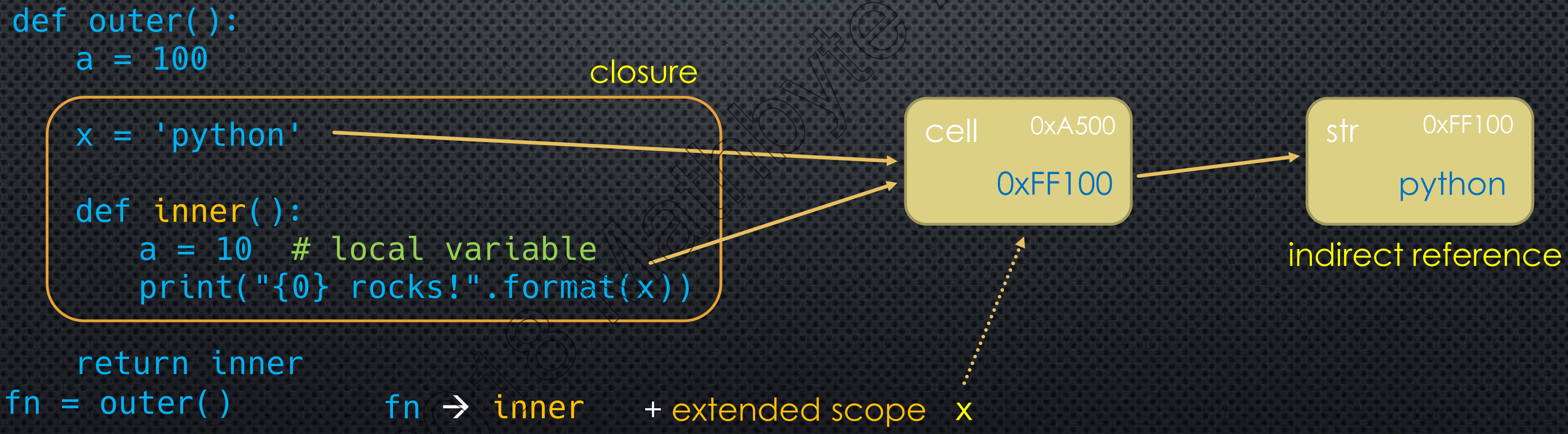
Closures

You can think of the closure 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

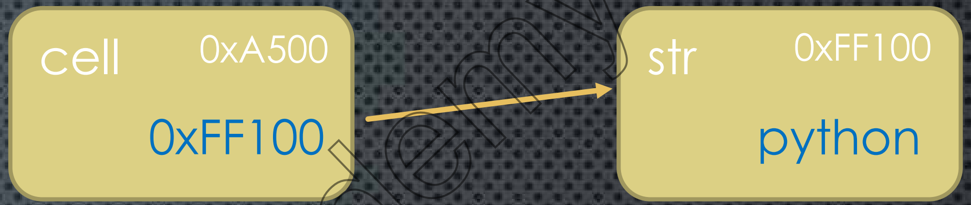


```
def outer():
    a = 100

    x = 'python'

    def inner():
        a = 10  # local variable
        print("{0} rocks!".format(x))

    return inner
fn = outer()
```

fn → inner + extended scope x

Introspection

```
def outer():
    a = 100
    x = 'python'
    def inner():
        a = 10   # local variable
        print("{0} rocks!".format(x))
    return inner

fn = outer()
```



indirect reference

`fn.__code__.co_freevars` → ('x',)   (a is not a free variable)

`fn.__closure__` → `(<cell at 0xA500: str object at 0xFF100>, )`

```
def outer():
    x = 'python'
    print(hex(id(x)))
    def inner():
        print(hex(id(x)))
        print("{0} rocks!".format(x))
    return inner

fn = outer()
fn()
```

→ `0xFF100` indirect reference

→ `0xFF100` indirect reference

Modifying free variables

closure

```
def counter():
    count = 0

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

    return inc
```

count is a free variable

it is bound to the cell count

fn → inc + count → 0

```
fn = counter()
```

fn() → 1   count's (indirect) reference changed from the object 0 to the object 1

fn() → 2

Multiple Instances of Closures

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

closure

```
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 and f2 do not have the same extended scope

they are different instances of the closure

the cells are different

Shared Extended Scopes

```
def outer():
    count = 0
    def inc1():
        nonlocal count
        count += 1
        return count
    def inc2():
        nonlocal count
        count += 1
        return count
    return inc1, inc2

f1, f2 = outer()

f1() → 1
f2() → 2
```

count is a free variable – bound to count in the extended scope

count is a free variable – bound to the same count

returns a tuple containing both closures

Shared Extended Scopes

You may think this shared extended 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 scopes

```
add_1(10)    → 11
add_2(10)    → 12
add_3(10)    → 13
```

Shared Extended Scopes

But suppose we tried doing it this way:

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

`n = 1:` the free variable in the lambda is n, and it is bound to the n we created in the loop

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

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

Now we could call the adders in the following way:

```
adders[0](10)    → 13
adders[1](10)    → 13
adders[2](10)    → 13
```

Remember, Python does not "evaluate" the free variable n until the `adders[i]` function is called

Since all three functions in `adders` are bound to the same n

by the time we call `adders[0]`, the value of n is 3
(the last iteration of the loop set n to 3)

Nested Closures

```
def incrementer(n):
    # inner + n is a closure
    def inner(start):
        current = start
        # inc + current + n is a closure
        def inc():
            nonlocal current
            current += n
            return current

        return inc
    return inner
```

```
(inner)
fn = incrementer(2)  → fn.__code__.co_freevars → 'n'  n=2
(inc)
inc_2 = fn(100)   → inc_2.__code__.co_freevars → 'current', 'n'
                                                  current=100, n=2
(calls inc)
inc_2()         → 102   (current = 102, n=2)
inc_2()         → 104   (current = 104, n=2)
```

Code