GLOBAL AND LOCAL SCOPES

Scopes and Namespaces

When an object is assigned to a variable `a = 10`

that variable points to some object

and we say that the variable (name) is bound to that object

That 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 that 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 file only.

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

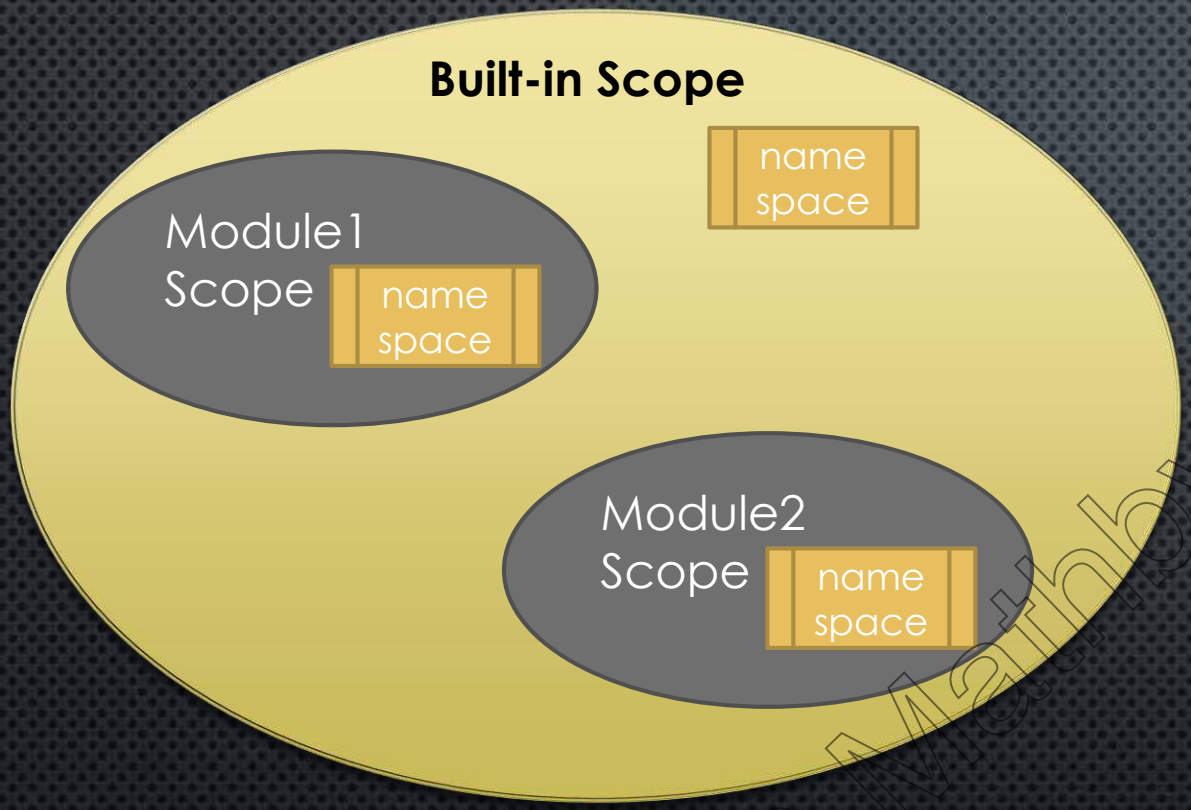
The only exception to this are some of the built-in globally available objects, such as:

```
True   False   None   dict   print
```

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

including inside any function

Global scopes are nested inside the built-in scope



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

Examples

module1.py

```
print(True)
```

Python does not find `True` or `print` in the current (module/global) scope

So, it looks for them in the enclosing scope → built-in

Finds them there → `True`

module2.py

```
print(a)
```

Python does not find `a` or `print` in the current (module/global) scope

So, it looks for them in the enclosing scope → built-in

Find `print`, but not `a` → `run-time Name Error`

module3.py

```
print = lambda x: 'hello {0}!'.format(x)

s = print('world')
```

Python finds `print` in the module scope

So it uses it!

`s` → `hello world!`

The Local Scope

When we create functions, we can create variable names inside those functions (using assignments)

e.g. `a = 10`

Variables defined inside a function are not created until the function is called

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

Variables defined inside the function are assigned to that scope → Function Local scope

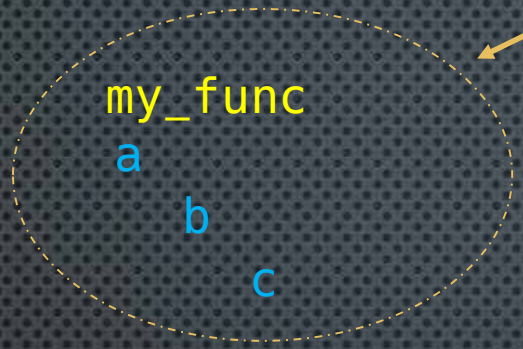
→ Local scope

The actual object the variable references could be different each time the function is called

(this is why recursion works!)

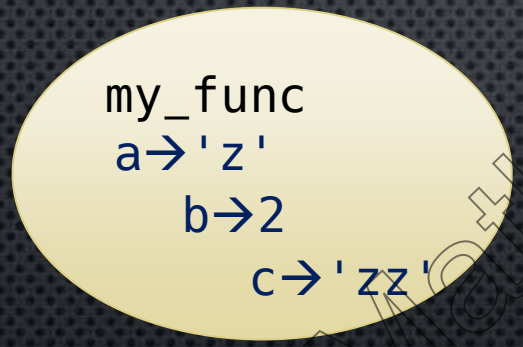
Example

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



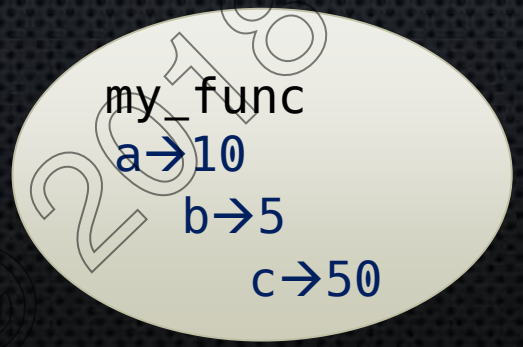
these names will considered local to my_func

```
my_func('z', 2)
```



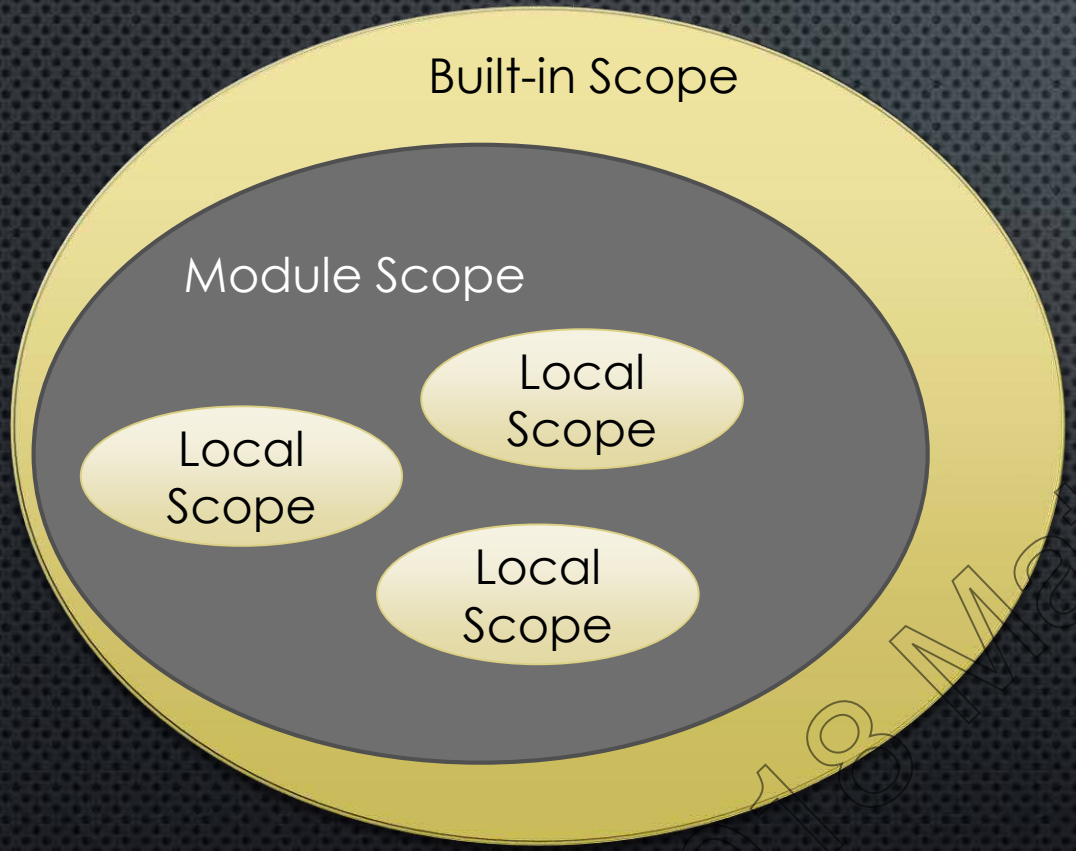
same names, different local scopes

```
my_func(10, 5)
```



Nested Scopes

Scopes are often nested



Namespace lookups

When requesting the object bound to a variable name:

e.g. `print(a)`

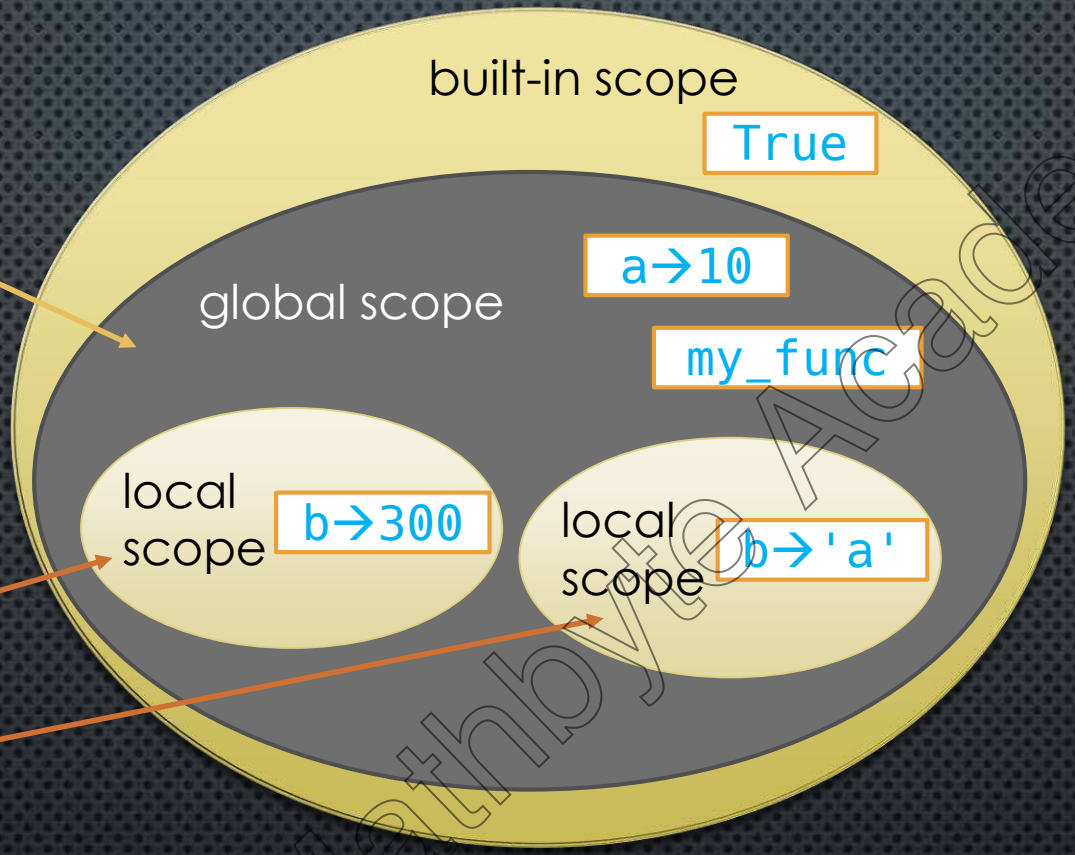
Python will try to find the object bound to the variable

- in current local scope first
- works up the chain of enclosing scopes

Example

module1.py

```
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, the scope is gone too!

and the reference count of the object `var` was bound to (referenced) is decremented

We also say that `var` goes out of scope

Accessing the global scope from a local scope

When retrieving the value of a global variable from inside a function, Python automatically searches the local scope's namespace, and up the chain of all enclosing scope namespaces

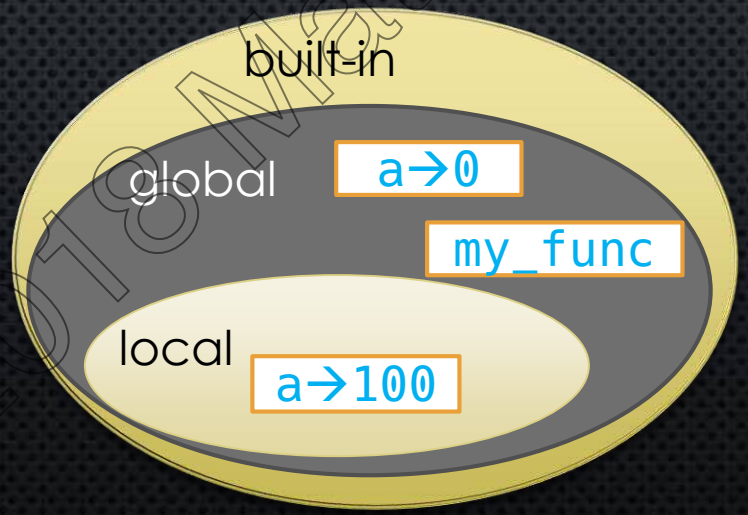
local → global → built-in

What about modifying a global variables value from inside the function?

```
a = 0

def my_func():
    a = 100
    print(a)

my_func()   → 100

print(a)  → 0
```

assignment → Python interprets this as a local variable (at compile-time)

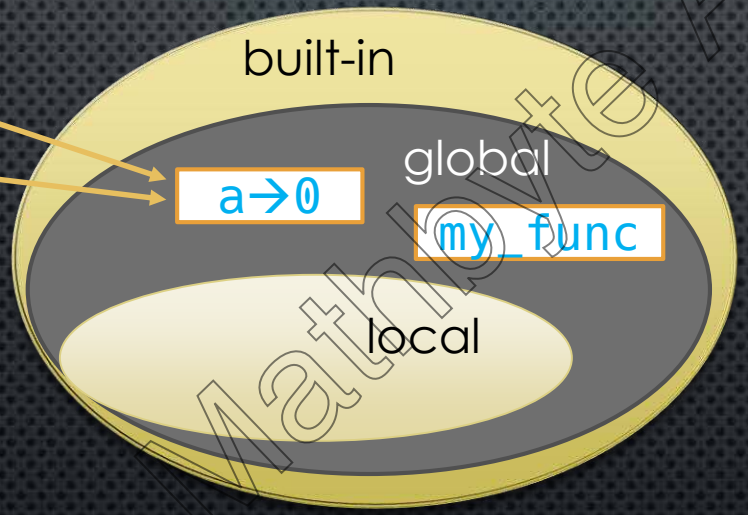
→ the local variable a masks the global variable a



The `global` keyword

We can tell Python that a variable is meant to be scoped in the global scope by using the `global` keyword

```
a = 0

def my_func():
    global a
    a = 100

my_func()

print(a)  → 100
```



Example

```
counter = 0

def increment():
    global counter
    counter += 1

increment()
increment()
increment()

print(counter)   → 3
```

Global and Local Scoping

When Python encounters a function definition at compile-time

it will scan for any labels (variables) that have values assigned to them (anywhere in the function)

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 referenced only 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 → a global (because we told Python a was global)

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

assignment at compile time → a local

→ when we call `func4()` `print(a)` results in a run-time error

because a is local, and we are referencing it before we have assigned a value to it!

Code