INTERMEDIATE PYTHON

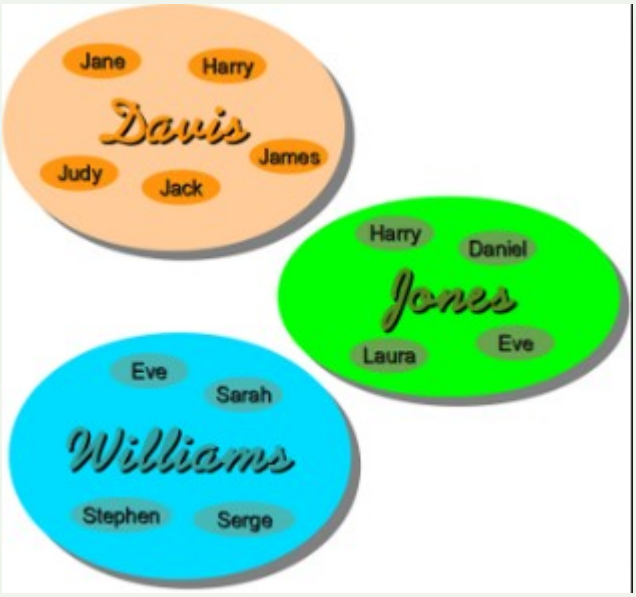
LESSON 1 |Namespace & Variable Scope | 03-12-18

A complete video tutorial on understanding variable scopes and namespace in python :

https://www.youtube.com/watch?v=Dnm2IWh_kxE&t=20s&list=PLqEbL1vopgvsQI9nLmaIhKV9qGOi1PYPn&index=16

Namespaces

A **namespace** is a naming system for making names unique to avoid ambiguity. Many programming languages use namespaces for identifiers. An identifier defined in a namespace is associated with that namespace. This way, the same identifier can be independently defined in multiple namespaces. (Like the same file names in different directories)

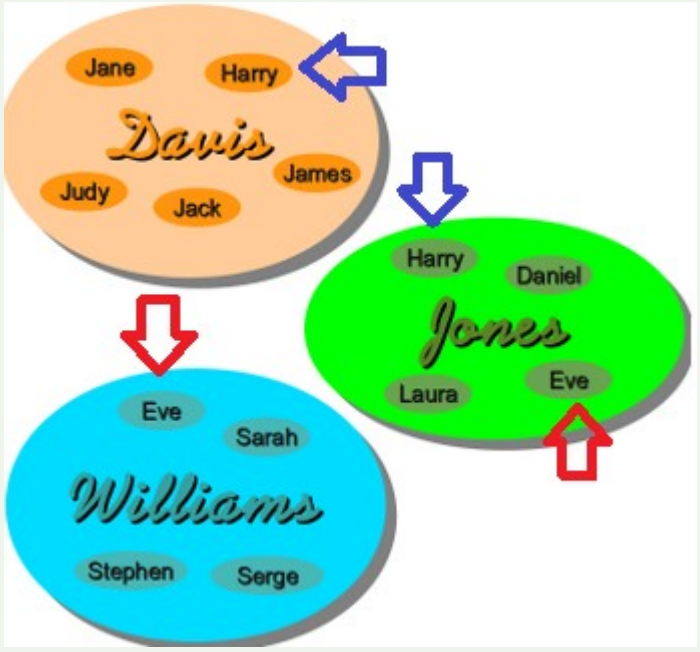


Davis , Jones and williams are namespaces

Jane, Harry, judy ,jack and james are identifiers *defined in the namespace Davis*

Harry, daniel, Laura and Eve are identifiers *defined in the namesoace Jones*

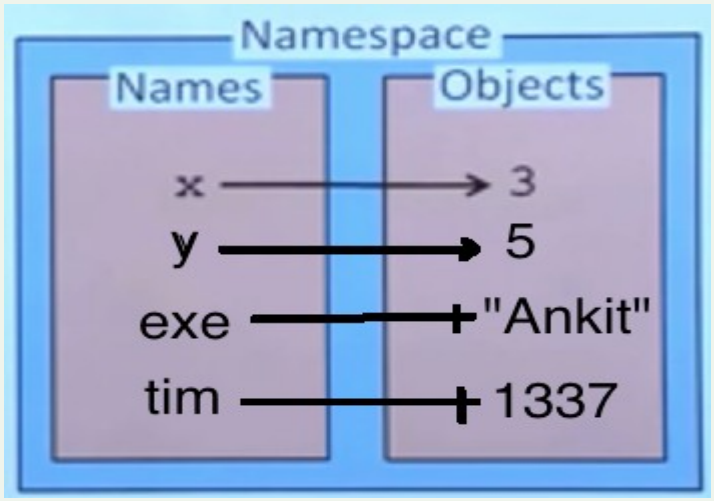
Eve, Sarah, Stephen and Serge are identifiers *defined in the namespace Wiliams*



The identifier *Eve is independently defined in namespaces Jones and Wiliams*

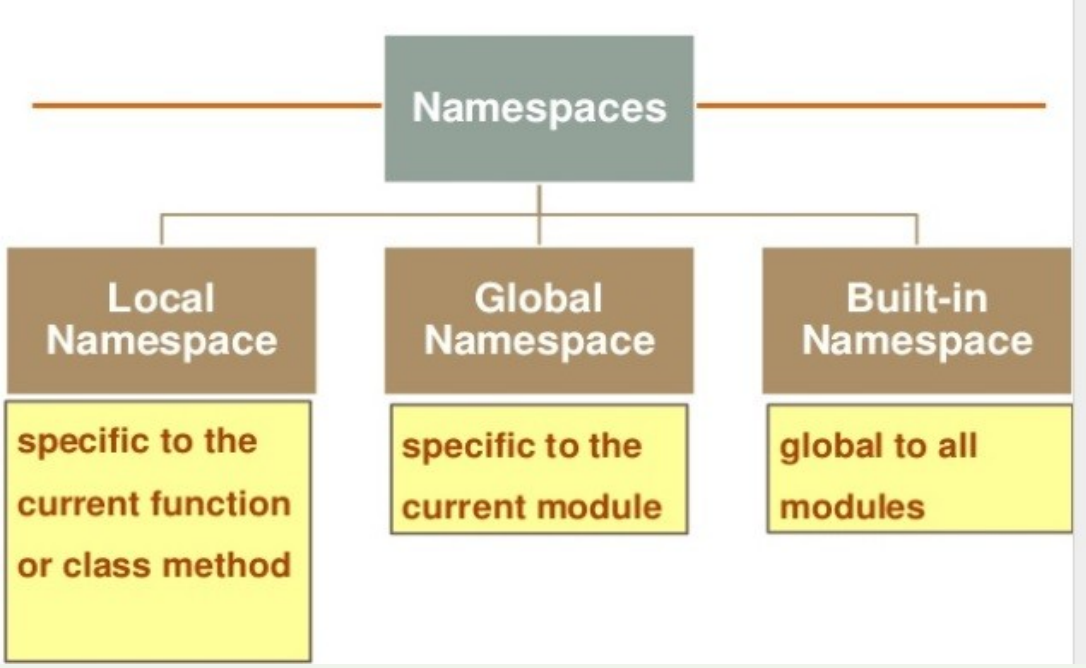
The identifier *Harry is independently defined in namespaces Jones and Davis*

Namespaces in Python are implemented as Python dictionaries, this means it is a mapping from names (keys) to objects (values).



Programming languages, which support namespaces, may have different rules that determine to which namespace a variable belongs.

Some namespaces in Python:



Not every namespace, which may be used in a script or program is accessible at any moment during the execution of the script.

Namespaces have different lifetimes, because they are often created at different points in time. There is one namespace which is present from beginning to end: The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace of a module is generated when the module is read in. Module namespaces normally last until the script ends, i.e. the interpreter quits. When a function is called, a local namespace is created for this function. This namespace is deleted either if the function ends, i.e. returns, or if the function raises an exception, which is not dealt with within the function.

Variable Scope - Understanding the LEGB rule and global/nonlocal statements

In this tutorial, you will learn about Python's scope of variables, the global and nonlocal keywords, closures and the LEGB rule.

If you're familiar with Python, or any other programming language, you'll certainly know that variables need to be defined before they can be used in your program. Depending on how and where it was defined, a variable will have to be accessed in different ways. Some variables are defined globally, others locally. This means that a variable referring to an entity in a certain part of a program, may refer to something different in another part of the program.

What are variables, really?

To understand the scope of variables, it is important to first learn about what variables really are. Essentially, they're references, or pointers, to an object in memory. When you assign a variable with `=` to an instance, you're binding (or mapping) the variable to that instance. Multiple variables can be bound to the same instance.

```
i = 5
j = i
i = 3

print("i: " + str(i))
print("j: " + str(j))

i: 3
j: 5
```

A namespace for the code above could look like `{i:3, j:5}`, and not `{i:3, j:i}` like you might expect.

The difference between defining a variable inside or outside a Python function

If you define a variable at the top of your script, it will be a global variable. This means that it is accessible from anywhere in your script, including from within a function. Take a look at the following example where a is defined globally.

```
a = 5

def function():
    print(a)

function()

print(a)
```

```
5
5
```

In the next example, a is defined globally as 5, but it's defined again as 3, within a function. If you print the value of a from within the function, the value that was defined locally will be printed. If you print a outside of the function, its globally defined value will be printed. The a defined in function() is literally sealed off from the outside world. It can only be accessed locally, from within the same function. So the two a's are different, depending on where you access them from.

```
a = 5

def function():
    a = 3
    print(a)

function()

print(a)
```

```
3
5
```

This is all good and well, but what are the implications of this?

Well, let's say you have an application that remembers a name, which can also be changed with a change_name() function. The name variable is defined globally, and locally within the function. As you can see, the function fails to change the global variable.

```
name = 'Théo'

def change_name(new_name):
    name = new_name

print(name)

change_name('Karlijn')

print(name)
```

```
Théo
Théo
```

Luckily, the global keyword can help.

The global keyword

With `global`, you're telling Python to use the globally defined variable instead of locally defining it. To use it, simply type `global`, followed by the variable name. In this case, the global variable `name` can now be changed by `change_name()`.

```
name = 'Théo'

def change_name(new_name):
    global name
    name = new_name

print(name)

change_name('Karlijn')

print(name)
```

```
Théo
Karlijn
```

The nonlocal keyword

The nonlocal statement is useful in nested functions. It causes the variable to refer to the previously bound variable in the closest enclosing scope. In other words, it will prevent the variable from trying to bind locally first, and force it to go a level 'higher up'.

Take a look at the three code examples below. In the first one, `inner()` binds `x` to `"c"`, `outer()` binds it to `"b"` and `x` is globally defined as `"a"`. Depending on where the variable is accessed from, a different binding will be returned.

```
x = "a"
def outer():
    x = "b"
    def inner():
        x = "c"
        print("from inner:", x)

    inner()
    print("from outer:", x)

outer()
print("globally:", x)
```

```
from inner: c
from outer: b
globally: a
```

With the nonlocal keyword, you're telling python that the x in the inner() function should actually refer to the x defined in the outer() function, which is one level higher. As you can see from the result, x in both inner() and outer() is defined as "c", because it could be accessed by inner().

```
x = "a"
def outer():
    x = "b"
    def inner():
        nonlocal x
        x = "c"
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)
```

```
inner: c
outer: c
global: a
```

If you use global, however, the x in inner() will refer to the global variable. That one will be changed, but not the one in outer(), since you're only referring to the global x. You're essentially telling Python to immediately go to the global scope.

```
x = "a"
def outer():
    x = "b"
    def inner():
        global x
        x = "c"
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)
```

```
inner: c
outer: b
global: c
```

Closures in Python

Closures are function objects that remember values in enclosing scopes, even if they are no longer present in memory.

Remember that a nested function is a function defined in another function, like inner() is defined inside outer() in the example below. Nested functions can access variables of the enclosing scope, but can't modify them, unless you're using nonlocal. In the first example, you'll see that the number = 3 assignment only holds within inner().

```
def outer(number):
    def inner():
        number = 3
        print("Inner: " + str(number))
    inner()
    print("Outer: " + str(number))

outer(9)
```

```
Inner: 3
Outer: 9
```

If you add nonlocal number in inner(), however, the enclosing scope variable from outer() will be changed by inner(). number = 3 now applies to number in inner() and outer().

```
def outer(number):
    def inner():
        nonlocal number
        number = 3
        print("Inner: " + str(number))
    inner()
    print("Outer: " + str(number))

outer(9)
```

```
Inner: 3
Outer: 3
```

You may want to preserve a variable defined in a nested function, without having to change a global varibale.

In Python, a function is also considered a object, which means that it can be returned and assigned to a variable. In the next example, you'll see that instead of inner() being called inside outer(), return inner is used. Then, outer() is called with a string argument and assigned to closure. Now, even though the functions inner() and outer() have finished executing, their message is still preserved. By calling closure(), the message can be printed.

```
def outer(message):
    # enclosing function
    def inner():
        # nested function
        print(message)
    return inner

closure = outer("Hello world!")
closure()
```

```
Hello world!
```

Notice that if you call closure without parentheses, only the type of the object will be returned. You can see that it's of the type function __main__.outer.<locals>.inner.

```
closure
<function __main__.outer.<locals>.inner>
```

The LEGB rule

As you saw before, namespaces can exist independently from each other, and have certain levels of hierarchy, which we refer to as their scope. Depending on where you are in a program, a different namespace will be used. To determine in which order Python should access namespaces, you can use the LEGB rule.

LEGB stands for:

- Local
- Enclosed
- Global
- Built-in

Let's say you're calling `print(x)` within `inner()`, which is a function nested in `outer()`. Then Python will first look if `x` was defined locally in that `inner()`. If not, the variable defined in `outer()` will be used. This is the enclosing function. If it also wasn't defined there, the Python interpreter will go up another level, to the global scope. Above that you will only find the built-in scope, which contains special variables reserverd for Python itself.

```
# Global scope

x = 0

def outer():
    # Enclosed scope
    x = 1
    def inner():
        # Local scope
        x = 2
```

Conclusion

Awesome! You now know what Python's namespace and scope of variables is, how you should use the `global` and `nonlocal` keywords, and the LEGB rule. You'll be able to easily manipulate variables in nested functions, without any problem. This will be very useful in your future career as a Data Scientist.

Test Your Knowledge: Quiz

1. What is the output of the following code, and why?

```
>>> X = 'Spam'
>>> def func():
... print(X)
...
>>> func()
```

2. What is the output of this code, and why?

```
>>> X = 'Spam'
>>> def func():
... X = 'NI!'
...
>>> func()
>>> print(X)
```

3. What does this code print, and why?

```
>>> X = 'Spam'
>>> def func():
... X = 'NI'
... print(X)
...
>>> func()
>>> print(X)
```

4. What output does this code produce? Why?

```
>>> X = 'Spam'
>>> def func():
... global X
... X = 'NI'
...
>>> func()
>>> print(X)
```

5. What about this code—what's the output, and why?

```
>>> X = 'Spam'
>>> def func():
... X = 'NI'
... def nested():
... print(X)
... nested()
>>> func()
>>> X
```

6. How about this example: what is its output in Python 3.0, and why?

```
>>> def func():
... X = 'NI'
... def nested():
... nonlocal X
... X = 'Spam'
... nested()
... print(X)
>>> func()
```

7. Name three or more ways to retain state information in a Python function.

Test Your Knowledge: Answers

1. The output here is 'Spam', because the function references a global variable in the enclosing module (because it is not assigned in the function, it is considered global).

2. The output here is 'Spam' again because assigning the variable inside the functionmakes it a local and effectively hides the global of the same name. The print statementfinds the variable unchanged in the global (module) scope.

3. It prints 'NI' on one line and 'Spam' on another, because the reference to the variablewithin the function finds the assigned local and the reference in the print statement finds the global.

4. This time it just prints 'NI' because the global declaration forces the variable assigned inside the function to refer to the variable in the enclosing global scope.

5. The output in this case is again 'NI' on one line and 'Spam' on another, because the print statement in the nested function finds the name in the enclosing function's local scope, and the print at the end finds the variable in the global scope.

6. This example prints 'Spam', because the nonlocal statement (available in Python 3.0 but not 2.6) means that the assignment to X inside the nested function changes X in the enclosing function's local scope. Without this statement, this assignment would

classify X as local to the nested function, making it a different variable; the code would then print 'NI' instead.

7. Although the values of local variables go away when a function returns, you can make a Python function retain state information by using shared global variables, enclosing function scope references within nested functions, or using default argument values. Function attributes can sometimes allow state to be attached to the function itself, instead of looked up in scopes. Another alternative, using OOP with classes, sometimes supports state retention better than any of the scope-based techniques because it makes it explicit with attribute assignments

Python Namespaces and Scopes Practice Problems

1. Without running the code, predict the output of the following:

```
a,b,x,y,z = 13,5,2,8,16
def mystery(x,y):
    global a
    a = 35
    x,y = y,x
    b = 14
    b = 3
    c = 97
    print(a,b,x,y,z)

mestyry(12,18)
print(a,b,x,y,z)
```

2. Without running the code, predict the output of the following:

```
x = "2"
def example():
        x = "3"
        def method():
                global x
                x = "4"
                def function():
                        global x
                        x = "5"
                        print("Function Scope: " + x)
                function()
                print("Method Scope: " + x)
        method()
        print("Example Scope: " + x)
example()
print("Module Scope: " + x)
```

3. Without running the code, predict the output of the following:

```
x = "2"
def example():
        x = "3"
        def method():
                global x
                x = "4"
                def function():
                        nonlocal x
                        x = "5"
                        print("Function Scope: " + x)
                function()
                print("Method Scope: " + x)
        method()
        print("Example Scope: " + x)
example()
print("Module Scope: " + x)
```

4. Without running the code, predict the output of the following:

```
x = "2"  # x is now defined within the module namespace
def example():
        x = "3" # x is now defined as 3 within the local namespace of example
        def method():
                x = "4" # x is now defined as 4 within the local namespace of method
                def function():
                        x = "5" # x is now defined as 5 within the local namespace of function
                        print("Function Scope: " + x)
                function()
                print("Method Scope: " + x)
        method()
        print("Example Scope: " + x)
example()
print("Module Scope: " + x)
```

5. Without running the code, predict the output of the following:

```
x = "2"  # x is now defined within the module namespace
def example():
        x = "3" # x is now defined as 3 within the local namespace of example
        def method():
                global x  # x will now be defined as being within the module scope
                x = "4" # x is now defined as 4 within the local and module namespace
                def function():
                        x = "5" # x is now defined as 5 within the local namespace of function
                        print("Function Scope: " + x)
                function()
                print("Method Scope: " + x)
        method()
        print("Example Scope: " + x)
example()
print("Module Scope: " + x)
```

6. Without running the code, predict the output of the following:

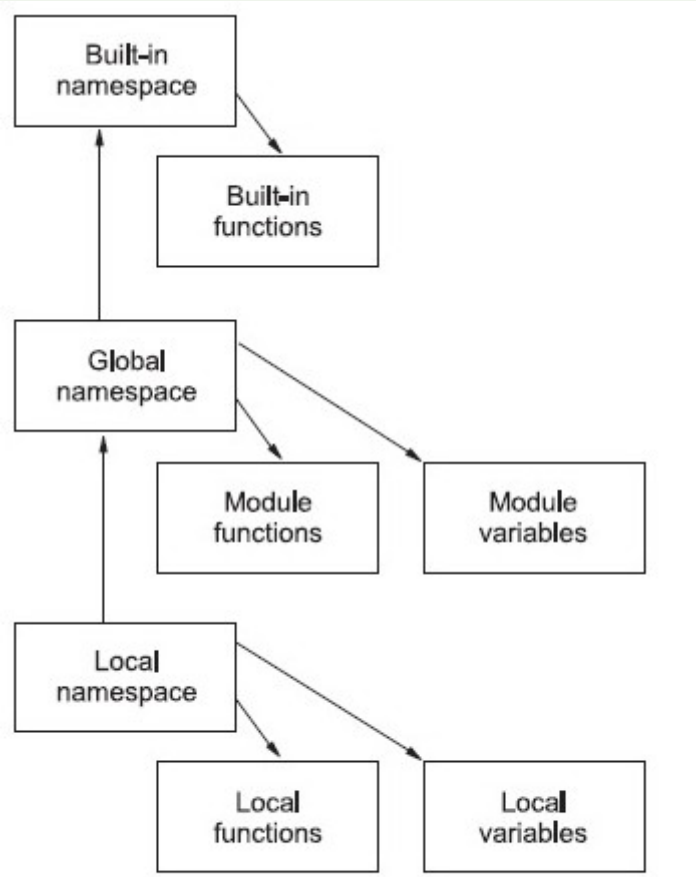
```
x = "2"  # x is now defined within the module namespace
def example():
        x = "3" # x is now defined as 3 within the local namespace of example
        def method():
                nonlocal x  # x will now be defined as being within the example scope
                x = "4" # x is now defined as 4 within the local and example namespace
                def function():
                        x = "5" # x is now defined as 5 within the local namespace of function
                        print("Function Scope: " + x)
                function()
                print("Method Scope: " + x)
        method()
        print("Example Scope: " + x)
example()
print("Module Scope: " + x)
```

7. Without running the code, predict the output of the following:

```
x = "2"  # x is now defined within the module namespace
def example():
        nonlocal x  # x will now be defined as being within the example scope
        x = "3" # x is now defined as 3 within the local namespace of example
```

Practice problems Answers

1. https://pastebin.com/XAvgJSAL
2. https://pastebin.com/1t7cxZdV
3. https://pastebin.com/ZvZjPdow
4. https://pastebin.com/EV8JyWjh
5. https://pastebin.com/KDtJKUXC
6. https://pastebin.com/2Ka1unxD
7. https://pastebin.com/7fHt7xsu



The order in which namespaces are checked to locate identifiers

note: ***I am also a beginner, now intermediate. The text and the questions are not conceived by me.***

Many thanks for the experienced programmer Lenin

Many thanks for admin

Lesson 2 : 10/11/12