PROGRAMACIÓN I

FUNCTIONS II

Business Analytics

FUNCTIONS II

- *Execution frames and namespaces*
- *Scope of variables*
- *Local and global keywords*

FUNCTIONS II

- *Execution frames and namespaces*
- *Scope of variables*
- *Local and global keywords*

Assignments and namespaces

An **assignment statement** creates a **symbolic name** that you can use to reference an object.

- The statement x = 'foo' creates a symbolic name x that refers to the string object 'foo'.

A **namespace** is a collection of currently defined symbolic names along with information about the object that each name references.

- You can think of a namespace as a dictionary:
  - keys are the object names
  - values are the objects themselves.

- Each key-value pair maps a name to its corresponding object.

Resolving

- When you do an assignment, like a = 1, you're changing/updating a namespace.
- When you make a reference, like print(a), Python looks through a list of namespaces to try and find one with the name as a key.

We are going to see the structure of namespaces and how a reference is resolved.

Namespaces

Local Namespace: local names inside a function. Created when a function is called, and it only lasts until the function returns.

Global Namespace:

- Names defined at the level of the main program
- includes names from various imported modules that you are using in a project. Created when the module is imported in the project, and it lasts until the script ends.

Built-in Namespace: built-in functions and built-in exception names.

When we have functions inside functions, from the perspective of the inner (enclosed) function, the namespace of the outer (enclosing function) is the **enclosing namespace**

https://docs.python.org/3/library/functions.html#dir returns names in the current **local** scope

FUNCTIONS II

- *Execution frames and namespaces*
- *Scope of variables*
- *Local and global keywords*

Scope

- A *scope* defines which namespaces will be looked in and in what order.
- The scope of any reference always starts in the local namespace, and moves outwards until it reaches the module's global namespace, before moving on to the built-ins (the namespace that references Python's predefined functions and constants, like range and print), which is the end of the line.

Resolving variables in the inner namespace

Variable resolved in the local namespace.

The scope of `a_variable` includes the `inner` namespace, so `print(a_variable)` works

```
A_CONSTANT = 3

def outer():
    def inner():
        a_variable = 7
        print(a_variable)

    another_variable = 2
    print(another_variable)
    inner()


outer()
print(A_CONSTANT)
```

```
A_CONSTANT = 3

def outer():
    def inner():
        print(another_variable)

    another_variable = 2
    print(another_variable)
    inner()


outer()
```

```
A_CONSTANT = 3

def outer():
    def inner():
        print(A_CONSTANT)

    another_variable = 2
    print(another_variable)
    inner()


outer()
```

Resolving variables enclosing namespace

Universidad Francisco de Vitoria

UFV Madrid

Variable resolved in the namespace of an enclosing function.

The scope of `another_variable` includes the namespace of `outer`, as enclosing function of `inner`, so `print(another_variable)` in `inner` works

```
A_CONSTANT = 3

def outer():
    def inner():
        a_variable = 7
        print(a_variable)

    another_variable = 2
    print(another_variable)
    inner()


outer()
print(A_CONSTANT)
```

```
A_CONSTANT = 3

def outer():
    def inner():
        print(another_variable)

    another_variable = 2
    print(another_variable)
    inner()


outer()
```

```
A_CONSTANT = 3

def outer():
    def inner():
        print(A_CONSTANT)

    another_variable = 2
    print(another_variable)
    inner()


outer()
```

Resolving variables in the global namespace

Universidad Francisco de Vitoria

UFV Madrid

Variable resolved in the global namespace (of the module where it is called).

The `A_CONSTANT` is defined in the global namespace, which is reachable by any function in the module, in particular the function of `inner`, so `print(A_CONSTANT)` in `inner` works

```
A_CONSTANT = 3

def outer():
    def inner():
        a_variable = 7
        print(a_variable)

    another_variable = 2
    print(another_variable)
    inner()


outer()
print(A_CONSTANT)
```

```
A_CONSTANT = 3

def outer():
    def inner():
        print(another_variable)

    another_variable = 2
    print(another_variable)
    inner()


outer()
```

```
A_CONSTANT = 3

def outer():
    def inner():
        print(A_CONSTANT)

    another_variable = 2
    print(another_variable)
    inner()


outer()
```

Resolving built-ins

The function print is built in, and can be used in any function in the model.

```
A_CONSTANT = 3

def outer():
    def inner():
        print(another_variable)

    another_variable = 2
    print(another_variable)
    inner()


outer()
```

Resolution, summary

Whenever you define a function, you create a new namespace and a new scope.

- The namespace is the new, local hash of names.
- The scope is the implied chain of namespaces that starts at the new namespace, then works its way through any outer namespaces (outer scopes), up to the global namespace (the global scope), and on to the built-ins.

Scope, summary

Scope: defines the parts of the program where names can be used without using any prefix

- A local scope: innermost scope that contains a list of local names available in the current function.
- A scope of all the enclosing functions. The search for a name starts from the nearest enclosing scope and moves outwards.
- A module level scope that contains all the global names from the current module.
- The outermost scope that contains a list of all the built-in names. This scope is searched last to find the name that you referenced.

Good practices

If you need a constant, to be used across multiple functions in a module, define it as a global variable (typically after imports). Use `CAPITAL` convention.

Try to avoid shadowing: occurs when a variable declared within a certain scope has the same name as a variable declared in an outer scope.

Avoid ”side effects” of functions: use parameters for inputs and returns for outputs. Python will not create copies

FUNCTIONS II

- *Execution frames and namespaces*
- *Scope of variables*
- *Local and global keywords*

Global, local and non local variables

- **Global variable:** defined at module level (in the global namespace). Can be used by any function in the module.
- **Local variable:** defined in a function used inside the function
- **Non local variable:** defined in a function, used in a nested function.

Global and local variables

Universidad Francisco de Vitoria

UFV Madrid

```
# This is OK. In my_function we can resolve the
# global variable global_var, even if defined later

def my_funciton():
    local_var = 2
    print(global_var, local_var)


global_var = 1
my_funciton()
```

Shadowing


```
# This generates an Exception, because global_var is defined in the
# function and therefore it is considered local,
# but in print, it is not defined

# It is, in general, bad practice to "shadow" with local variables
# variables names from outer scope

def my_funciton():
    local_var = 2
    print(global_var, local_var)
    global_var = 3


global_var = 1
my_funciton()
```

Exception (error)

Shadowing (II)


```
# This does not generate an error, but, again, local_var
# in inner is shadowing the same name in an outer scope
# Inner does not change outer's local_var

def outer():
    def inner():
        local_var = 7
        print(local_var)

    local_var = 2
    print(local_var)
    inner()
    print(local_var)

outer()
```

Global keyword

We want to ensure that the reference is to a global variable, and we are not defining a new variable:

We indicate that `a_variable`, in `inner`, is not redefined, but tied to global variable.

```
def outer():
    def inner():
        global a_variable
        a_variable = 7
        print(a_variable)

    a_variable = 2
    print(a_variable)
    inner()
    print(a_variable)


a_variable = 10
outer()
print(a_variable)
```

With no `global` keyword, `a_variable` is defined here as a local variable in `inner`

```
def outer():
    def inner():
        a_variable = 7
        print(a_variable)

    a_variable = 2
    print(a_variable)
    inner()
    print(a_variable)


a_variable = 10
outer()
print(a_variable)
```

Nonlocal keyword

Functions within functions

The global keyword is no help here as the outer function's variables are not global, they are local to a function.

Nonlocal: look within the scope in which the function is defined to find a local variable with the same name

```
def outer():
    title = "original title"

    def inner():
        nonlocal title
        title = "another title"
        print("inner:", title)

    inner()
    print("outer:", title)
```