UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES
Centro de e-Learning

UNIDAD DIDÁCTICA IV

DIPLOMATURA EN PYTHON

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Módulo I – Nivel Inicial

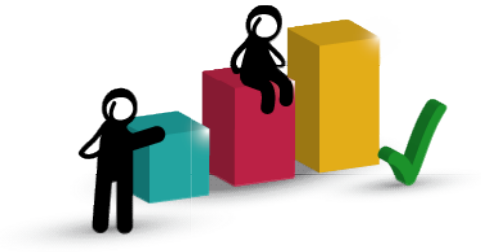
Unidad IV – Funciones y asignación de argumentos.

Bloques temáticos:

1.- Global vs nonlocal.

2.- Asignación de argumentos.

3.- Funciones.



Consignas para el aprendizaje colaborativo

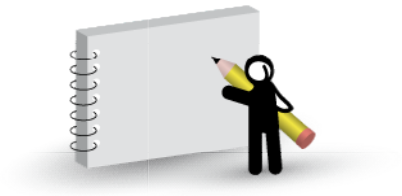
En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.



Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1. Global vs local

Para comprender el concepto de variable global o local a una función, analicemos el siguiente ejemplo:

funciones_1

```
a = 5  # a es global

def sumar_cinco(b):

    c = a + b  # b y c son locales a la función

    return c

print(sumar_cinco(5))
```

En el ejemplo anterior, la variable “a” es global y por lo tanto el valor puede ser utilizado tanto dentro de cualquier función que lo invoque como fuera, sin embargo las variables “b” y “c” son locales a la función “sumarCinco()” con lo cual los valores que toman no pueden ser utilizados fuera de la función. Al ejecutar el código anterior el resultado es igual a 10.

No uso de global

Para reasignar el valor de una variable global dentro de una función, es necesario utilizar la palabra “global”, por lo que, todo intento de modificar una variable global desde dentro de una función mediante una nueva asignación fracasa. Veamos el siguiente ejemplo:

funciones_2

```
a = 5

b = 6

def nopisa():

    a = 10

    print("La variable 'a' dentro de la función tiene por valor", a)
```

```
    print("'b' es global, por lo que puedo impirmirla acá", b)

nopisa()

print("La variable 'a' fuera de la función tiene por valor", a)

print("", end="\n##################\n")
```

En este caso podemos ver que:

- La variable “a” dentro de la función tiene por valor 10
- “b” es global, por lo que puedo imprimirla acá 6
- La variable “a” fuera de la función tiene valor 5

Una variable no puede ser global y local a la vez

Dentro de una función, la misma variable no puede ser en unos momentos global y en otros local. Si hay una asignación, aunque sea posterior a su uso como variable global, la variable será considerada local y se producirá un error:

funciones_3

```
a = 5

def funcion():

    print(a)  # esta es global

    a = 10  # esta es local

funcion()
```

El resultado de invocar la función da un error.

Una variable no puede ser global y local a la vez - global

Cuando queremos acceder a una variable global desde dentro de una función y cambiar su valor mediante una nueva asignación hay que calificar la variable externa dentro de la función empleando la palabra global.

funciones_4

```
a = 5

def funcion():

    global a

    a = 10  # redefino "a"

    print(a)  # esta es global

funcion()

print(a)
```

El resultado de la ejecución del código anterior retorna dos valores de 10 pues la variable global ha sido reasignada.

Nota 1: Más adelante en este curso veremos el uso de nonlocal y anidamiento y niveles de variables globales.

Nota 2: Más adelante veremos la utilización de conceptos de funciones más avanzados, como son el uso de “nonlocal”, “yield” y “lamda”, por ahora no es conveniente introducir muchos conocimientos juntos para que los aprehendidos se puedan asentar correctamente, además de que es necesario estudiar antes los bucles y estructuras de control.

Atención al establecer variables globales.

Hay que usar las variables globales con precaución. Permitir a las funciones que modifiquen nuestras variables externas es una práctica que puede dificultar la localización de errores cuando las cosas no funcionan como debieran. Sin embargo, son muy útiles para almacenar información de estado que luego podrá recuperarse al invocar nuevamente la función u otra diferente. En el ejemplo siguiente, utilizamos la variable global suma, para retener el efecto de cada invocación a la función:

cambiar_valor.py

```
ingreso = 0

def nuevoIngreso():
    global ingreso
    ingreso += 1
    print('Se ha realizado un nuevo ingreso',ingreso)

nuevoIngreso()
nuevoIngreso()
nuevoIngreso()
nuevoIngreso()
print(ingreso)
```

En el ejemplo anterior cada vez que llamo a la función, redefino el valor de la variable "ingreso"; la salida retorna:

```
Se ha realizado un nuevo ingreso 1
Se ha realizado un nuevo ingreso 2
Se ha realizado un nuevo ingreso 3
Se ha realizado un nuevo ingreso 4
4
```

Anidamiento y niveles – global vs nonlocal.

Una función puede incluir definiciones de otras funciones, y de esta forma crear un anidamiento de funciones, veamos un ejemplo simple para comprender el concepto:

global_vs_local_anidar.py

```
nivel0 = 0
```

```
def f1():
    nivel1 = 1

    def f2():
        nivel2 = 2
        print(nivel0, nivel1, nivel2)

    f2()
    print(nivel0, nivel1)


f1()
print(nivel0)
```

Nota 1: En el ejemplo anterior “nivel0” lo tomamos fuera de toda función, por lo que accedemos a dicho valor con un simple print(), y que además es una variable global a las funciones f1() y f2() por lo que su valor puede ser obtenido dentro de ambas funciones.

Nota 2: al ejecutar f2() dentro de f1() se imprimen los valores de nivel0, nivel1 y nivel2.

Nota 3: Desde fuera de las funciones, no se puede acceder a nivel1 y nivel2

Nota 4: Desde f2() podemos consultar el valor de nivel 1 y nivel0 sin poder modificar su valor, y de f1() podemos consultar el valor de nivel0 sin poder modificar su valor. Para modificar los valores podríamos utilizar “global”.

Modifiquemos el ejercicio anterior para poder alterar el valor de nivel 1 desde dentro de f2().

global_vs_local_anidar2.py

```
nivel0 = 0


def f1():
    nivel1 = 1

    def f2():
        global nivel1
```

```
        nivel1 = 7
        nivel2 = 2
        print(nivel0, nivel1, nivel2)

    f2()
    print(nivel0, nivel1)


f1()
print(nivel0)
```

El resultado sería:

```
0 7 2
0 1
0
```

Nota: Aún cuando dentro de f2() se ha utilizado "global" para modificar el valor de nivel1, esta modificación no se ha registrado dentro de f1() ya que al imprimir el valor de nivel 1 desde dentro de f1() su valor sigue siendo igual a 1. ¿Qué es lo que está mal?

En realidad nada está mal, solo que el calificativo global solo puede usarse para variables globales a nivel de módulo (es decir fuera de toda función). Para poder modificar la variable a nivel de función nivel1, necesitamos un nuevo calificador: **nonlocal**.

Si realizamos esta modificación dentro de f2(), de forma tal que nos quede como en el siguiente ejemplo, vemos que ahora la modificación realizada sobre nivel1 desde dentro de f2() es registrada:

global_vs_local_anidar3.py

```
nivel0 = 0


def f1():
    nivel1 = 1

    def f2():
        nonlocal nivel1
```

```
        nivel1 = 7
        nivel2 = 2
        print(nivel0, nivel1, nivel2)

    f2()
    print(nivel0, nivel1)


f1()
print(nivel0)
```

El resultado sería:

```
0 7 2
0 7
0
```

Nota 1: Se debe utilizar global para poder modificar una variable creada a nivel de módulo desde dentro de una función definida en ese módulo, aunque esté anidada dentro de otra función.

Nota 2: Se debe utilizar **nonlocal** para modificar una variable creada a nivel de función desde otra función definida dentro.

2. Funciones - Sintaxis de asignación de argumentos.

Un tema interesante que vamos a analizar ahora es cómo indicar los argumentos pasados a una función, veámoslo punto por punto.

Asignación por posición

La forma más común de pasar argumentos, es respetando la posición, en el siguiente ejemplo notemos cómo pasamos a argumentos dos objetos totalmente distintos (un número y una lista). El primer elemento pasado es el número y el segundo corresponde a la lista.

argumentos1.py

```
def modificar(a, b):
    a = 2
    b[0]="Manzana"
A=1
L=[1,2]
modificar(A,L)
print('',end='\n##################\n' )
print(A, L)
```

Dado que las listas no son INMUTABLES, podemos modificar uno de sus valores, en este caso el primero, por lo que el resultado del script queda como sigue:

```
##################
1 ['Manzana', 2]
```

Asignación por nombre

Es posible indicar el nombre del argumento a pasar, e incluso realizar una mezcla entre asignación por posición y por nombre, veamos tres ejemplos para clarificar este punto.

argumentos2.py

```
# ################################################
# Por posición ####
# ################################################
def f(x, y, z):print(x,y,z)
f(1,2,3)
print('',end='\n################\n' )
# ################################################
# Por nombre ####
# ################################################
def f(x, y, z):print(x,y,z)
f(z=3, y=2, x=1)
print('',end='\n################\n' )
# ################################################
# Por mixto - primero de izquierda a derecha ##
# y luego por nombre ####
# ################################################
def f(x, y, z):print(x,y,z)
f(1 ,z=3, y=2)
```

Asignación por defecto

Algo que vamos a utilizar muy a menudo, es establecer un valor por defecto, es decir que si no se especifica un valor, el valor que toma una variable es el que establecemos en la declaración de la función. Como ejemplo pensemos en los estilos de una aplicación que por defecto viene seteado con color de fondo blanco y letras azules y que posteriormente podemos modificar desde un menú.

Un ejemplo nos va a permitir fijar el conocimiento:

argumentos3.py

```
def f(a, b=2, c=3): print(a, b, c)
f(1)
```

Nota: En este caso el único valor que debemos pasarle de forma obligada es el valor de "a", sin embargo tanto en el caso de "b" como de "c" no es necesario pasar un valor a no ser que queramos pisar el valor determinado por defecto.

Uso de * y **

Mediante el uso de * podemos pasar tuplas o listas a una función, si dentro de la función usamos "args" podemos manejar los elementos como si fueran una tupla.

argumentos4.py

```
def f(a, *args):
    print("Tupla de valores: ", args)
        for arg in args:
        print("Elemento de la tupla: " , arg)

f(0, 1, 2, "Manzana")
```

La salida queda:

```
Tupla de valores: (1, 2, 'Manzana')
Elemento de la tupla: 1
Elemento de la tupla: 2
Elemento de la tupla: Manzana
```

Nota: El uso del nombre args es arbitrario podría usar otro nombre.

Si en lugar de un asterisco ponemos dos, python interpreta los datos pasados como un diccionario.

argumentos5.py

```
def funcion(**kwargs):
    if kwargs is not None:
        for clave, valor in kwargs.items():
            print( "%s == %s" %(clave, valor))
funcion(nombre="Juan", edad=1, sexo="Masculino")
```

Retorna:

```
edad == 1
nombre == Juan
sexo == Masculino
```

Un ejemplo mixto de lo visto hasta ahora nos puede evidenciar lo amplio del lenguaje de Python en la asignación de argumentos.

argumentos6.py

```
def funcion(a, *pargs, **kwargs):
    print(a, pargs,kwargs)
    print("-----------")
    print(a)
    print("-----------")
    print(pargs)
    print("-----------")
    if kwargs is not None:
        for clave, valor in kwargs.items():
            print( "%s == %s" %(clave, valor))

funcion(1, 2, 3, nombre="Juan", edad=1, sexo="Masculino")
```

Retorna:

```
1 (2, 3) {'sexo': 'Masculino', 'edad': 1, 'nombre': 'Juan'}
-----------
1
-----------
(2, 3)
-----------
sexo == Masculino
edad == 1
nombre == Juan
```

Nota: Los nombres pargs, y kwargs son totalmente arbitrarios.

Nota: Lo que sí es importante es en el caso mixto respetar el orden a, *pargs, **kwargs

3. Funciones avanzadas

Recursividad.

Recurrencia, recursión o recursividad es la forma en la cual se especifica un proceso basado en su propia definición. Nada mejor que un ejemplo gráfico para comenzar a darnos una idea de qué se trata.

https://www.quora.com/Is-there-a-blank-mirror-at-the-end-of-an-Infinity-Mirror-reflection

Este concepto es muy útil en programación, en donde muchas veces tomamos el resultado de la ejecución de una función como entrada de la misma función, veamos un ejemplo para clarificar el concepto:

Supongamos que tenemos una función como la que sigue, que lo que hace es imprimir el valor de la lista que le pasamos, luego si la lista es vacía retorna cero, si no nos da el primer elementos de la lista y ejecuta nuevamente la función, pero esta vez quitándole el primer elemento y sumando el resultado de la nueva ejecución al primer elementos guardado.

recursividad.py

```
def misuma(L):
    print(L)
    if not L:
        return 0
    else:
        return L[0] + misuma(L[1:])
print(misuma([1, 2, 3, 4, 5]))
```

Como resultado lo que obtenemos es la impresión de los valores obtenidos en cada vuelta y al final cuando misuma() ya no se ejecuta el valor de la suma de cada elemento.

Retorna:

```
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[]
15
```

Nota: Podemos utilizar una función ternaria para obtener la suma de los elementos con una notación un poco más escueta, con la ayuda de if/else.

recursividad_ternarias.py

```
def miSuma(L):
    return 0 if not L else L[0] + miSuma(L[1:])
print(miSuma([1, 2, 3, 4, 5]))
```

Recursividad indirecta.

En algunas ocasiones mientras trabajamos podemos definir que una parte de las cuentas o mejor dicho de la lógica de un script lo lleve a cabo otra función, para facilitar la lectura de código y poder encontrar errores más rápidamente.

Veamos cómo queda el ejercicio anterior, si utilizamos una segunda función que se encarga de la suma de los elementos.

recursividad_indirecta.py

```
def misuma(L):
    if not L:
        return 0
    return novacia(L)            # llama a otra función que llama a misuma

def novacia(L):
    return L[0] + misuma(L[1:]) # Recursividad indirecta

print(misuma([1.1, 2.2, 3.3, 4.4]))
```

Recursividad vs loop.

Lo que podemos hacer con una recursión, lo podemos realizar con un loop como por ejemplo un while

recursividad_loop.py

```
L = [1, 2, 3, 4, 5]
suma = 0
while L:
    suma += L[0]
    L=L[1:]
print(suma)
```

Datos de la función.

A continuación ejecutamos algunas líneas de código para obtener datos de la función ejecutada.

datos_funcion.py

```
def misuma(L):
    print(L)
    if not L:
        return 0
    else:
        return L[0] + misuma(L[1:])

print(misuma([1, 2, 3, 4, 5]))
print('',end='\n##################\n' )
print(misuma.__name__) #Obtengo el nombre
print('',end='\n##################\n' )
print(dir(misuma)) #Obtengo sus atributos
print('',end='\n##################\n' )
print(misuma.__code__) #Datos de la función
print('',end='\n##################\n' )
print(dir(misuma.__code__)) #Datos de la función
print('',end='\n##################\n' )
print(misuma.__code__.co_varnames) #Nombre de variables utilizadas
print('',end='\n##################\n' )
print(misuma.__code__.co_argcount) #Cantidad de argumentos
```

Función Lambda.

Una función lambda, es una función con una sintaxis mínima que puede ser definida en cualquier momento dentro del código.

La forma general de definir una función lambda es:

lamda.py
```
lambda argument1, argument2,... argumentN : expression using arguments
```

Tomemos como ejemplo la siguiente función

```
def func1(a, b, c):return a + b + c
print(func1(1, 2, 3))
```

Puedo escribir una función lambda equivalente como a continuación:

```
func2 = lambda a,b,c: a+b+c
print(func2(1, 2, 3))
```

Nota 1: La lista de argumentos no está entre paréntesis.

Nota 2: La palabra reservada return está implícita, ya que la función entera debe ser una única expresión.

Nota 3: La función no tiene nombre, pero puede ser llamada mediante la variable a que se ha asignado.

Nota4: Es posible definir una función lambda sin asignarla a una variable, pero su aplicación no es muy útil.

Incluso puedo establecer valores por defecto:

```
func3 =(lambda a= 4, b= 5, c= 6 :a+b+c)
print(func3(1, 2, 3))
```