UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

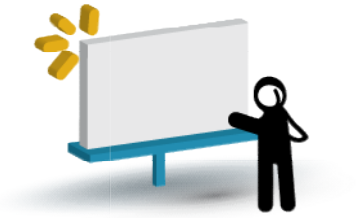
## UNIDAD DIDÁCTICA IV

# 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 I

# Unidad IV – Funciones I, ámbito de variable, estructura de control y bucles.



# Presentación:

Para poder modificar el flujo de ejecución de las instrucciones dentro de un programa, contamos con la implementación de estructuras de control, mientras que por medio de los bucles podemos ejecutar una sentencia o bloque de código repetidas veces mientras se cumpla una determinada condición. En esta unidad profundizaremos en el análisis e implementación de funciones y haremos uso de las estructuras de control y de los bucles para potenciar nuestros desarrollos.

Si realizamos un análisis de varios lenguajes, como php, javascript, java, ruby, c#, etc veremos que tanto las estructuras de control como los bucles tienen sintaxis similares, aunque python en particular tiene algunas diferencias puntuales, especialmente en la forma de implementar el bucle for.

# Objetivos:

## Que los participantes:

Profundicen en el conocimiento de las funciones.

Se introduzcan en el uso y análisis de las estructuras de control y bucles.

## Bloques temáticos:

1.- Funciones.

2.- Estructuras de control y bucles.

3.- Ejercicio 1.

4.- Ejercicio 2.

5.- Más sobre funciones.

## Consignas para el aprendizaje colaborativo

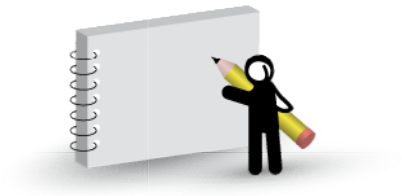
En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- 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. Funciones

Las variables se utilizan para almacenar información, en cambio las ***funciones*** se utilizan para realizar algo, como por ejemplo sumar, restar, establecer movimientos complejos de objetos, etc.

La estructura de una función es como sigue:

**funciones_1**

```
def    printer(mensaje):
         print('hola' + mensaje)
```

En primer lugar aparece la declaración de función “def”, luego viene el nombre de la función, a continuación entre paréntesis se le pueden pasar argumentos que la función puede utilizar dentro del código, el cual va luego de dos puntos “:” en una nueva línea y aplicando una indentación. La indentación en Python indica que el código indentado ( código con espacios en blanco al inicio de cada línea) se encuentra dentro de la declaración, por lo que todo el código dentro de la función debe tener el mismo tipo de indentación (cantidad de espacios antes de una línea de código) en cada línea para que se considere dentro del mismo nivel. Las indentaciones indican niveles.

Para ejecutar el código se llama a la función y se le indican el parámetro a utilizar como argumentos de la siguiente forma:

```
Nombre(parámetro1, ….., parámetron)
```

O sea, el nombre de la función conteniendo los parámetros a ser pasados como argumentos de la función, en nuestro caso sería por ejemplo:

```
printer(“Hola Mundo”)
```

Existen algunas palabras reservadas al trabajar con funciones que veremos a continuación.

## Uso de return

El uso de return es similar a otros lenguajes de programación, en donde se utiliza para retornar un valor que luego pueda llegar a ser utilizado en otra parte del código u otras funciones. En el caso siguiente, el valor retornado es guardado dentro de la variable a y luego se imprime a.

**funciones_2**

```
def    printer(mensaje):
         return mensaje
a = printer("Hola Mundo")
print(a)
```

## Global vs local

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

**funciones_3**

```
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_4**

```
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_5**

```
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_6

```
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.

# 2. Estructuras de control y bucles

Otro componente fundamental en nuestros scripts lo constituyen los bucles y las estructuras de control, diseñados para cumplir con la tarea de tomar decisiones, seleccionar elementos, etc. Todos los lenguajes de programación cuentan con estas herramientas aún cuando su sintaxis puede variar un poco entre lenguajes. A continuación veremos el uso de: if – if/else y for

## if if/elif/else

La estructura if nos sirve para tomar decisiones, su estructura básica es como sigue:

```
if x:
    if y:
        Declaración1
else:
    Declaración2
```

El código anterior se lee como: si x es verdadero; si y es verdadero entonces se cumple el código “Declaración 1” , sino se cumple el código “Declaración 2”.

**Notar que como regla:**

- La condición no va entre paréntesis.
- Los finales de línea no llevan punto y coma.
- El código de cada sección no lleva delimitadores.
- Se indica a quién pertenece por medio de la indentación.
- No tiene por qué ser una tabulación, puede ser espacios pero siempre deben ser la misma cantidad para cada declaración
- Las declaraciones pueden ponerse en una misma línea si se separan por punto y coma

## Declaraciones largas

Es posible poner una condición en múltiples líneas si ésta se pone entre paréntesis, un ejemplo podría ser:

```
a = 1
b = 2
c = 3
print('spam' * 3)
if a == 1 and b == 2 and c == 3:
        print('REPETIR ' * 3)

if (a == 1 and
   b == 2 and
   c == 3):
        print('REPETIR' * 3)
```

## Uso de elif

El uso de elif nos permite evaluar varias condiciones, su estructura de uso es como sigue:

```
if condición1:
   Código1
elif condición2:
   Código2
else:
   Código3
```

## Uso de “in” dentro de la condición

Dentro de una estructura if, podemos utilizar la palabra reservada “in” la cual nos puede ayudar en el caso que necesitemos determinar si por ejemplo un determinado substring se encuentra dentro de un string

estructura.py

```
nombre = 'juan'
if 'an' in 'juan':
        print('a')
        if nombre.endswith('n'):
                nombre *=2
```

```
        print(nombre)
```

**Nota:** endswith() se utiliza para determinar si un string termina con un determinado substring.

## Expresión ternaria

La estructura if/else se puede abreviar dentro de una línea, por lo que el código:

```
if x:
    a = y
else:
    a = z
```

Quedaría de la siguiente forma:

```
a = y if x else z
```

## for

La estructura for, es mucho más rica que en otros lenguajes de programación, por lo que más adelante daremos un análisis completo de esta estructura. Por ahora y para comenzar a realizar ejercicios simples vamos a definir su forma básica como sigue:

```
for objetivo in objeto:
    código1
else:
    código2
```

Esta forma poco típica de utilización de un bucle for puede sorprender a quien viene de otros lenguajes de programación, sin embargo es de uso frecuente en Python y nos permite ejecutar un código dependiendo de si un determinado objeto se encuentra dentro de otro o no. Notar que es posible combinar una estructura for con un else.

# 3. Ejercicio 1

Comencemos por generar una lista que recolecta datos de personas (nombre, edad, salario, profesión) y los guarde como si fuera una base de datos.

¿Qué tipo de datos quiero guardar?

Para almacenar datos, lo podemos realizar en listas, aunque en principio tenemos dos desventajas.

1.- Una vez que salimos de Python, los objetos en memoria dejan de existir

2.- Debemos recordar la posición de cada tipo de dato en la lista.

listas_dentro_de_listas.py

```
# ######################################################
# Definir listas
# ######################################################
juan = ["Juan Garcia", 24, 5000, "Pintor"]
susana = ["Susana Gomez", 25, 6000, "Empleada"]
print(juan[0])
# ######################################################
# Definir lista como Base de datos
# ######################################################
personas = [juan, susana]
# ######################################################
# Imprimo elementos de la lista de base de datos
# ######################################################
for x in personas:
    print(x)
print("----------------------------")
# ######################################################
# Imprimo los apellidos y les doy aumento del 20%
# ######################################################
for x in personas:
    print((x[0].split()[-1]))
    x[2] *= 1.20
for x in personas:
    print(x)
print("----------------------------")
# ######################################################
# Agregamos un registro a la lista de base de datos
# ######################################################
```

```
personas.append(["Pedro", 37, 7000, "Plomero"])
for x in personas:
    print(x)
print("----------------------------")
```

Retorna:

```
Juan Garcia
['Juan Garcia', 24, 5000, 'Pintor']
['Susana Gomez', 25, 6000, 'Empleada']
----------------------------
Garcia
Gomez
['Juan Garcia', 24, 6000.0, 'Pintor']
['Susana Gomez', 25, 7200.0, 'Empleada']
----------------------------
['Juan Garcia', 24, 6000.0, 'Pintor']
['Susana Gomez', 25, 7200.0, 'Empleada']
['Pedro', 37, 7000, 'Plomero']
----------------------------
```

# 4. Ejercicio 2

## Listas dentro de listas (recorrerlas con for if else) - isinstance()

**listas_dentro_de_listas_b.py**

```
lista = ["elemento1n1", "elemento2n1", "elemento3n1",
["elemento1n2", "elemento2n2", "elemento3n2",
["elemento1n3", "elemento2n3", "elemento3n3"]]]
# imprimimos la lista
print(lista)
print('', end='\n##################\n')
# imprimimos elementos de primer nivel
for a in lista:
    print(a)
print('', end='\n##################\n')
# imprimimos elementos de primer y segundo nivel
# con el if veo si el elemento de lista no es una lista
# si es una lista recorro la lista y sino paso al siguiente elemento
for a in lista:
    if isinstance(a, list):
        for b in a:
            print(b)
    else:
        print(a)
print('', end='\n##################\n')
# imprimimos elementos de primer y segundo y tercer nivel
# con el if veo si el elemento de lista no es una lista
# si es una lista recorro la lista y sino paso al siguiente elemento
for a in lista:
    if isinstance(a, list):
        for b in a:
            if isinstance(b, list):
                for c in b:
                    print(c)
            else:
                print(b)
    else:
        print(a)
```

Retorna:

```
['elemento1n1', 'elemento2n1', 'elemento3n1', ['elemento1n2', 'elemento2n2', 'elemento3n2', ['elemento1n3', 'elemento2n3', 'elemento3n3']]]

##################
elemento1n1
elemento2n1
elemento3n1
['elemento1n2', 'elemento2n2', 'elemento3n2', ['elemento1n3', 'elemento2n3', 'elemento3n3']]

##################
elemento1n1
elemento2n1
elemento3n1
elemento1n2
elemento2n2
elemento3n2
['elemento1n3', 'elemento2n3', 'elemento3n3']

##################
elemento1n1
elemento2n1
elemento3n1
elemento1n2
elemento2n2
elemento3n2
elemento1n3
elemento2n3
elemento3n3
```

# Recursividad (Ahorrar código con funciones)

Para ahorrar trabajo podemos invocar a una función que se invoque a sí misma.

**recursividad.py**

```
lista = ["elemento1n1", "elemento2n1", "elemento3n1",
["elemento1n2", "elemento2n2", "elemento3n2",
["elemento1n3", "elemento2n3", "elemento3n3"]]]
# imprimimos la lista
def recorre_lista(item):
    for x in item:
        if isinstance(x, list):
            recorrer_lista(x)
        else:
            print(x)
recorrer_lista(lista)
```

Retorna:

```
elemento1n1
elemento2n1
elemento3n1
elemento1n2
elemento2n2
elemento3n2
elemento1n3
elemento2n3
elemento3n3
```

# Optimizando el código

Podemos optimizar nuestro código si ahora agregamos un segundo parámetro a la función que imprima una separación extra por cada nivel de la lista

**optimizacion1.py**

```
lista = ["elemento1n1", "elemento2n1", "elemento3n1",
["elemento1n2", "elemento2n2", "elemento3n2",
["elemento1n3", "elemento2n3", "elemento3n3"]]]

def recorrer_lista(item, nivel):          # Agrego segundo parámetro
    for x in item:
        if isinstance(x, list):
            recorrer_lista(x, nivel + 1)
        else:
            for y in range(nivel):
                print(("))
            print(x)

recorrer_lista(lista, 0)
```

Retorna:

```
elemento1n1
elemento2n1

elemento3n1

elemento1n2

elemento2n2

elemento3n2


elemento1n3


elemento2n3


elemento3n3
```

Lo que hemos logrado es que se agregue por cada nivel un salto de línea extra.

## Optimizando aún más el código

Podría darse el caso en el cual una persona se olvide de agregar el parámetro que indica el nivel, en cuyo caso el programa daría un error. Para evitar esto le pasamos un valor por defecto al parámetro nivel, y si la persona se olvida de indicarlos al llamar a la función, el programa se seguiría ejecutando.

**optimizacion2.py**

```
lista = ["elemento1n1", "elemento2n1", "elemento3n1",
["elemento1n2", "elemento2n2", "elemento3n2",
["elemento1n3", "elemento2n3", "elemento3n3"]]]

def recorrer_lista(item, nivel=0):      # Agrego valor por defecto
    for x in item:
        if isinstance(x, list):
            recorrer_lista(x, nivel + 1)
        else:
            for y in range(nivel):
                print("\t", end="")          # Agrego indentación en lugar de saltos de línea
            print(x)

recorrer_lista(lista)
```

Retorna:

```
elemento1n1
elemento2n1
elemento3n1
    elemento1n2
    elemento2n2
    elemento3n2
        elemento1n3
        elemento2n3
        elemento3n3
```

Notar que aquí se ha agregado una indentación en lugar del salto de línea con lo cual la salida queda más legible.

# 5. Más sobre funciones

Existen algunas palabras que no podemos usar para nombrar funciones pues son palabras usadas para funciones que vienen con python, podemos obtener el listado con:

```
import builtins
print(dir(builtins))
```

## Uso de Yield.

Yield funciona de manera similar al return, pero la gracia de usar el yield es que conserva la iteración del bucle para la siguiente vez que se le invoque, esto queda más claro con un ejemplo, así que abrimos el intérprete en modo interactivo para hacer el siguiente ejemplo:

**yield.py**

```
def contador_yield1(max):
    n=0
    while n < max:
        yield n
        n+=1
d = contador_yield1(3)
print(d)
print('',end='\n##################\n' )
print(next(d))
print('',end='\n##################\n' )
print(next(d))
print('',end='\n##################\n' )
print(next(d))
```

La salida nos retorna:

```
<generator object contador_yield1 at 0x000000000251B558>
##################
0
##################
1
##################
2
```

Como podemos ver, al imprimir d obtenemos una referencia al objeto creado, y luego con el uso de next, tenemos acceso a las salidas retornadas por yield, en donde cada vez que se invoca a la función con el uso de next() obtenemos un registro de que recuerda el valor retornado la vez anterior (esto podría ser utilizado para contar cuántas veces un usuario ha realizado alguna actividad mientras está logueado)

## 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.

# Bibliografía utilizada y sugerida

## Libros

Programming Python 5th Edition – Mark Lutz – O'Reilly 2013

Programming Python 4th Edition – Mark Lutz – O'Reilly 2011

## Manual online

*https://docs.python.org/3.7/tutorial/*

*https://docs.python.org/3.7/library/index.html*

## Lo que vimos

En esta unidad comenzamos a analizar los tipos de datos que podemos utilizar en python.

## Lo que viene:

En la siguiente unidad seguiremos trabajando y aprendiendo sobre los tipos de datos que podemos utilizar.