

Laboratorios de computación

salas A y B

<i>Profesor:</i>	<i>Marco Antonio Martinez Quintana</i>
<i>Asignatura:</i>	<i>Estructuras de datos y algoritmos I</i>
<i>Grupo:</i>	<i>17</i>
<i>No de Práctica(s):</i>	<i>9</i>
<i>Integrante(s):</i>	<i>González Cuellar Arturo</i>
<i>No. de equipo de cómputo empleado:</i>	
<i>No. de Lista o Brigada</i>	<i>16</i>
<i>Semestre:</i>	<i>2020-2</i>
<i>Fecha de entrega:</i>	<i>1- Abril - 2020</i>
<i>Observaciones:</i>	

Calificación: _____

Introducción a Python (I).

Objetivo:

Aplicar las bases del lenguaje de programación Python en el ambiente de Jupyter notebook.

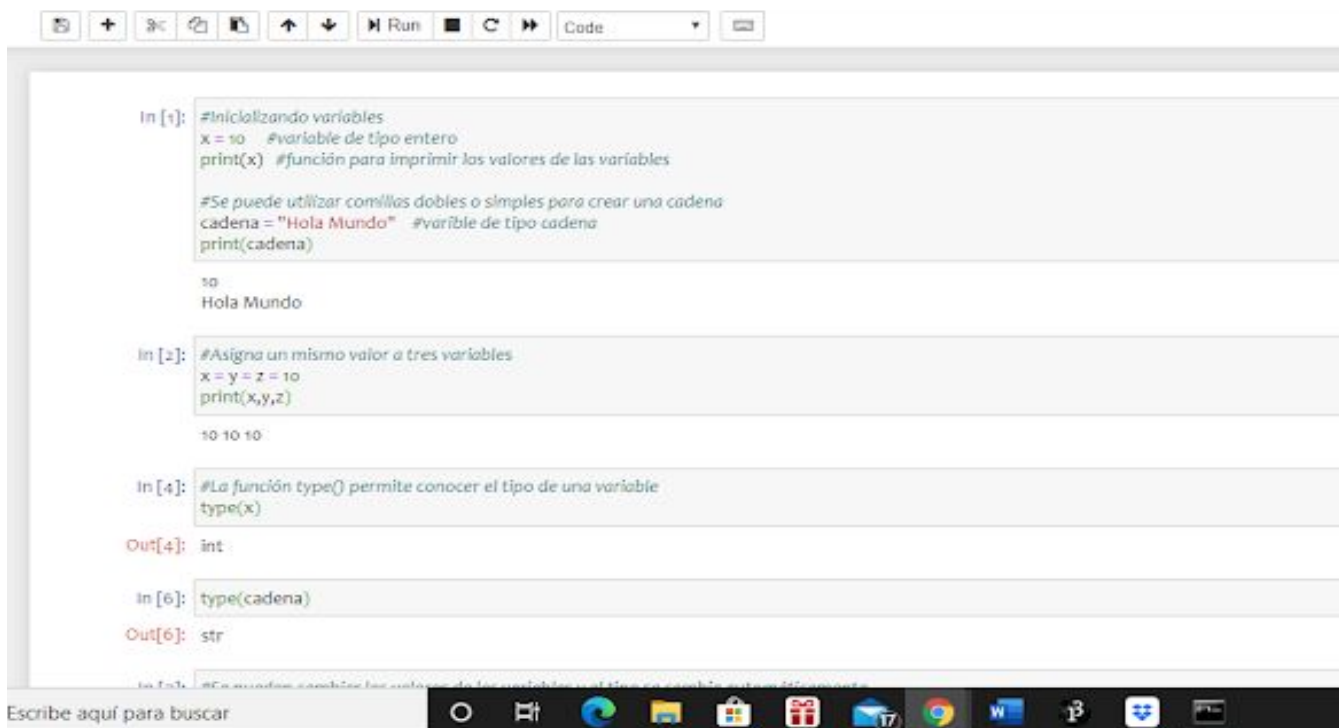
Introducción:

Se aplicaran las bases del lenguaje de programación Python para introducirnos en este lenguaje conociendo cómo interactuar en Jupyter notebook, declarar variables, cadenas, aplicar operadores, crear: listas, tuplas y diccionarios además de crear funciones, todo esto conociendo la sintaxis de este lenguaje.

Desarrollo y resultados:

Variables y tipos:

En este caso se declaran dos variables de nombre “x” y “cadena”, se le asigna un valor, en este caso no se especifica el tipo de valor ya que esta se asigna en el momento de darle un valor, también nos muestra que se pueden asignar un mismo valor para varias variables, además de que si queremos conocer el tipo de una variable, se te utiliza la función type().



```
In [1]: #Iniciando variables
x = 10  #variable de tipo entero
print(x) #función para imprimir los valores de las variables

#Se puede utilizar comillas dobles o simples para crear una cadena
cadena = "Hola Mundo" #variable de tipo cadena
print(cadena)

10
Hola Mundo

In [2]: #Asigna un mismo valor a tres variables
x = y = z = 10
print(x,y,z)

10 10 10

In [4]: #La función type() permite conocer el tipo de una variable
type(x)

Out[4]: int

In [6]: type(cadena)

Out[6]: str
```

```
File Edit View Insert Cell Kernel Help Trusted Python 3
type(x)
Out[4]: int
In [6]: type(cadena)
Out[6]: str
In [7]: #Se pueden cambiar los valores de las variables y el tipo se cambia automáticamente
x = "Hola Mundo"
cadena = 10
In [9]: type(x)
Out[9]: str
In [11]: type(cadena)
Out[11]: int
In [12]: SEGUNDOS_POR_DIA = 60 * 60 * 24
PI = 3.14
In [ ]:
```

Cadenas

Los elementos de las cadenas los podemos definir usando comillas dobles o simples, las cadenas tienen como característica que son inmutables, es decir, que no se pueden cambiar los caracteres que se definieron. En estos ejemplos declaran cadenas con diferentes elementos, además de que las concatenan para que muestren un solo texto, para concatenar hacen uso de la función `format()`.

```
File Edit View Insert Cell Kernel Help Trusted Python 3
In [ ]:
In [13]: #Iniciando cadenas
cadena1 = "Hola"
cadena2 = "Mundo"
print(cadena1)
print(cadena2)
concat_cadenas = cadena1 + cadena2 #Concatenación de cadenas
print(concat_cadenas)

Hola
Mundo
Hola Mundo

In [14]: #Para concatenar un número y una cadena se debe usar la función str()
num_cadena = concat_cadenas + " " + str(3) #Se agrega una cadena vacía para agregar un espacio
print(num_cadena)

Hola Mundo 3

In [15]: #El valor de la variable se va a imprimir en el lugar donde se encuentre {} en la cadena
num_cadena = "{} {} {}".format(cadena1, cadena2, 3)
print(num_cadena)

Hola Mundo 3

In [16]: #Cuando se agrega un número dentro de {}, el valor la variable que se encuentra en esa posición
#dentro de la función format(), será impreso.
num_cadena = "Cambiando el orden: {1} {2} {0} #".format(cadena1, cadena2, 3)
print(num_cadena)

Cambiando el orden: Mundo 3 Hola #
```

Operadores

En estos ejemplos nos muestran como es la sintaxis de los operadores que ya conocemos, **Aritméticos**: +, -, *, / ; **Booleanos**: and, not, or y **Comparación**: >, <, >=, <=, ==.

```
In [1]: #Para el exponente se puede utilizar asterisco
```

```
print( 1 + 5 )
print( 6 * 3 )
print( 10 - 4 )
print( 100 / 50 )
print( 10 % 2 )
print( ((20 * 3) + (10 +1)) / 10 )
print( 2**2 )
```

```
6
18
6
2.0
0
7.1
4
```

```
In [3]: False and True
```

```
Out[3]: False
```

```
In [4]: print( 7 < 5 ) #Falso
```

```
print( 7 > 5 ) #Verdadero
```

```
print( ((11 * 3) + 2 == 36 - 1) ) #Verdadero
```

```
print( ((11 * 3) + 2 >= 36) ) #Falso
```

```
print( "curso" != "CuRsO" ) #Verdadero
```

```
False
True
True
False
True
```

Listas

Las listas son valores que están separados por paréntesis cuadrados, podemos declararlas con cualquier tipo de datos, ya se caracteres, cadenas, números. además de que podemos acceder a cualquier dato de nuestra lista por medio de un índice, comenzando desde el cero. Las listas son mutables.

En estos ejemplos se declara una lista simple y una anidada para que veamos como el la sintaxis para poder acceder a los índices de las listas.

In []:

```
In [5]: #Declaracion de una lista simple
lista_diasDelMes=[31,28,31,30,31,30,31,31,30,31,30,31]

print (lista_diasDelMes)    #imprimir la lista completa
print (lista_diasDelMes[0]) #imprimir elemento 1
print (lista_diasDelMes[6]) #imprimir elemento 7
print (lista_diasDelMes[11]) #imprimir elemento 12

[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
31
31
31
```

In [6]: #Declaracion de listas anidadas

```
lista_numeros=[['cero', 0], ['uno', 1, 'UNO'], ['dos', 2], ['tres', 3], ['cuatro', 4], ['X', 5]]

print (lista_numeros)    #imprimir lista completa

print (lista_numeros[0]) #imprime el elemento 0 de la lista
print (lista_numeros[1]) #imprime el elemento 1 de la lista

print (lista_numeros[2][0]) #imprime el primer elemento de la lista en la posicion 2
print (lista_numeros[2][1]) #imprime el segundo elemento de la lista en la posicion 2

print (lista_numeros[1][0])
print (lista_numeros[1][1])
print (lista_numeros[1][2])

[['cero', 0], ['uno', 1, 'UNO'], ['dos', 2], ['tres', 3], ['cuatro', 4], ['X', 5]]
['cero', 0]
['uno', 1, 'UNO']
dos
2
uno
1
UNO
```

In [7]: #Cambiano el valor de uno de los elementos de la lista

```
lista_numeros[5][0] = "cinco"
print (lista_numeros[5])

['cinco', 5]
```

Tuplas

Las tuplas son muy parecidas a las listas, sin embargo la diferencia que tienen es que estas no son mutables, una ventaja que tienen en comparación a las listas es que estas consumen menos memoria para almacenar.

En estos ejemplos que tenemos nos presentan la declaración de una tupla simple, una anidada y otra en la que nos hacen una prueba de la mutabilidad de las listas contra la inmutabilidad de las tuplas.

```
In [8]: #Declaracion de una tupla
tupla_diasDelMes=(31,28,31,30,31,30,31,31,30,31,30,31)

print(tupla_diasDelMes) #imprimir la tupla completa
print(tupla_diasDelMes[0]) #imprimir elemento 1
print(tupla_diasDelMes[3]) #imprimir elemento 4
print(tupla_diasDelMes[1]) #imprimir elemento 2

(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
31
30
28

In [9]: #Declaracion de tuplas anidadas
tupla_numeros=((('cero', 0),('uno',1,'UNO'), ('dos',2), ('tres', 3), ('cuatro',4), ('X',5)))

print(tupla_numeros) #imprimir tupla completa

print(tupla_numeros[0]) #imprime el elemento 0 de la tupla
print(tupla_numeros[1]) #imprime el elemento 1 de la tupla

print(tupla_numeros[2][0]) #imprime el primer elemento de la tupla en la posicion 2
print(tupla_numeros[2][1]) #imprime el segundo elemento de la tupla en la posicion 2

print(tupla_numeros[1][0])
print(tupla_numeros[1][1])
print(tupla_numeros[1][2])

((('cero', 0), ('uno', 1, 'UNO'), ('dos', 2), ('tres', 3), ('cuatro', 4), ('X', 5)))
('cero', 0)
('uno', 1, 'UNO')
dos
2
UNO
```

```
1
UNO
```

```
In [10]: #Probando la mutabilidad de las listas vs la no mutabilidad de las tuplas
print("valor actual {}".format(lista_diasDelMes[0]))
lista_diasDelMes[0] = 50
print("valor cambiado {}".format(lista_diasDelMes[0]))
tupla_diasDelMes[0] = 50 #Esta asignación manda un error, ya que no se pueden cambiar los valores de las tuplas

valor actual 31
valor cambiado 50

-----
TypeError                                Traceback (most recent call last)
<ipython-input-10-9709ba0ea40a> in <module>
      3 lista_diasDelMes[0] = 50
      4 print("valor cambiado {}".format(lista_diasDelMes[0]))
----> 5 tupla_diasDelMes[0] = 50 #Esta asignación manda un error, ya que no se pueden cambiar los valores de las tuplas

TypeError: 'tuple' object does not support item assignment
```

Tupla con nombre

En comparación con las tuplas anteriores, este tipo especial de tuplas permite especificar un nombre para describirla. Se crea la tupla con nombre, el primer argumento es el nombre de la tupla, mientras que el segundo argumento son los campos #p es la referencia a la tupla. Posteriormente se crea otro argumento y se agregan a la tupla los valores correspondientes a los campos.

```
In [11]: #Se debe importat la librería para hacer uso de namedtuple
from collections import namedtuple

#Se crea la tupla con nombre
#El primer argumento es el nombre de la tupla, mientras que el segundo argumento son los campos
#p es la referencia a la tupla
planeta = namedtuple('planeta', ['nombre', 'numero'])

#Se crea el planeta 1 y se agregan a la tupla los valores correspondientes a los campos
planeta1 = planeta('Mercurio', 1)
print(planeta1)

#Se crea el planeta 2
planeta2 = planeta('Venus', 2)

#Se imprimen los valores de los campos
#Usando la referencia se llama a cada uno de sus campos
print(planeta1.nombre, planeta1.numero)
#Se obtienen los valores por el orden de los campos
print(planeta2[0], planeta2[1])

print('Campos de la tupla: {}'.format(planeta1._fields))

planeta(nombre='Mercurio', numero=1)
Mercurio 1
Venus 2
Campos de la tupla: ('nombre', 'numero')
```

```
print('Campos de la tupla: {}'.format(planeta1._fields))
```

```
planeta(nombre='Mercurio', numero=1)
Mercurio 1
Venus 2
Campos de la tupla: ('nombre', 'numero')
```

```
In [12]: #Al igual que las tuplas, éstas no son mutables, si se trata de cambiar el contenido, se genera un error
planeta1.nombre = 'Tierra'
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-12-0e05c2ba9d3f> in <module>
      1 #Al igual que las tuplas, éstas no son mutables, si se trata de cambiar el contenido, se genera un error
----> 2 planeta1.nombre = 'Tierra'

AttributeError: can't set attribute
```

Diccionarios

Un diccionario consta de dos partes: llave y valor, se crea {}, estas deben tener un solo tipo de dato, ya que una vez creado no se puede cambiar su tipo, además los elementos en un diccionario no están ordenados, ya que al agregar elementos estos se pueden presentar de manera desordenada.

```
In [13]: #Creando un diccionario
elementos = {'hidrogeno': 1, 'helio': 2, 'carbon': 6}

#El momento de la impresion, pueden aparecer en diferente orden del introducido
print (elementos)

print (elementos['hidrogeno'])

{'hidrogeno': 1, 'helio': 2, 'carbon': 6}
1

In [14]: #Se pueen agregar elementos al diccionario
elementos['litio'] = 3
elementos['nitrogeno'] = 8

print (elementos) #Imprimiendo todos los elementos, nótese que los elementos no están ordenados

{'hidrogeno': 1, 'helio': 2, 'carbon': 6, 'litio': 3, 'nitrogeno': 8}

In [15]: #Creando un nuevo diccionario
elementos2 = {}
elementos2['H'] = {'name': 'Hydrogen', 'number': 1, 'weight': 1.00794}
elementos2['He'] = {'name': 'Helium', 'number': 2, 'weight': 4.002602}

print (elementos2)

{'H': {'name': 'Hydrogen', 'number': 1, 'weight': 1.00794}, 'He': {'name': 'Helium', 'number': 2, 'weight': 4.002602}}

In [16]: #Imprimiendo los datos de un elemento del diccionario
print (elementos2['H'])
print (elementos2['H']['name'])
print (elementos2['H']['number'])
elementos2['H']['weight'] = 4.30 #Cambiando el valor de un elemento
print (elementos2['H']['weight'])

{'name': 'Hydrogen', 'number': 1, 'weight': 1.00794}
Hydrogen
1
4.3

In [17]: #Agregando elementos a una llave
elementos2['H'].update({'gas noble': True})
print (elementos2['H'])

{'name': 'Hydrogen', 'number': 1, 'weight': 4.3, 'gas noble': True}

In [18]: #Muestra todos los elementos del diccionario
print (elementos2.items())

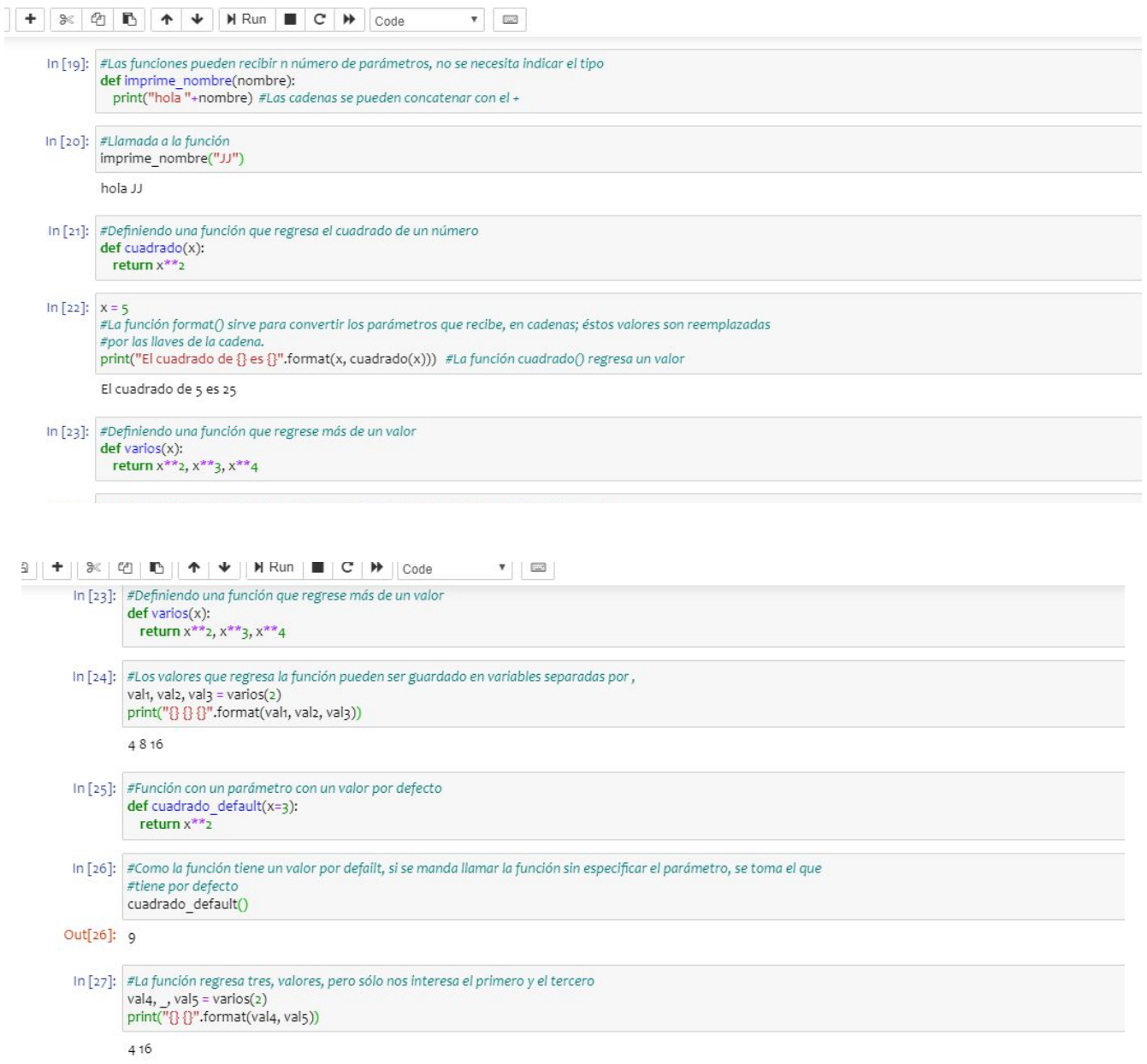
#Muestra todas las llaves del diccionario
print (elementos2.keys())

dict_items([('H', {'name': 'Hydrogen', 'number': 1, 'weight': 4.3, 'gas noble': True}), ('He', {'name': 'Helium', 'number': 2, 'weight': 4.002602})])
dict_keys(['H', 'He'])
```


Funciones

Una función sirve para “guardar” código que puede ser ocupado en otro momento, se puede usar el mismo código con diferentes datos y obtener resultados de acuerdo a lo que se solicite. En estos ejemplos vemos cómo se declara una función, como puede ser llamada y como puede ser ocupado para realizar diversas funciones con la misma. Las funciones pueden recibir n número de parámetros, no se necesita indicar el tipo, La función `format()` sirve para convertir los parámetros que recibe, en cadenas; éstos valores son reemplazados por las llaves de la cadena.

Cuando una función regresa más de una función, se puede usar el operado `'_'`, para no guardar un valor no deseado.



```
In [19]: #Las funciones pueden recibir n número de parámetros, no se necesita indicar el tipo
def imprime_nombre(nombre):
    print("hola "+nombre) #Las cadenas se pueden concatenar con el +

In [20]: #Llamada a la función
imprime_nombre("JJ")

hola JJ

In [21]: #Definiendo una función que regresa el cuadrado de un número
def cuadrado(x):
    return x**2

In [22]: x = 5
#La función format() sirve para convertir los parámetros que recibe, en cadenas; éstos valores son reemplazados
#por las llaves de la cadena.
print("El cuadrado de {} es {}".format(x, cuadrado(x))) #La función cuadrado() regresa un valor

El cuadrado de 5 es 25

In [23]: #Definiendo una función que regrese más de un valor
def varios(x):
    return x**2, x**3, x**4

In [23]: #Definiendo una función que regrese más de un valor
def varios(x):
    return x**2, x**3, x**4

In [24]: #Los valores que regresa la función pueden ser guardado en variables separadas por ,
val1, val2, val3 = varios(2)
print("{} {} {}".format(val1, val2, val3))

4 8 16

In [25]: #Función con un parámetro con un valor por defecto
def cuadrado_default(x=3):
    return x**2

In [26]: #Como la función tiene un valor por default, si se manda llamar la función sin especificar el parámetro, se toma el que
#tiene por defecto
cuadrado_default()

Out[26]: 9

In [27]: #La función regresa tres, valores, pero sólo nos interesa el primero y el tercero
val4, _, val5 = varios(2)
print("{} {}".format(val4, val5))

4 16
```

Variables Globales.

Al momento de iniciar con nuestro programa se crea un espacio de nombres para las variables, hay dos tipos de espacio el primero es el global y el segundo el local, las globales las podremos utilizar durante todo el programa y no se necesita añadir un modificador y las locales solo se podrán utilizar dentro de la función donde fueron declaradas. Una nota que nos recalcan es que el manejo de variables globales dentro de una función en el lenguaje Python se considera como una mala práctica, se recomienda que se pase como parámetro a la función y que se regrese un valor.

```
In [28]: #Se crea una variable en el espacio global de nombres
vg = 'Global'

In [29]: #Se crea una función que imprime la variable global
def funcion_v1():
    print(vg)

In [30]: #Llamada a la función que imprime la variable global
funcion_v1()

#Imprime la variable global
print(vg)

Global
Global

In [31]: #Se crea una variable local que tiene el mismo nombre que la variable global
def funcion_v2():
    vg = "Local"
    print(vg)

In [32]: #Llamada a la función
funcion_v2() #imprime valor local

#imprime la variable global
print(vg)

Local
```

```
Global

In [33]: #Se trata de imprimir el valor de la variable global, a diferencia de la función_v1(), se creó en el
#espacio local de la función_v3() una variable con el mismo nombre, por lo que se reemplaza la variable
#global
def funcion_v3():
    print(vg)
    vg = "Local"
    print(vg)

In [34]: #Como se tiene una variable local y no se le ha asignado un valor, se genera un error
funcion_v3()

UnboundLocalError                                Traceback (most recent call last)
<ipython-input-34-2612dc71fc9c> in <module>
      1 #Como se tiene una variable local y no se le ha asignado un valor, se genera un error
----> 2 funcion_v3()

<ipython-input-33-fazb4e9bfe2a> in funcion_v3()
      3 #global
      4 def funcion_v3():
----> 5     print(vg)
      6     vg = "Local"
      7     print(vg)

UnboundLocalError: local variable 'vg' referenced before assignment
```

```
print(vg)
```

UnboundLocalError: local variable 'vg' referenced before assignment

```
In [35]: #Para resolver el problema anterior y especificar que se quiere hacer uso de la variable global dentro de la
#función funcion_v4(), se tiene que agregar la palabra reservada global
def funcion_v4():
    global vg
    print(vg)
    vg = "Local"
    print(vg)
```

```
In [36]: #Al momento de ejecutar la función se imprime el valor que tenía asignado vg antes de se modificado por la función.
#Después de asignar el valor, éste es impreso
funcion_v4()

#Se imprime la variable global con su valor modificado
print(vg)
```

Global
Local
Local

```
In [ ]: |
```

Ejercicio adicional :

Realizar un programa que calcule el área de un triángulo, un círculo, un rectángulo, y un trapecio.

```
50.26544
```

```
In [43]: lista_numeros=[['Base mayor', 5], ['base menor', 3], ['Radio',4], ['altura',6]]
PI= 3.14159

print("Area del rectangulo")
area_rec = ((lista_numeros[0][1])* (lista_numeros[1][1]))
print(area_rec)

print("Area del triangulo")
area_trian = (((lista_numeros[1][1])* (lista_numeros[3][1]))/2)
print(area_trian)

print("Area del Circulo")
area_circulo = ((PI)*(lista_numeros[2][1])**2)
print(area_circulo)

print("Area del trapecio")
area_trap = (((((lista_numeros[0][1])+(lista_numeros[1][1]))*(lista_numeros[3][1]))/2)
print(area_trap)
```

Area del rectangulo
15
Area del triangulo
9.0
Area del Circulo
50.26544
Area del trapecio
24.0

Conclusión:

Gracias al desarrollo de esta práctica hicimos una introducción a un nuevo lenguaje de programación, hay que mencionar que no es tan complicado entenderlo a estas alturas, ya que tenemos antecedentes del lenguaje C, lo que implica sólo conocer la sintaxis y las características principales de este lenguaje de programación, aunque este es un lenguaje de alto nivel por lo cual nos estaremos introduciendo posteriormente a programación más avanzada con un mayor número de aplicaciones que estos pueden tener.

Referencia:

Manual de prácticas.