



**Universidad Nacional  
Autónoma de México**  
Facultad de Ingeniería



# **Laboratorios de computación salas A y B**

**PROFESOR:** M.I. Marco Antonio Martínez Quintana

**ASIGNATURA:** Estructura de Datos y Algoritmos I

**GRUPO:** 17

**NO DE PRÁCTICA:** 9

**NOMBRE:** Reyes Mendoza Miriam Guadalupe

**SEMESTRE:** 2020-2

**FECHA DE ENTREGA:** 31/03/2020

**OBSERVACIONES:**



**CALIFICACIÓN:**

# INTRODUCCIÓN A PYTHON (I)

## OBJETIVO

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

## INTRODUCCIÓN

### HISTORIA, ANTECEDENTES Y EVOLUCIÓN

Python es un lenguaje de programación orientado a objetos. La programación orientada a objetos (OOP) se refiere a un tipo de programación de computadora en la que los programas definen no sólo el tipo de datos de una estructura, sino también los tipos de operaciones (*funciones*) que pueden aplicarse a la estructura de datos.

De esta manera la estructura de datos se convierte en un objeto que incluye tanto datos como funciones. Además, los programadores pueden crear relaciones entre un objeto y otro, por ejemplo.

Con Python trabajaremos con dos de los múltiples paradigmas de programación: imperativa y funcional. La característica distintiva entre ellos es el concepto de estado. En un lenguaje **imperativo** el estado de la computación *se representa en los valores de las variables que tengamos en el programa*. Cada sentencia (instrucción) revisa un cambio definitivo del estado, añadiendo, cambiando o eliminando una variable. De forma ideal cada sentencia avanza el estado del cómputo desde un estado inicial hasta el resultado final deseado.

La programación **funcional** reemplaza el estado (valores cambiantes de las variables) por la noción simple de *evaluación de funciones*. Cada evaluación de Función crea un nuevo objeto u objeto a partir de objetos existentes. Dado que un programa funcional es una composición de funciones podemos diseñar funciones de nivel inferior factibles de entender, siendo esa composición más fácil de visualizar que una secuencia compleja de sentencias.

### VARIABLES Y TIPOS

La denominación de las variables sigue el concepto más general de identificador, que es un nombre utilizado para identificar entidades (variables, funciones, clases, módulos u otros objetos).

- Pueden ser una combinación de letras en minúscula (**a - z**) o en mayúsculas (**A - Z**), dígitos (**0 - 9**) y el símbolo de guión bajo ("**\_**").
- No puede comenzar con un dígito.

- No podemos usar símbolos especiales (“@”, “!”, “#”, “\$”, “%”, ...).
- No se necesita poner “;” al final de cada instrucción.
- Puede tener cualquier longitud.

The image shows two screenshots of a Jupyter Notebook interface. The top screenshot displays a notebook titled 'Practica(9).ipynb' with seven code cells. The first cell initializes variables `x = 10` and `cadena = "Hola Mundo"`, and prints them. The second cell assigns the value 10 to three variables `x`, `y`, and `z`, and prints them. The third cell uses the `type()` function to check the type of `x`. The fourth and fifth cells are interactive prompts where the user enters `int` and `type(cadena)` respectively. The sixth cell demonstrates variable reassignment, changing `x` to a string and `cadena` to an integer, and then checks their types. The bottom screenshot shows a continuation of the notebook with the output of `type(cadena)` as `int`, and a new cell where `SEGUNDOS_POR_DIA` is calculated as `60 * 60 * 24` and `PI` is assigned the value `3.14`.

```
[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

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

10 10 10

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

[3]: int

[4]: type(cadena)

[4]: str

[5]: #Se pueden cambiar los valores de las variables y el tipo se cambia automáticamente
x = "Hola Mundo"
cadena = 10

[6]: type(x)

[6]: str

[7]: type(cadena)

[7]: int

[8]: SEGUNDOS_POR_DIA = 60 * 60 * 24
PI = 3.14

[ ]:
```

## CADENAS

En su forma más sencilla son vectores de letras que definimos para formar un texto. Podemos utilizarlas para almacenar mensajes que nos sirvan para formar texto. Las cadenas de texto pueden expresarse de diferentes maneras. Una forma de definir las es encerrarlas entre comillas simples (‘ ’) o dobles (“ ”).

**Nota:** Existen caracteres especiales que no pueden ingresarse de manera directa en la cadena y para poder introducirlos se suele utilizar \.

```

[9]: #Inicializando cadenas
cadena1 = 'Hola '
cadena2 = "Mundo"
print(cadena1)
print(cadena2)
concat_cadenas = cadena1 + cadena2 #Concatenación de cadenas
print(concat_cadenas)

Hola
Mundo
Hola Mundo

[10]: #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

[11]: #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

[12]: #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

Un operador en Python es un símbolo especial que realiza un cálculo aritmético o lógico sobre uno o varios elementos. El operador se denomina *unario* si actúa sobre un único operando; si lo hace sobre dos se denomina *binario*. Una *expresión* es una construcción sintáctica compuesta por objetos junto a los operadores.

Existen varios tipos de operadores: aritméticos, de comparación, lógicos, a nivel de bit (bitwise), de asignación o especiales (de identidad y de membresía)

```

[13]: #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

[14]: False and True

[14]: False

[15]: 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

Una lista es una *secuencia contenedora mutable*. Una secuencia es cualquier estructura de datos iterable, con un tamaño conocido, que permite el acceso a sus ítems vía índice entero base 0. Que sea contenedora significa que almacena referencias de objetos en lugar del valor de sus ítems como lo hacen las secuencias planas. Ser mutable nos permitirá cambiar el valor de sus elementos que la componen. Las listas no tienen un tamaño determinado.

Podemos crear listas de varias maneras:

- Usando un par de corchetes para indicar una lista vacía: `[]`
- Usar un par de corchetes incluyendo los ítems separados por comas: `[2, 7, 8]`
- Usando el *constructor*: `list('Hola')`
- Usando las denominadas *listas de compresión*.

```
File Edit View Run Kernel Tabs Settings Help
Practica(9).ipynb
Download GitHub Binder

[16]: #Declaracion de una lista simple
lista_diasDelMes=[31,28,31,30,31,30,31,30,31,30,31]

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

[17]: #Declaracion de listas anidadas

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

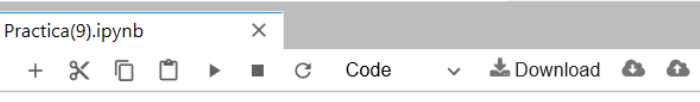
print (lista_numeros)      #imprimir Lista completa

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
```



The screenshot shows a Jupyter Notebook window titled "Practica(9).ipynb". The interface includes a top menu bar with "File", "Edit", "View", "Run", "Kernel", "Tabs", "Settings", and "Help". Below the menu is a toolbar with icons for file operations and execution. The code cell contains the following text:

```
[18]: #Cambiando el valor de uno de los elementos de la lista

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

['cinco', 5]
```

The output area below the code cell is empty, showing only the prompt `[ ]:`.

## TUPLAS

Las tuplas son *secuencias contenedoras inmutables*. Al ser secuencias inmutables, en ellas no podremos cambiar el valor de sus elementos, y es ésta la diferencia principal respecto a las listas. Las tuplas no tienen un tamaño no determinado.

Además, aportan mayor seguridad sobre las listas y tienen un ligero aumento de rendimiento al iterar sobre ellas.

Podemos crear tuplas de varias maneras:

- Usando un par de paréntesis para indicar una tupla vacía: `()`
- Separando los ítems por comas: `(2, 7, 8)`
- Usando el *constructor*: `tuple('Hola')`

En realidad, son las comas las que crean las tuplas, siendo opcionales los paréntesis (salvo en el caso de la tupla vacía en el que pueda haber algún tipo de ambigüedad).

```
File Edit View Run Kernel Tabs Settings Help
Practica(9).ipynb
+ ✂ 📄 ▶ ⏏ Code Download GitHub Binder

[19]: #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

[20]: #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
```

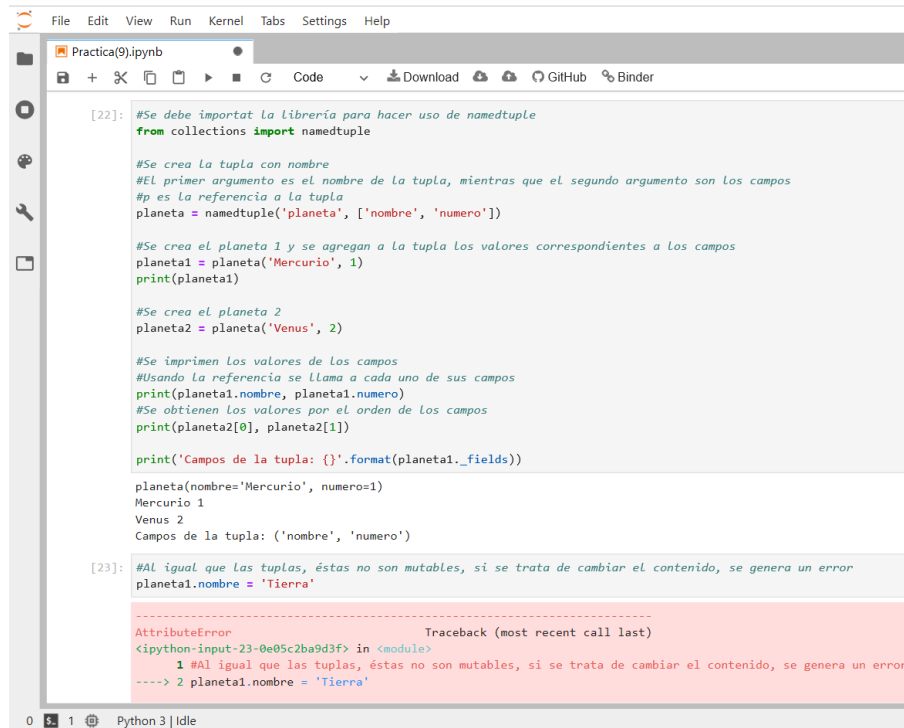
```
File Edit View Run Kernel Tabs Settings Help
Practica(9).ipynb
+ ✂ 📄 ▶ ⏏ Code Download GitHub Binder

[21]: #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-21-9709ba0ea50a> 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
```



```
[22]: #Se debe importar 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')

[23]: #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-23-0e05c2ba9d3f> 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'
```

## DICCIONARIOS

Los diccionarios son **mapas** de colecciones *contenedoras mutables desordenadas* de objetos cada uno de ellos en lugar de una posición relativa, mediante una clave.

Los diccionarios son contenedores, por lo que pueden albergar cualquier tipo de objeto y no tienen un tamaño determinado.

Los diccionarios son mutables como las listas, y al igual que ellas son una herramienta flexible para representar colecciones, sus claves más nemotécnicas se adaptan mejor cuando se nombran los elementos de una colección o tenemos campos etiquetados de un registro de base de datos, por ejemplo.

Las claves pueden ser cualquier objeto y no pueden estar duplicadas (se desaconseja el uso de números punto flotante como claves del diccionario ya que sabemos que se almacenan como aproximaciones).

Los diccionarios son objetos (instancias) de la clase ***dict***. Podemos crearlos de varias maneras:

- Usando un par de llaves para indicar un diccionario vacío: **`{ }`**
- Usando un par de llaves incluyendo los pares clave: valor separados por comas: **`{ 1: 'Ana', 5: 'Pepe', 12: 'Eva' }`**
- Usando el constructor: **`dict (x = 21)`**
- Usando los denominados ***diccionarios de compresión***.

```
File Edit View Run Kernel Tabs Settings Help
Practica(9).ipynb
+ X Copy Paste Run Code Download GitHub Binder

[24]: #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

[25]: #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}

[26]: #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}}

[27]: #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

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

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

[29]: #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

Las funciones sobre un dispositivo de estructuración de programas casi universal en los lenguajes de programación, donde podremos encontrarlas con el nombre de subrutinas o procedimientos. Las funciones sirven para maximizar la reutilización del código minimizar la redundancia. Además de ello nos permiten la descomposición procedural.

De programación procedural viviremos el programa en una serie de subprogramas, y usaremos una función para cada ello. Es más fácil llevar a cabo tareas más pequeñas de forma aislada que la ejecución de todo el proceso a la vez. Las funciones proporcionan una herramienta para dividir los programas en piezas bien definidas.



Al afrontar un problema complejo no intentaremos resolverlo directamente de forma total (con todos los detalles) sino que usaremos el diseño de arriba a abajo dividiendo, sucesivamente el problema original.

```

File Edit View Run Kernel Tabs Settings Help

Practica(9).ipynb
Code Download GitHub Binder

[30]: #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 +

[31]: #Llamada a La función
imprime_nombre("JJ")
hola JJ

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

[33]: 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

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

[35]: #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

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

[37]: #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

```

```

File Edit View Run Kernel Tabs Settings Help

Practica(9).ipynb
Code Download GitHub Binder

[37]: #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()

[37]: 9

[38]: #La función regresa tres, valores, pero sólo nos interesa el primero y el tercero
val4, _, val5 = varios(2)
print("{} {}".format(val4, val5))
4 16

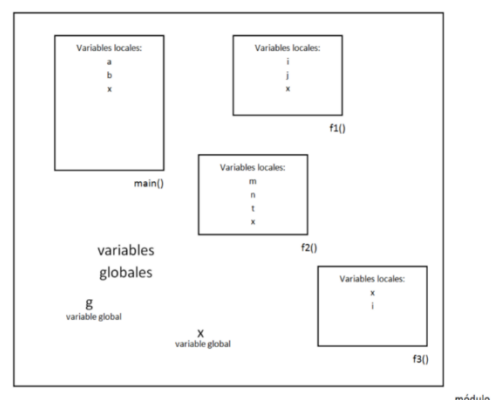
[ ]:

```

## VARIABLES GLOBALES

Si se realiza una operación a nivel módulo (fuera de toda función) su ámbito será **global** es cierto todas las funciones incluidas en el módulo podrán acceder a ellas (pero no modificarlas).

Si se hace una función su ámbito será **local**, por lo que solo existirá ahí. Cualquier modificación de la variable en el cuerpo de la función no tendrá ningún efecto en otras variables fuera de ella, incluso si tienen el mismo nombre.



```
File Edit View Run Kernel Tabs Settings Help

Practica(9).ipynb
+ 🔍 📄 ▶ ⏏ Code ⬇️ Download 📁 GitHub 🌐 Binder

[39]: #Se crea una variable en el espacio Global de nombres
      vg = 'Global'

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

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

      #Imprime la variable global
      print(vg)

      Global
      Global

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

[43]: #Llamada a la función
      funcion_v2() #Imprime valor Local

      #Imprime la variable global
      print(vg)

      Local
      Global

[44]: #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)
```

```
File Edit View Run Kernel Tabs Settings Help

Practica(9).ipynb
+ 🔍 📄 ▶ ⏏ Code ⬇️ Download 📁 GitHub 🌐 Binder

[44]: #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)

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

      -----
      Traceback (most recent call last)
      <ipython-input-45-2612dc71fc9c> 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-44-fa2b4e9bfe2a> 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

[46]: #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)
```

```
File Edit View Run Kernel Tabs Settings Help

Practica(9).ipynb
+ 🔍 📄 ▶ ⏏ Code ⬇️ Download 📁 GitHub 🌐 Binder

[47]: #Al momento de ejecutar la función se imprime el valor que tenía asignado vg antes de ser 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

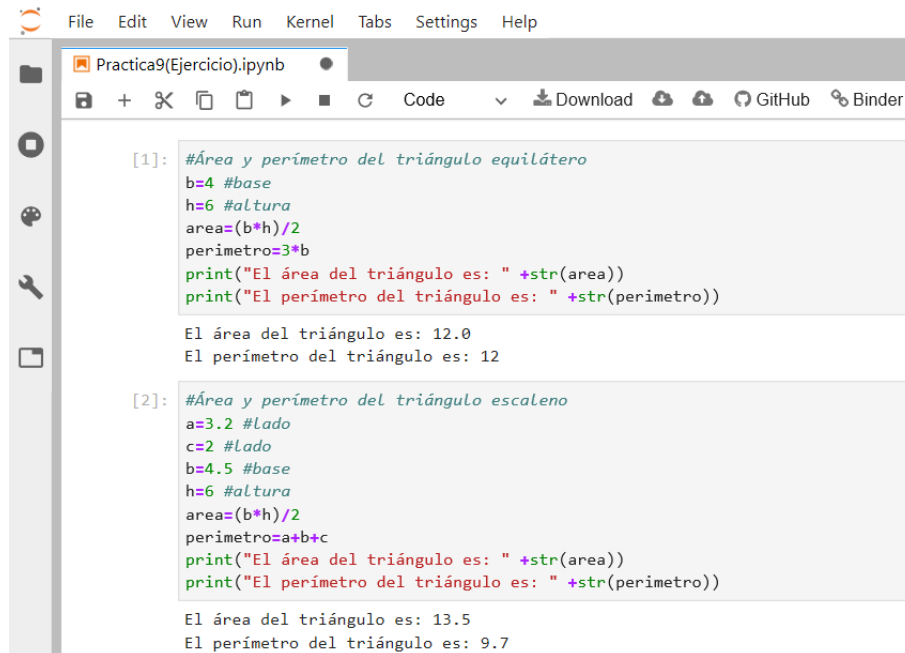
[ ]:
```

## DESARROLLO Y RESULTADOS

### JUPYTER NOTEBOOK

Realizar un programa que calcule el **área** y el **perímetro** de las siguientes figuras:

#### TRIÁNGULO



```
File Edit View Run Kernel Tabs Settings Help
Practica9(Ejercicio).ipynb
+ - Copy Paste Run Stop Code Download GitHub Binder

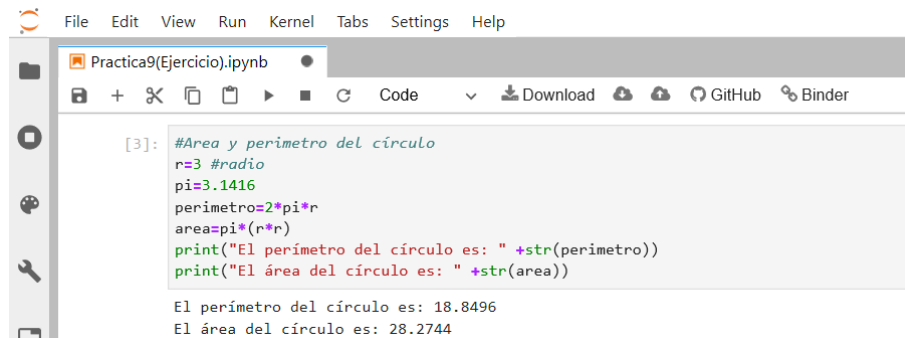
[1]: #Área y perímetro del triángulo equilátero
b=4 #base
h=6 #altura
area=(b*h)/2
perimetro=3*b
print("El área del triángulo es: " +str(area))
print("El perímetro del triángulo es: " +str(perimetro))

El área del triángulo es: 12.0
El perímetro del triángulo es: 12

[2]: #Área y perímetro del triángulo escaleno
a=3.2 #lado
c=2 #lado
b=4.5 #base
h=6 #altura
area=(b*h)/2
perimetro=a+b+c
print("El área del triángulo es: " +str(area))
print("El perímetro del triángulo es: " +str(perimetro))

El área del triángulo es: 13.5
El perímetro del triángulo es: 9.7
```

#### CÍRCULO




```
File Edit View Run Kernel Tabs Settings Help
Practica9(Ejercicio).ipynb
+ - Copy Paste Run Stop Code Download GitHub Binder

[3]: #Area y perímetro del círculo
r=3 #radio
pi=3.1416
perimetro=2*pi*r
area=pi*(r*r)
print("El perímetro del círculo es: " +str(perimetro))
print("El área del círculo es: " +str(area))

El perímetro del círculo es: 18.8496
El área del círculo es: 28.2744
```

#### RECTÁNGULO



```
File Edit View Run Kernel Tabs Settings Help
Practica9(Ejercicio).ipynb
+ - Copy Paste Run Stop Code Download GitHub Binder

[4]: #Area y perímetro del rectángulo
b=5 #base
h=9 #altura
area=b*h
perimetro=(2*b)+(2*h)
print("El área del rectángulo es: " +str(area))
print("El perímetro del rectángulo es: " +str(perimetro))

El área del rectángulo es: 45
El perímetro del rectángulo es: 28
```

### TRAPECIO



```
[5]: #Area y perimetro del trapecio
B=10 #base mayor
b=8 #base menor
i=2.5 #Lado inclinado
a=2 #altura
area=((B+b)*a)/2
perimetro=B+b+(2*i)
print("El área del trapecio es: " +str(area))
print("El perímetro del trapecio es: " +str(perimetro))

El área del trapecio es: 18.0
El perímetro del trapecio es: 23.0
```

## CONCLUSIONES

Con esta práctica que es una breve introducción a Python pude observar que presenta muchas ventajas entre las cuales están su facilidad de uso y la legibilidad de código de una manera más fácil para el usuario, es decir, es un lenguaje muy versátil el cual permite que pueda ser aplicado en diferentes ámbitos dentro de la programación.

Su sintaxis es muy sencilla, además de requiere menos líneas para realizar tareas básicas que si programamos en Java o C. Otra gran ventaja es que tiene una librería estándar, la cual permite ejecutar otras funciones y tareas más complejas con mayor facilidad que otros lenguajes.

Anteriormente no había tenido la oportunidad de trabajar con este lenguaje de programación, sin embargo, con esta práctica he podido darme cuenta de que es muy práctico y eficiente. Esto me hizo buscar información al respecto y darme cuenta de que realmente vale la pena aprender sus bases de forma precisa, por lo cual, me he dado a la tarea de buscar un libro que me ayude a dedicar parte de mi tiempo a conocer más acerca de Python, pues he leído que existen grandes compañías internacionales que lo utilizan para el *desarrollo de aplicaciones y sitios web*.

## BIBLIOGRAFÍA

- Cuevas, A. (2019). *Programar con Python 3* (1ª edición). Madrid, España: RA-MA.
- Guagliano, C. (2019). *Programación en Python 1: Entorno de Programación- Sintaxis. Estructuras de Control* (1ª edición). Buenos Aires, Argentina: Six Ediciones.