RESUMEN CURSO BÁSICO PYTHON

DOCENTE: OLGA CECILIA GARATEJO ESCOBAR

Vídeo de motivación:

https://www.udemy.com/course/master-en-python-aprender-python-django-flask-y-tkinter/?couponCode=SEPTIEMBRE23

INTRODUCCIÓN:

Python es un lenguaje de programación de uso libre, permite crear sitios web, juegos, software científico, gráficos, entre otros, siendo uno de los lenguajes más utilizados hoy en la industria desarrolladora.

Creado a finales de 1989 en los países bajos por Guido Van Rossum, quien fue desarrollador de Google y ahora desarrollador de Dropbox, dicho lenguaje de programación hace parte de CWI (Centro de investigaciones holandés de carácter oficial).

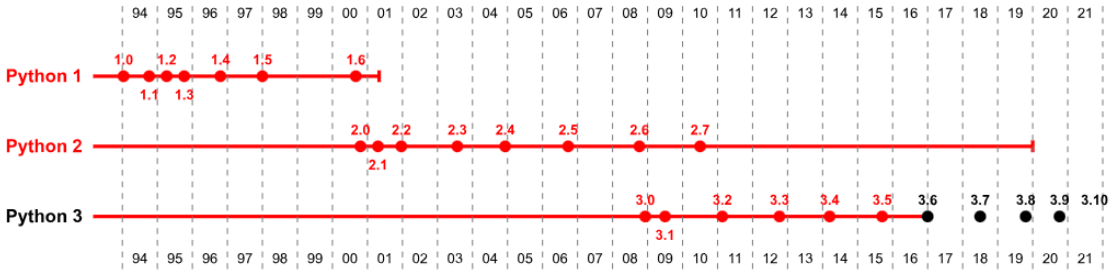
Hinojosa (2016) menciona que el nombre "Python" viene dado por la afición de Van Rossum al grupo musical Monty Python Flying Circus, un grupo cómico británico de seis humoristas de los años 1960. Actualmente el logo de Python son dos serpientes pitón (python en inglés) en colores azul y amarillo, ver figura 1.

Figura 1. Logo actual de Python

En 1991 se publicó la versión 0.9.0. En 1994 se publicó la versión 1.0, en el 2000 se publicó la versión 2.0 y en el 2008 se publicó la versión 3.0.

Desde 2001 hasta 2018, la Python Software Foundation, es dueña de todo el código, documentación y especificaciones del lenguaje. Después del 2019, el desarrollo de Python está dirigido por un consejo de dirección de cinco miembros elegidos entre los desarrolladores de Python.

La Figura 2 presenta la fecha de publicación de las diferentes versiones de Python. Las versiones con punto rojo se consideran obsoletas, las versiones con punto azul se encuentran en actualizaciones y las versiones con punto blanco corresponden a futuras versiones.



Python es un lenguaje de programación interpretado, por lo que funciona en cualquier sistema operativo que integre su interpretador.

Dicho lenguaje es multiplataforma y multiparadigma, sirve para desarrollar aplicaciones web o móviles. Este lenguaje de programación dispone de frameworks que apoyan el desarrollo de juegos y algoritmos científicos de cálculos avanzados. Además, es una herramienta conveniente para el área de Machine Learning.

Este lenguaje cuenta con una gran calidad en su sintaxis, utiliza bloques de código, los cuales llevan indentaciones, lo que garantiza una gran legibilidad en el código fuente. El código de Python 2.x no necesariamente debe correr en Python 3.0.

```
# Ejemplo
---------------------------------
Python 3.x
print ("Hola Mundo")
---------------------------------
Python 2.x
print "Hola mundo"
```

1. Proceso de instalación de Python

Para realizar el proceso de instalación es necesario ir al sitio oficial de Python (http://www.python.org) y descargar el instalador ejecutable en el menú superior: la opción *Downloads.*

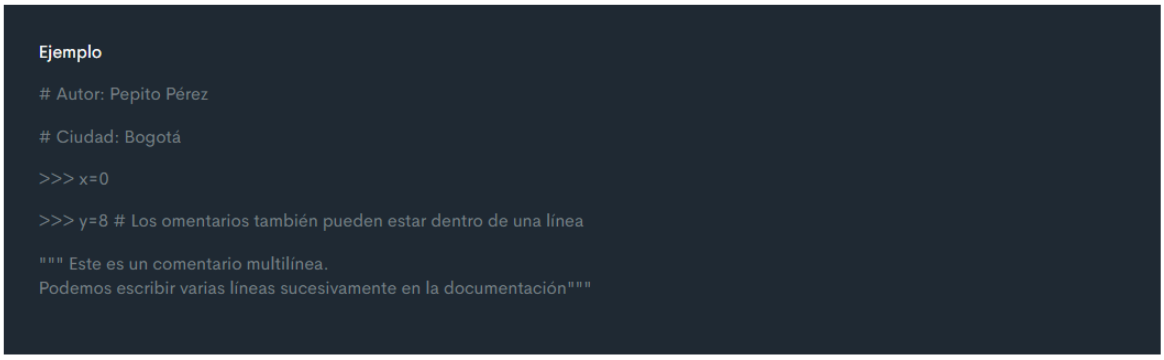
1.1 Comentarios en Python

Un comentario es una línea de texto no ejecutable, esto quiere decir que el compilador o intérprete no la tomará como una línea de código.

Es una buena práctica en programación documentar el código para mayor claridad en un proyecto. Se pueden hacer comentarios en Python para dar pequeñas explicaciones acerca de lo que hace el programa. Se pueden usar tantos comentarios como sean necesarios.

Hay dos formas para hacer comentarios:

1. Digitando el símbolo # al comienzo del comentario.
2. Digitando triple comilla (“) al principio y al final del comentario, cuando ocupan varias líneas.



1.2 Errores de tecleo y excepciones

Si se digita incorrectamente una expresión, Python nos lo indicará con un mensaje de error. El mensaje de error proporciona información acerca del tipo de error cometido y del lugar en el que este ha sido detectado.

En este ejemplo se ha cerrado un paréntesis cuando no había otro abierto previamente, lo cual es incorrecto. Python indica que ha detectado un error de sintaxis (SyntaxError) y señala con el carácter ^ al lugar en el que se encuentra.

En Python los errores se denominan excepciones. Cuando Python es incapaz de analizar una expresión, produce una excepción. Cuando el intérprete interactivo detecta la excepción, nos muestra por pantalla un mensaje de error.

2. Tipos de datos

En Python se encuentran diferentes tipos de datos con sus respectivas características y clasificaciones. A continuación, se detallarán los tipos de datos básicos y otros tipos de datos predefinidos por el lenguaje.

Arias (2019) se refiere a los siguientes tipos de datos básicos de Python: **Numéricos**, **booleanos y cadenas de caracteres**

Python también define otros tipos de datos, entre los que se encuentran:

Secuencias: list, tuple y range.
Conjuntos: set.
Mapas: dict.

2.1 Conceptos: datos numéricos, booleanos, cadena de caracteres, otros tipos de datos:

A. Datos numéricos

Python define tres tipos de datos numéricos: enteros, punto flotante y números complejos.

Números enteros

Se denominan ***int***. Este tipo de dato comprende el conjunto de todos los números enteros, cuyo límite depende de la capacidad de memoria del computador.

Un número de tipo ***int*** se crea a partir de un literal que represente un número entero, o como resultado de una expresión o como una llamada a una función.

```
# Ejemplo
# v es de tipo int y su valor asignado es -3
>>> v = -3
# m es de tipo int y su valor calculado es 5
>>> m = v + 8
>>>print (m)
5
# z es de tipo int y con la función redondeo es 2
>>> z = round(m/2)
>>> print(z)
2
```

También se pueden representar los números enteros en formato binario, octal o hexadecimal.

Para crear un número entero en binario, se antepone 0b a una secuencia de dígitos 0 y 1.
Para crear un número entero en octal, se antepone 0o a una secuencia de dígitos del 0 al 7.
Para crear un número entero en hexadecimal, se antepone 0x a una secuencia de dígitos del 0 al 9 y de la A la F.

```
Ejemplo
>>> decimal = 8
>>> binario = 0b1101
>>> octal = 0o11
>>> hexadecimal = 0xc
>>> print(decimal)
8
>>> print(binario)
13
>>> print(octal)
9
>>> print(hexadecimal)
12
```

Números de punto flotante

Se denominan ***float***. Se usa el tipo ***float*** para representar cualquier número real que represente valores de temperaturas, velocidades, estaturas y otras.

```
# Ejemplo
>>> real = 1.1 + 2.2    # real es un float
>>> print(real)
3.3000000000000003    # representación aproximada de 3.3
>>> print(round(real,1))
3.3    # real mostrando únicamente 1 cifra decimal
```

Números complejos:

Este tipo de datos en Python se denomina **complex.**

Los números complejos tienen una parte real y otra imaginaria y cada una de ellas se representa como un **float.** Los números imaginarios son múltiplos de la unidad imaginaría (la raíz cuadrada de -1)

Para definir un nímero complejo, se hace así:

<parte_real>+<parte_imaginaria>j

Ejemplo: 4+7j

Para acceder a la parte real se usa el atributo *real*

Para acceder a la parte imaginaria se usa el atributo *imag*

```
# Ejemplo
>>> complejo = 5+3j
>>> complejo.real
5.0
>>> complejo.imag
3.0
```

Para tener acceso a los equivalentes complejos del módulo math, se debe usar cmath.

Aritmética de los tipos numéricos

Para todos los tipos numéricos se pueden aplicar las operaciones: suma, resta, producto o división. Para exponentes se usa ** y para la división entera se usa //.

Se permite realizar una operación aritmética con números de distinto tipo. En este caso, el tipo numérico “más pequeño” se convierte al del tipo “más grande”, de forma que el tipo del resultado siempre es el del tipo mayor.

El tipo int es menor que el tipo float, el tipo float es menor que el tipo complex.

Si vamos a sumar un int y un float, el resultado es un float.

Si vamos a sumar un int y un complex, el resultado es un complex.

```
# Ejemplo
>>> x = 2
>>> a = x**3    # a es 8 elevado a la 3
>>>print(a)
8
>>> b = 31
>>> c = b//4    # c es la parte entera de dividir b entre 4
>>> print(c)
7
>>> g = 31.0
>>> h = d/4    # h es la parte entera de dividir g entre 4
>>>print(h)
7.0
>>> 1 + 2.0
3.0
>>> 2+3j + 5.7
(7.7+3j)
```

B. Datos tipo cadena de caracteres

Salazar (2019) denomina este tipo de dato como *string*. Para crear un *string*, se deben encerrar entre comillas simples o dobles una secuencia de caracteres. En Python las cadenas de caracteres se representan con ***str***. Se puede usar comillas simples o dobles. Si en la cadena de caracteres se necesita usar una comilla simple, existen dos opciones: usar comillas dobles para encerrar el *string*, o bien, usar comillas simples, pero anteponer el carácter \ a la comilla simple del interior de la cadena.

```
# Ejemplo
>>> saludo1 = 'Hola "María"'
>>> type(saludo1)
<class ' str ' >
>>> saludo2 = 'Hola \'María\' '
>>> saludo3 = "Hola 'María' "
>>> print(saludo1)
Hola "María"
>>> print(saludo3)
Hola 'María'
```

A diferencia de otros lenguajes, en Python no existe el tipo «caracter». Pero se puede simular con un *string* de un solo caracter:

```
# Ejemplo
>>> caracter1 = 'z'
>>> print(caracter1)
z
```

C. Otros tipos de datos

Adicional a los tipos básicos, se encuentran otros tipos fundamentales de Python denominados secuencias *(list y tuple)*, los conjuntos *(set)* y los mapas *(dict)*.

Pérez (2016) aclara que todos ellos son tipos de datos compuestos y se utilizan para agrupar valores del mismo o diferente tipo

Las listas son arreglos unidimensionales de elementos donde podemos ingresar cualquier tipo de dato, para acceder a estos datos debemos usar un índice. La posición inicial es la posición 0.

```
# Ejemplo
>>> lista = [3, 4.2, 'SENA', [8,9] ,5]    # lista contiene int, real, cadena, list, int
>>> print lista[0]    # la posición 0 de la lista contiene el valor 3
>>> print lista[1]    # la posición 1 de la lista contiene el valor 4.2
>>> print lista[2]    # la posición 2 de la lista contiene la cadena 'SENA'
>>> print lista[3]    # la posición 3 de la lista contiene la lista [8,9]
>>> print lista[4]    # la posición 4 de la lista contiene el valor 5
>>> print lista[3][0]    # la posición 3,0 de la lista contiene 8
>>> print lista[3][1]    # la posición 3,1 de la lista contiene 9
>>> print lista[1:3]    # las posiciones de la 1 a la 3 contienen [4.2, 'SENA']
>>> print lista[1:4]    # las posiciones de la 1 a la 4 contienen [4.2, 'SENA', [8, 9]
>>> print lista[1:5]    # las posiciones de la 1 a la 5 contienen [4.2, 'SENA', [8, 9 ],5 ]
```

Las tuplas se representan escribiendo los elementos entre paréntesis y separados por comas. La función len() devuelve el número de elementos de una tupla. Una tupla puede no contener ningún elemento, es decir, puede ser una tupla vacía. Una tupla puede incluir un único elemento seguido de una coma.

```
# Ejemplo
>>> tupla= (8, "b", 4.91)
>>> tupla
(8, 'b', 4.91)
>>> len(tupla)
3
>>> len(())
0
>>> (3,)
(3,)
>>> len((3,))
1
```

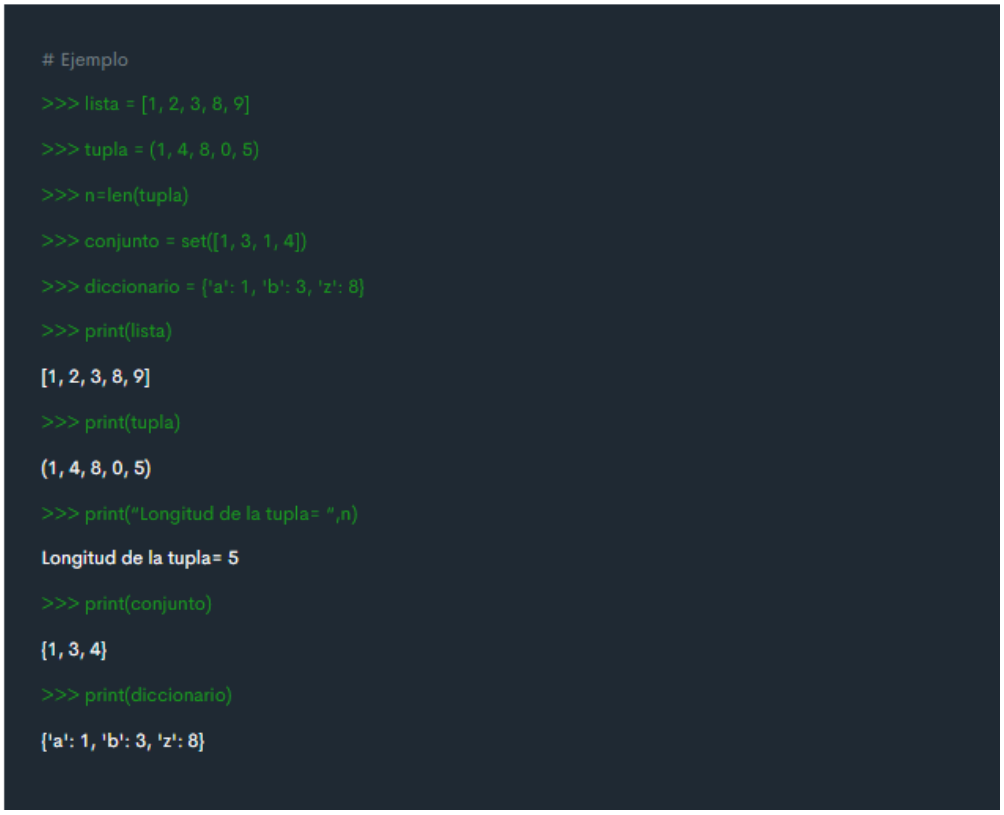
Los conjuntos son una colección no ordenada y sin elementos repetidos. Se definen con la palabra set, seguida de llaves que contienen los elementos separados por comas. Si se desea remover un elemento de un conjunto, se puede usar el método *remove()*.

```
# Ejemplo
>>> frutas = set([ 'mango', 'pera', 'manzana', 'limón' ])
>>> frutas
{'mango', 'manzana', 'pera', 'limón'}
>>> frutas.remove('manzana')
>>> frutas
{'mango', 'pera', 'limón'}
```

Los diccionarios son un tipo de estructuras de datos que permiten guardar un conjunto no ordenado de pares clave-valor, existiendo las claves únicas dentro de un mismo diccionario (es decir, que no pueden tener dos elementos con una misma clave). El diccionario se declara entre los caracteres '{ }' y los elementos se separan por comas (','). Los diccionarios denominados ***dict*** para Python, son estructuras de datos muy extendidos en otros lenguajes de programación, aunque en otros lenguajes como java se les denominan con distintos nombres como *"Map"*.

```
# Ejemplo
# Defino la variable 'futbolistas' como un diccionario.
futbolistas = dict()
futbolistas = {
13 : "Mina", 21 : "Lucumi",
17 : "Fabra", 11 : "Cuadrado",
9 : "Falcao", 19 : "Muriel",
15 : "Uribe", 10 : "James Rodriguez",
16 : "Lerma", 5 : "Wilmar Barrios",
3 : "Murillo"
}
>>> futbolistas
{13: 'Mina', 21: 'Lucumi', 17: 'Fabra', 11: 'Cuadrado', 9: 'Falcao', 19: 'Muriel', 15: 'Uribe', 10: 'James
Rodriguez', 16: 'Lerma', 5: 'Wilmar Barrios', 3: 'Murillo'}
>>> futbolistas[9]
'Falcao'
```

Ejemplo completo de listas, tupla, conjunto y diccionario:



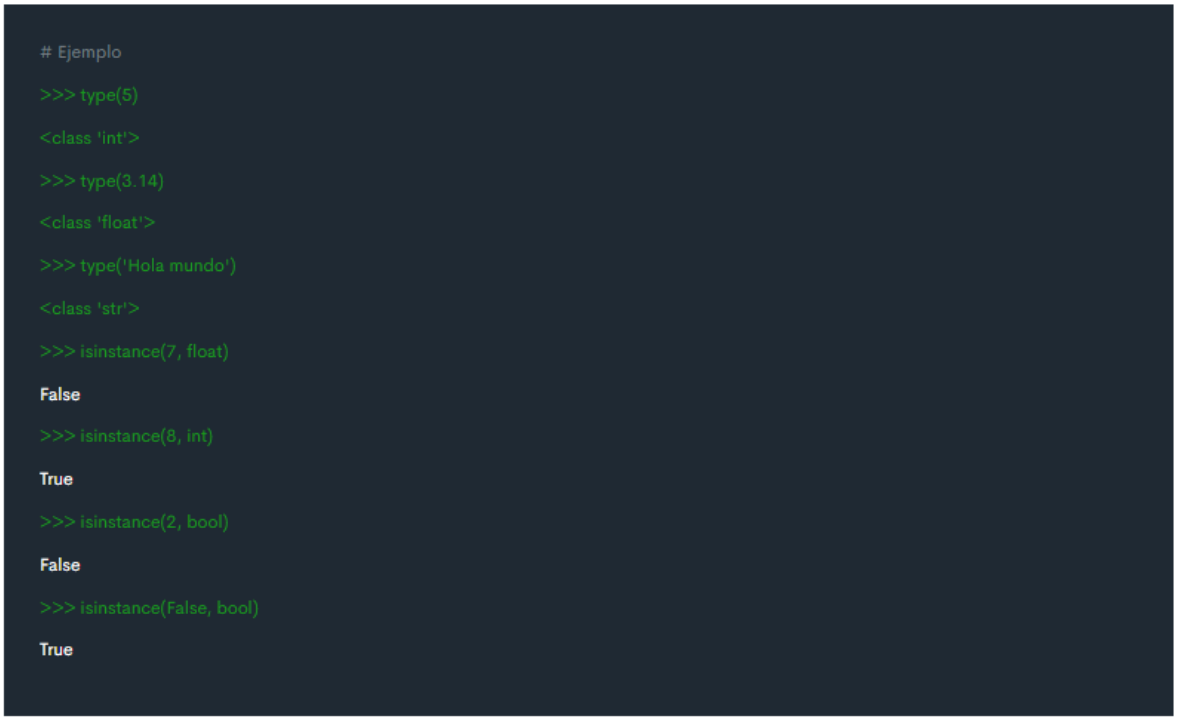
```
# Ejemplo
>>> lista = [1, 2, 3, 8, 9]
>>> tupla = (1, 4, 8, 0, 5)
>>> n=len(tupla)
>>> conjunto = set([1, 3, 1, 4])
>>> diccionario = {'a': 1, 'b': 3, 'z': 8}
>>> print(lista)
[1, 2, 3, 8, 9]
>>> print(tupla)
(1, 4, 8, 0, 5)
>>> print("Longitud de la tupla= ",n)
Longitud de la tupla= 5
>>> print(conjunto)
{1, 3, 4}
>>> print(diccionario)
{'a': 1, 'b': 3, 'z': 8}
```

2.2 Identificar el tipo de variable

Guzdial (2013) afirma que existen dos funciones en Python para determinar el tipo de dato que contiene una variable: type() e isinstance():

type(): Recibe como parámetro un objeto y devuelve el tipo del mismo.

isinstance(): Recibe dos parámetros: un objeto y un tipo. Devuelve True si el objeto es del tipo que se pasa como parámetro y False en caso contrario.



2.3 Conversión de tipos de datos

En algunos casos se requiere convertir el tipo de datos a otro que sea más adecuado. Por ejemplo, si una cadena contiene el valor “10” para poderlo sumar a otra variable tipo entero, se debe convertir la cadena en un dato tipo entero.

Según Cuevas (2017), Python ofrece las siguientes funciones:

str(): devuelve la representación en cadena de caracteres del objeto que se pasa como parámetro.

int(): devuelve un int a partir de un número o secuencia de caracteres.

float(): devuelve un *float* a partir de un número o secuencia de caracteres.

NOTA: Si a las funciones anteriores se les pasa como parámetro un valor inválido, el intérprete mostrará un error.

```
# Ejemplo
>>> edad="25"
>>> edad = int(edad) + 10    # Convierte edad a int
>>> edad    # edad es un int
35
>>> edad = str(edad)    # Convierte edad a str
>>> edad    # edad es un str (se muestran las '')
'35'
>>> loat('18.66')    # Convierte un str a float
18.66
>>> float('hola')    # Convierte un str a float (pero no es válido)
Traceback (most recent call last):
File "<input>" , line 1, in <module>
ValueError: could not convert string to float: 'hola'
```

Entrada y salida de datos con Python

Introducción:

Es importante para un programador definir los datos de entrada y el formato adecuado para ellos. Igualmente es valioso saber cómo realizar cálculos y aplicar funciones predefinidas por Python, para generar unos resultados que deben ser entregados al usuario en un formato legible y claro.

En este componente formativo se explicará paso a paso cómo codificar la entrada y salida de datos, el uso de instrucciones secuenciales para manejo de constantes y variables y la aplicación de funciones definidas por el lenguaje. Además, se detallará el procedimiento para importar las librerías disponibles de Python con funciones de fecha, de números aleatorios y de matemáticas.

1. Entrada de datos

Cuando se usa la consola o el terminal, es común solicitar al usuario introducir datos a través del teclado. A continuación, se detallará la codificación para realizar la entrada de información por consola.

1.1 Entrada estándar

Para solicitar al usuario que introduzca algún dato a través del teclado, se debe usar el método **input().**

Este método recibe como parámetro un mensaje al usuario entre comillas:

```
Ejemplo
>>> edad = input("¿Cuántos años tienes?")
¿Cuántos años tienes? 28
>>> edad
'21'
```

Otra forma de realizar la entrada de datos a través del teclado sería:

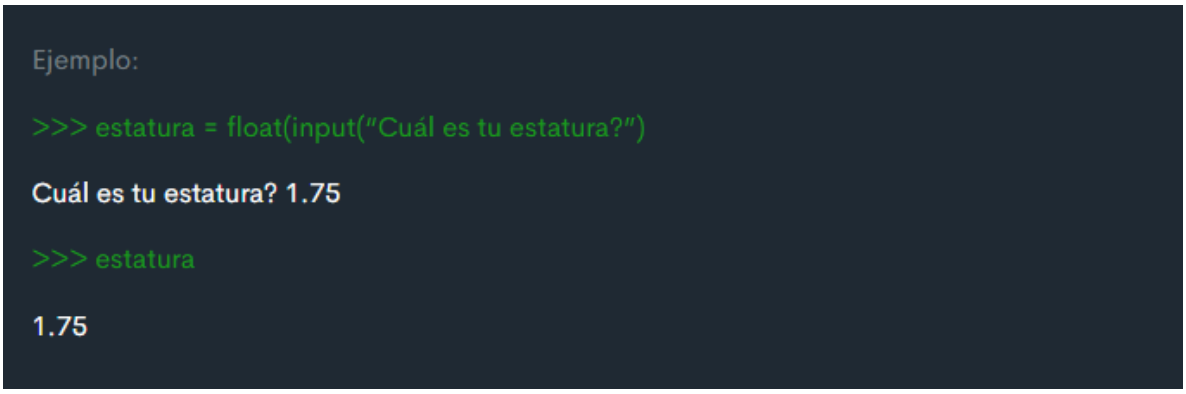
```
Ejemplo:
print("¿Cómo se llama?")
nombre = input()
print("Usted", nombre, "es un aprendiz SENA: " )
```

La entrada siempre es tipo cadena de caracteres **(str)**. Esto es útil para la entrada de datos tales como nombre, ciudad, cargo, deporte, etc.

Por tanto, si lo que se necesita es un dato de cierto tipo especial, por ejemplo, un **int**, habrá que hacer la conversión correspondiente en **input**, de esta manera:

```
Ejemplo:
>>> celular = int(input("Danos tu número de celular:"))
Danos tu número de celular: 3125320125
>>> celular
3125320125
```

Pero si lo que se requiere es un dato de tipo ***float***, habrá que hacer la siguiente conversión:



1.2 Entrada por script

Hasta ahora se ha escrito código directamente en la consola del intérprete, pero es necesario aprender a realizar programas informáticos que contengan todas las instrucciones en archivos llamados *scripts* (o guiones de instrucciones). Luego se envía este archivo al intérprete desde la terminal y se ejecuta todo el bloque de instrucciones. Guzdial y Vidal (2013) recomiendan aplicar buenas prácticas en programación.

De esta forma se pueden realizar todas las variaciones deseadas, sencillamente modificando el bloque de instrucciones almacenadas en un archivo con **extensión .py**

IDE para Python

Existen múltiples IDE (Entornos de Desarrollo Integrado o *Integrated Development Environment*), los cuales pueden usarse para digitar bloques de código en lenguaje Python.

Según Salazar (2019), un IDE consta de un editor de código fuente, un resaltador de sintaxis, unas herramientas de construcción automáticas y un depurador. La mayoría de los IDE tienen auto-completado inteligente de código.

Entre los IDE más utilizados para Python tenemos: *Sublime Text, PyCharm, Atom, Pythonista, Eclipse, Komodo, CodePad, VIM y Spyder Python*.

Utilizar el IDLE (*Integrated DeveLopment Environment for Python*) el cual es un entorno gráfico que permite editar, guardar y ejecutar programas en Python.

IDLE es también un entorno interactivo similar a una consola, en el que se pueden ejecutar instrucciones directas de Python.

En Windows, el IDLE viene junto con el intérprete de Python, es decir, al instalar Python en Windows también se instala el IDLE.

El IDLE aplica color al texto de acuerdo con su sintaxis. Los colores facilitan identificar los distintos tipos de datos y permiten detectar errores:

Las palabras reservadas de Python se muestran en color naranja.
Las cadenas de texto se muestran en verde.
Las funciones se muestran en púrpura.
Los resultados de las instrucciones se escriben en azul.
Los mensajes de error aparecen en rojo.

Para guardar el programa se selecciona File Save o se oprimen CTRL+S. El programa queda almacenado con el nombre deseado y la extensión de Python que es **.py**.

Para ejecutar el programa digitado, seleccionar en el ***menú Run > Run Module*** u oprimir la tecla F5.

2. Salida de datos

Para lograr la salida de datos se pueden combinar textos y variables separados con comas. Es una buena práctica acompañar esta información con mensajes claros al usuario, de manera tal que este identifique el tipo de dato generado y la unidad de medida en la cual se expresa.

Ahora se detallará la sintaxis para codificar la salida de información por pantalla.

2.1 Sentencia print

Para mostrar datos por pantalla, con frecuencia se utiliza la función **print**.

La sintaxis básica de **print** para presentar un mensaje es mencionando directamente el mensaje o almacenando el mensaje en una variable de texto y luego imprimir la variable.

Dentro del mensaje se pueden utilizar los siguientes caracteres especiales:

\n = salto de línea. '\n' termina la línea y mueve el cursor a otra línea.

\t = es el carácter "tab horizontal" y se usa para simular sangrías de texto.
Una forma de colocar varias cadenas de texto o intercalarlas con variables es colocarlas separadas por comas, el lenguaje incluye un espacio entre ellas.

Ejemplo:

```
print ("El salario devengado por ", empleado, "es la suma de ", salario)
El salario devengado por MARIA es la suma de 234500
print("El salario devengado por", empleado, "\nes la suma de $", salario)
El salario devengado por MARIA es la suma de $ 234500
print("El salario devengado por \t", empleado, "\tes la \tsuma de $", salario)
El salario devengado por MARIA es la suma de $ 234500
```

2.2 Salida de datos con formato

Python es compatible con la salida de datos con formato. El caracter módulo % es un operador integrado en Python. Este es conocido como el operador de interpolación. Se deben digitar los formatos deseados, el signo % (porcentaje), seguido por paréntesis con los datos que necesitan ser convertidos.

La sintaxis de la instrucción es:

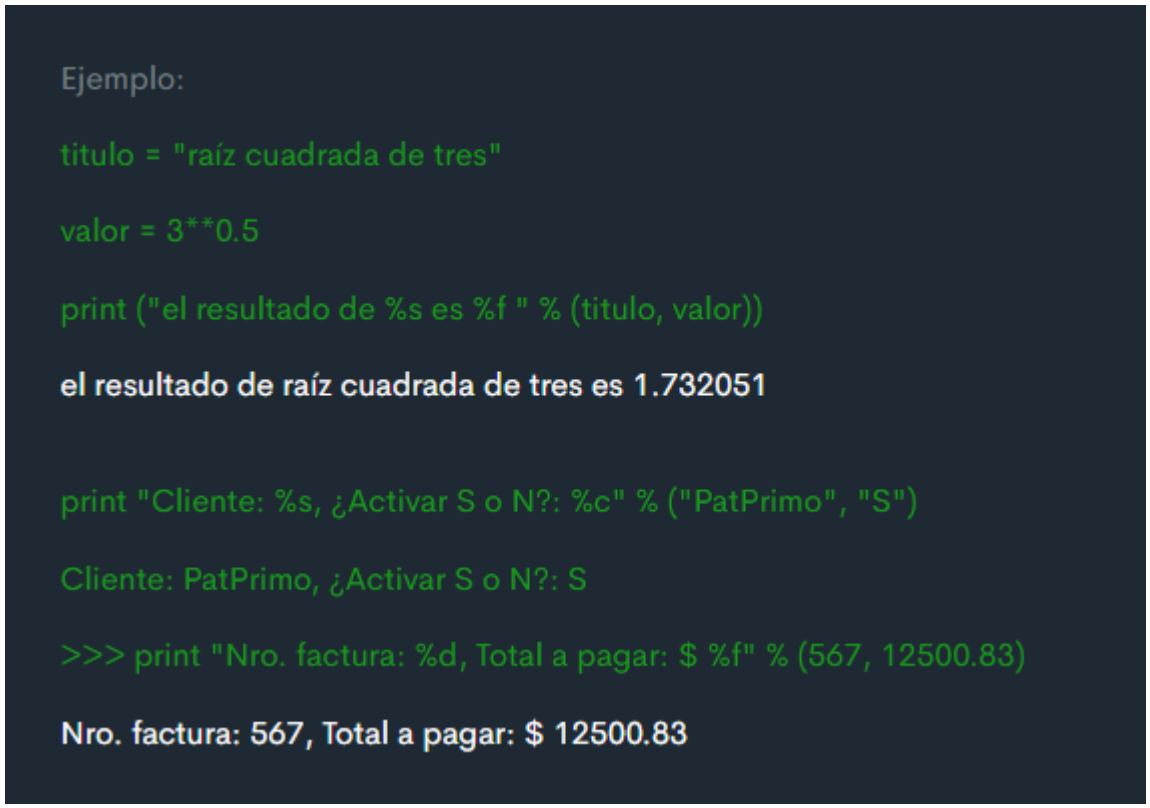
print ("cadena con formato" % (variables separadas por comas))
Se utilizan los siguientes formatos para la salida de datos:

%c = un caracter

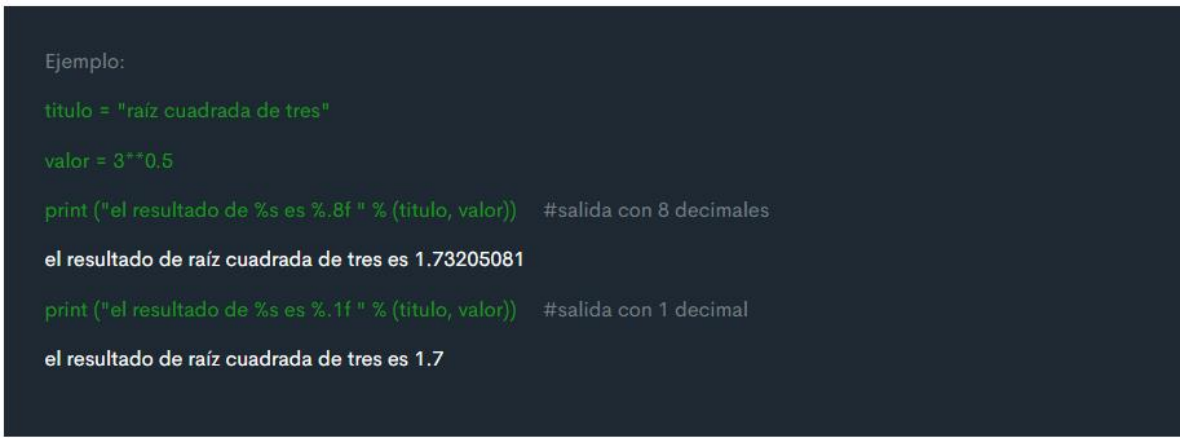
%s = str, cadena de caracteres

%d = int, enteros

%f = float, flotantes



Es posible controlar el formato de salida, por ejemplo, para obtener el valor con ocho (8) dígitos después del punto decimal, se digita %.8f; para una salida con 2 decimales se digita %.2f



2.3 Impresión de cadenas

Las cadenas tienen varias formas de ser impresas en pantalla. Veamos el siguiente cuadro:

frase1= “El perro”

frase2= “Manchas”

frase3= “ladra en la oscuridad”

Notación	Uso	Ejemplo	Salida en pantalla
+	Concatena cadenas	frase1+” “ + frase2+ “ “ + frase3 + “?”	El perro Manchas ladra en la oscuridad?
*	Repite cadenas	cadena4= frase2 *3	ManchasManchasManchas
cadena.capitalize ()	Inicia con mayúscula	frase3.capitalize()	Ladra en la oscuridad
cadena.center(ancho)	Centra en el ancho dado.	frase1.center(18)	“ El perro “
cadena.center(ancho)	Centra en el ancho dado.	frase1.center(18)	“ El perro “
cadena.lower()	Pasa todo a minúscula	frase2.lower()	manchas
cadena.upper()	Pasa todo a mayúscula	frase2.upper()	MANCHAS
cadena.title()	Mayúsculas iniciales	frase3.title()	Ladra En La Oscuridad

3. Instrucciones secuenciales con Python

Una vez leídos los datos de entrada, se debe proceder a realizar los cálculos necesarios para obtener la información requerida por el usuario. Estas instrucciones se deben digitar una tras otra, verificando que el procedimiento lógico sea correcto y los operadores empleados sean los adecuados. Es indispensable cumplir con la sintaxis aceptada por Python.

En la codificación de las instrucciones se pueden utilizar comentarios, constantes, variables, asignaciones, operadores aritméticos y funciones.

3.1 Comentarios en Python

Ortega (2018) define un comentario en Python como una línea de texto no ejecutable, es decir, no es una línea de código.

Es una buena práctica en programación documentar el código para mayor claridad. Se pueden hacer comentarios en Python para ilustrar acerca de lo que hace el programa. Se pueden incluir tantos comentarios como se requieran.

Hay dos formas para hacer comentarios:

Digitando el símbolo # al comienzo del comentario o dentro de la línea de código.

Digitando triple comilla **(“)** al principio y al final del comentario, cuando este ocupa varias líneas

3.2 Instrucciones de asignación

Una variable es una forma de identificar un dato que se encuentra almacenado en la memoria del computador. El valor de la variable puede ir cambiando a medida que se ejecutan las instrucciones del programa.

Nombres de variables válidos en Python:

Secuencias de letras y dígitos.

El nombre de la variable debe iniciar con una letra.

Se puede usar el guion bajo (_).
A las variables se les puede asignar un valor de cualquier tipo (int, float, str, booleano, lista) o una expresión aritmética.

En la siguiente tabla se verán varios operadores aritméticos que pueden ser utilizados en una asignación:

OPERADORES ARITMÉTICOS			
Si a=8 y b=5 los resultados serán:			
Operador	**Función**	**Ejemplo**	**Resultado**
+	Sumar	c = a + b	13
-	Restar	c = a - b	3
*	Multiplicar	c = b*2	10
/	Dividir	c = a / b	1.6
%	Módulo o residuo de la división	c = a%b	3
**	Potencia: x elevado a la potencia y	c = a ** b	85=32768
//	Devuelve la parte entera del cociente	c = a //b	1

3.3 Inicialización de variables

Algunas variables requieren ser inicializadas con un valor, usualmente ese valor es cero.

Cuando asignamos un valor a una variable por primera vez, se dice que se define e inicializa la variable. En un programa Python se pueden definir las variables en cualquier lugar del programa. Sin embargo, es una buena práctica definir las variables al principio del programa.

Para asignar un valor a una variable se utiliza el operador de asignación =.

No es necesario declarar inicialmente una variable con un tipo de datos; al asignar un valor a una variable, se declara e inicializa la variable con ese valor.

```
# Ejemplo:
b= 100    #b se inicializa con un valor entero
c= b**3    #c se inicializa con una expresión
print(c)
```

Si intentamos usar una variable que no ha sido inicializada, el intérprete mostrará un error:

```
# Ejemplo:
>>> print(z)
Traceback (most recent call last):
File "<input>", line 1, in <module>
NameError: name 'z' is not defined
```

3.4 Uso de constantes y variables

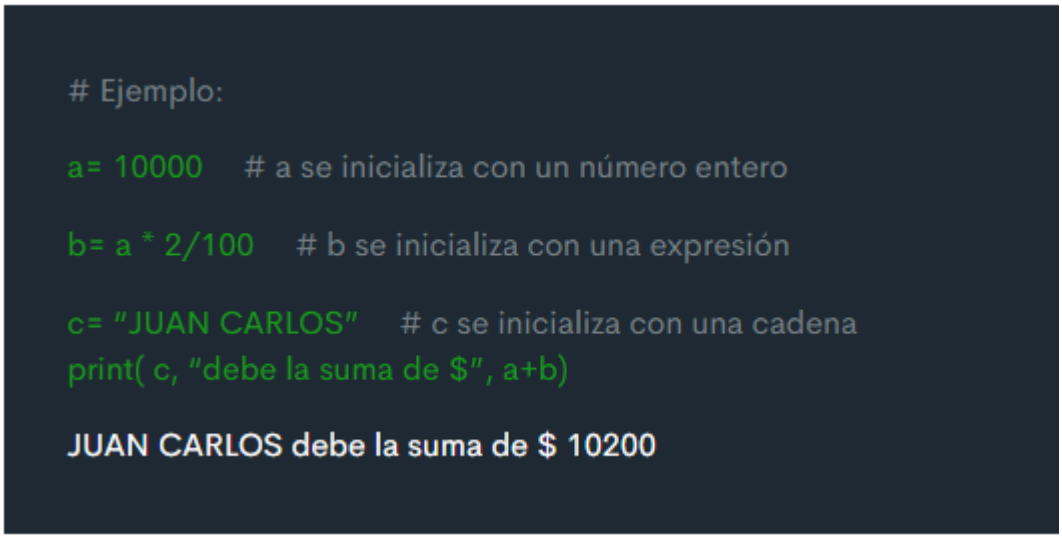
Una constante es un campo de memoria donde su valor o contenido no cambia: la única diferencia es que las constantes van en mayúscula, por ejemplo, CONSTANTE1. Desde el punto de vista funcional no existe ninguna diferencia entre las constantes y variables en Python. Es recomendable, como buena práctica, que no se modifique el valor de una constante posteriormente.

```
# Ejemplo:
IVA = 0.19
precioinicial = 3000
preciofinal = precioinicial * (1.0+IVA)
print(preciofinal)
3570
```

Una **variable** es una unidad de datos que puede cambiar de valor. El programador puede decidir el nombre de las variables, se recomienda que sea claro y conciso, y también puede escribir funciones que trabajen con los valores contenidos en ellas.

Para modificar el valor de una variable, simplemente se le asigna un nuevo valor en el programa, este puede ser un literal, una expresión, una llamada a una función o una combinación de todos ellos.

Un literal es un dato puro, que puede ser un número, una cadena de caracteres o un booleano.



Es posible en Python asignar un mismo valor a múltiples variables a la vez. Se pueden definir varias variables con un mismo dato, así:

```
a = b = c = 10    # Inicializan a, b y c con el valor entero 10
>>> print(a)
10
>>> print(b)
10
>>> print(c)
10
```

También está permitido, en una sola instrucción, inicializar varias variables con un valor diferente para cada una:

```
>>> a, b, c = 100, 200, 300
>>> print(a)
100
>>> print(b)
200
>>> print(c)
300
```

Existen unas variables especiales denominadas acumuladores y otras denominadas contadores.

Un **acumulador** es una variable a la que le vamos sumando un determinado valor.

Ejemplo: suma=suma+edad

La variable suma, acumula las edades.
Un **contador** es una variable a la cual le vamos sumando/restando un valor constante.

Ejemplo: total=total+1

La variable total se incrementa en 1 cada vez que se ejecuta esta instrucción.

3.5 Funciones predefinidas de Python

Existe una diversidad de funciones predefinidas en Python o funciones internas. Buttu (2016) menciona algunas de las funciones más utilizadas y conocidas:

Funciones de cadenas en Python			
Función	**Utilidad**	**Ejemplo**	**Resultado**
len()	Calcula la longitud en caracteres de una cadena.	len("Hola Mundo")	10
split()	Convierte una cadena en una lista.	a = ("El mundo está en aislamiento preventivo") listab = a.split() print (listab)	['El', 'mundo', 'está', 'en', 'aislamiento', 'preventivo']
lower()	Convierte una cadena en minúsculas.	texto = "Mario eS mI Aprendiz" texto.lower()	'mario es mi aprendiz'
upper()	Convierte una cadena en mayúsculas.	texto = "Mario es mi aprendiz" texto.upper()	'MARIO ES MI APRENDIZ'
replace()	Reemplaza una cadena por otra.	texto = "Mario es mi aprendiz" print (texto.replace ('aprendiz, 'estudiante'))	'Mario es mi estudiante'

Funciones numéricas en Python			
Función	**Utilidad**	**Ejemplo**	**Resultado**
range()	Crea un rango de números	x = range (6) print (list(x))	[0, 1, 2, 3, 4, 5]
str()	Convierte un valor numérico a texto	str(100)	'100'
int()	Convierte a valor entero	int('89')	89
float()	Convierte un valor a decimal	float('3.14')	3.14
max()	Determina el máximo entre un grupo de números	x = [1, 3, 2] print (max(x))	3
min()	Determina el mínimo entre un grupo de números	x = [1, 3, 2] print (min(x))	1
sum()	Suma el total de una lista de números	x = [1, 3, 2] print (sum(x))	6

Otras funciones en Python			
Función	**Utilidad**	**Ejemplo**	**Resultado**
list()	Crea una lista a partir de un elemento	x = range (4) print (list(x))	[0, 1, 2, 3]
ord()	Devuelve el valor ASCII de una cadena o carácter.	print(ord('C'))	67
round()	Redondea después de la coma de un float	print (round(15.92))	16

3.6 Librerías de fecha, random y matemáticas

En Python se pueden utilizar funciones predefinidas para manejar fechas, generar números aleatorios y aplicar funciones matemáticas, importando algunas librerías, las cuales, según Caballero (2019), se pueden aplicar en Big Data.

A. Librerías de fecha

Si se necesita usar fechas en un programa, es necesario importar las librerías Python: ***datetime y date***

```
from datetime import date

from datetime import datetime

hoy = date.today()    #fecha actual

print("Hoy es el dia: ", hoy)

Hoy es el día: 2020-10-28

fecha = datetime.now()    #Fecha actual con hora completa

print("La fecha completa de hoy es: ",fecha)

La fecha de hoy es: 2020-10-28 17:42:17.961801
```

B. Librerías random

Si se requiere generar números aleatorios (randómicos), es necesario cargar la librería random.

Aquí se presentan unos ejemplos de las posibles funciones con números aleatorios:

```
import random
a= random.randint(10,100)    #genera un número entero aleatorio entre 10 y 100
print (a)
99

b= random.randrange(0,100,5)    genera un número entero aleatorio entre 0 y 100 incrementos de 5
print (b)
25

c= random.random()    #genera un número float aleatorio entre 0.0 y 1.0
print (c)
0.998394986213598

d= random.uniform(100, 200)    #genera un número float aleatorio entre 100.0 y 200.0 inclusive.
print (d)
156.5039706179512
```

La función ***choice*** permite seleccionar al azar un dato desde un conjunto.

```
Import random
naipes = [1, 2, 3, 4, 5, 6, 7, 10, 11, 12]
random.shuffle(naipes)    #los naipes se barajaron al azar
print(naipes)
[3, 6, 2, 12, 5, 1, 4, 11, 10, 7]
```

La función ***sample.*** Esta función extrae una cantidad de números aleatorios de un conjunto.

```
Import random
naipes = [1, 2, 3, 4, 5, 6, 7, 10, 11, 12]
random.sample(naipes, 3)    #se tomaron 3 cartas del total de naipes
print(naipes)
[4, 11, 3]
```

C. Librerías matemáticas

Cervantes (2017) hace referencia a unas funciones matemáticas que vienen predefinidas por Python y otras que requieren ser importadas de la librería ***math***.

La función ***trunc()*** elimina los decimales de un número ***float***.

```
import math
a=123.56
b=math.trunc(a)
print("El valor truncado es: ",b)
123
```

La función ***fabs()*** calcula el valor absoluto de un número float, eliminando el signo

```
import math
a= - 200.45
b=math.fabs(a)
print("El valor absoluto es: ",b)
200.45
```

La función ***factorial()*** calcula el número de combinaciones posibles de una serie de objetos. El factorial se expresa como n!. ejemplo: 0! == 1. Es importante aclarar que la función factorial() solo se utiliza con números enteros.

```
import math
a= 6
b=math.factorial(a)
print("El valor factorial de ",a, " es: ",b)    # 6! es igual a 6x5x4x3x2x1
720
```

La función ***fmod()*** calcula el residuo de una división entre dos números float.

```
import math
c= math.fmod(16,5)
print("el residuo de dividir 16 entre 5 es ",c)
el residuo de dividir 16 entre 5 es 1.0
```

La función ***sqrt()*** calcula la raíz cuadrada de un número entero.

```
import math
a=3
c= math.sqrt(a)
print("la raíz cuadrada de 3 es: ",c)
la raíz cuadrada de 3 es: 1.7320508075688772
```

Las funciones trigonométricas seno, coseno y tangente se realizan usando ***sin(), cos() y tan()***.

Las funciones trigonométricas en la librería math toman los valores de los ángulos expresados como radianes. Por esta razón, debe realizarse la conversión de grados a radianes con la función ***radians***.

```
import math

angulo= 60

radianes = math.radians(angulo)

print("los radianes son:",radianes)

los radianes son: 1.0471975511965976


seno= math.sin(radianes)

coseno= math.cos(radianes)

tangente=math.tan(radianes)


print("el seno es:",seno)

el seno es: 0.8660254037844386

print("el coseno es:",coseno)

el coseno es: 0.5000000000000001

print("la tangente es:",tangente)

la tangente es: 1.7320508075688767
```

Existen muchas funciones matemáticas adicionales tales como exponenciales, hiperbólicas, trigonométricas inversas, distancia entre coordenadas, entre otras.

ESTRUCTURAS CONDICIONALES:

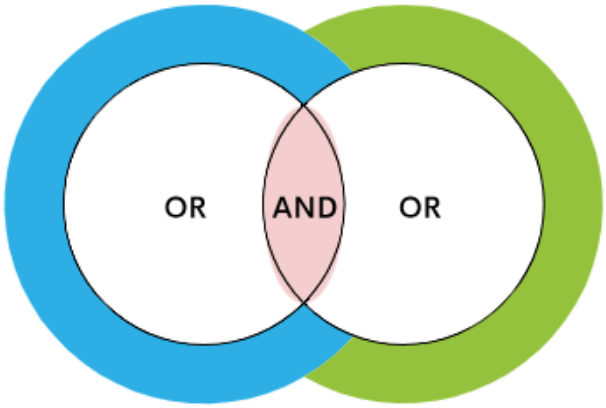
Introducción

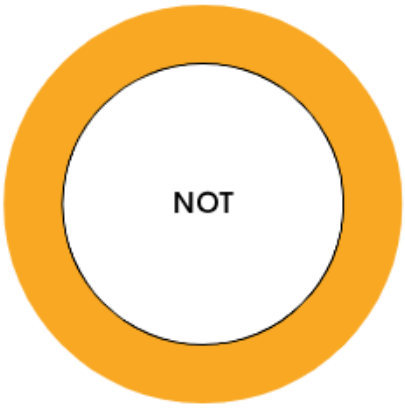
Las sentencias condicionales hacen que la programación tenga gran importancia. Esta es la capacidad de comparar el valor de una variable contra otra variable o una constante y ejecutar un bloque de instrucciones si se cumple una condición o de otra si no se cumple.

En este componente formativo se explicará el uso de las sentencias condicionales simples y las sentencias condicionales anidadas, utilizando expresiones booleanas simples y compuestas.

Existen operadores booleanos (también llamados lógicos) y operadores de comparación.

En Python, los tres operadores booleanos son: **“and” (y), “or “(o), “not”(no)**.





Para Buttu(2016) una sentencia u operación lógica puede ser evaluada Verdadera o Falsa de la siguiente manera:

El operador and genera como resultado el valor True, solamente cuando son ciertos sus dos operandos.

El operador or da como resultado el valor True, si cualquiera de sus operandos es True, y False sólo cuando ambos operandos son False.

El operador not es unario, y proporciona el valor True si su operando es False y viceversa.

and		
Operando		Resultado
Izquierdo	Derecho	
True	True	True
True	False	False
False	True	False
False	False	False

or		
Operando		Resultado
Izquierdo	Derecho	
True	True	True
True	False	True
False	True	True
False	False	False

not	
Operando	**Resultado**
True	False
False	True

Salazar(2019) menciona que es importante conocer las prioridades existentes al usar los operadores lógicos: en primer lugar se tiene el operador de negación **(not)**, luego el de conjunción ***(and)*** y por último el de disyunción ***(or)***.

En general, los elementos nulos o vacíos se consideran False y el resto se consideran ***True***.

Si se desea verificar si un elemento es ***True*** o ***False***, se puede convertir a su valor booleano mediante la función **bool()**.

```
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool("")
False
>>> bool(5)
True
>>> bool(3.2)
True
```

Operador	Descripción
>	**Mayor que**. (True) si el operando de la izquierda es estrictamente mayor que el de la derecha; (False) en caso contrario
>=	**Mayor o igual que.** (True) si el operando de la izquierda es mayor o igual que el de la derecha; (False) en caso contrario
<	**Menor que.** (True) si el operando de la izquierda es estrictamente menor que el de la derecha; (False) en caso contrario.
<=	**Menor o igual que.** (True) si el operando de la izquierda es menor o igual que el de la derecha; (False) en caso contrario.
==	**Igual.** (True) si el operando de la izquierda es igual que el de la derecha; (False) en caso contrario.
!=	**Distinto.** (True) si los operandos son distintos; (False) en caso contrario.

Ejemplo:

```
>>> 2==3
False
>>> 2>1
True
>>> 2>8
False
>>> 3>=4
False
>>> 8>=5
True
```

```
>>> 5!=6
True
```

Una sentencia condicional consta básicamente de los siguientes componentes:

Una prueba que evalúa verdadero o falso.
Un bloque de código que se ejecuta si la prueba es verdadera.
Un bloque opcional de código si la prueba es falsa.

1. **Condiciones Simples**

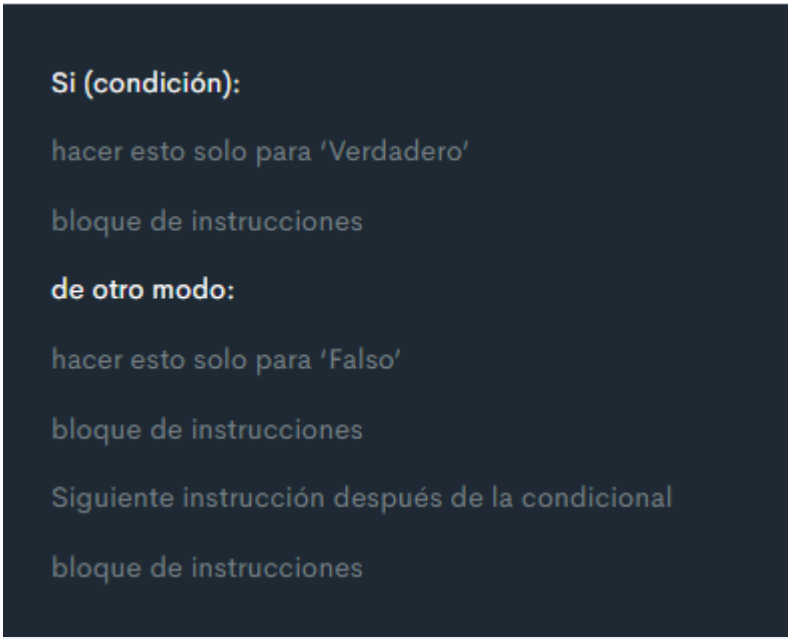
Cuando se evalúa una condición, existen dos respuestas posibles: verdadero (True) o falso (False). Si la condición es verdadera, el flujo del programa continúa en el bloque de instrucciones de la respuesta verdadera, de lo contrario, el programa continúa en el bloque de instrucciones de la respuesta falsa.

Una característica valiosa en la sintaxis de Python es el uso de la indentación, a diferencia de otros lenguajes como C que utilizan llaves { }. Salazar (2019) lo confirma en su libro.

La indentación consiste en dejar una sangría de cinco (5) espacios para señalar un bloque completo de instrucciones que se deben ejecutar, en caso que la sentencia condicional sea verdadera o falsa.

La indentación es un rasgo muy particular del código Python, y permite una lectura más agradable del programa y una fácil identificación de las distintas partes del programa.

En caso de un error de indentación, aparece el siguiente mensaje:



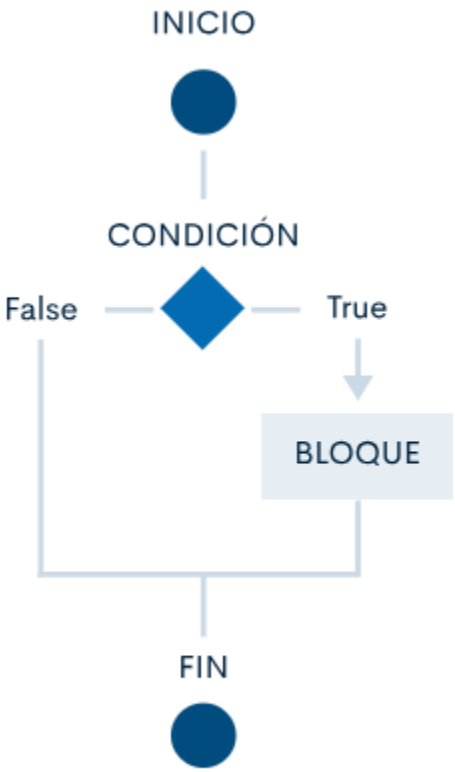
Programas ramificados:

En las instrucciones secuenciales, las sentencias se ejecutan en el orden en el que aparecen, en cambio, los programas ramificados permiten ejecutar las sentencias sin importar el orden, basándose en la toma de decisiones.

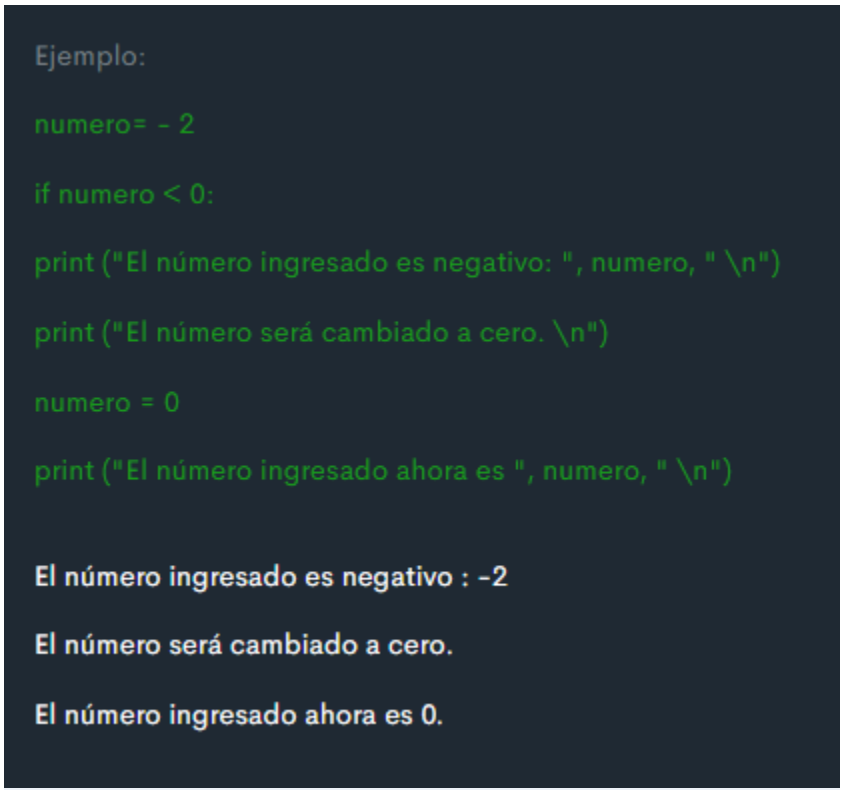
Según Marzal (2014), las sentencias condicionales se aplican dentro de este tipo de programación; si una sentencia condicional ha sido ejecutada, la ejecución del programa continúa en la siguiente línea de código condicional.

1.1 Uso de IF

La sentencia condicional ***if*** se usa para tomar una decisión. Esta sentencia realiza una operación lógica que debe dar como resultado True o False, y ejecuta el bloque de código siguiente, siempre y cuando el resultado sea verdadero.

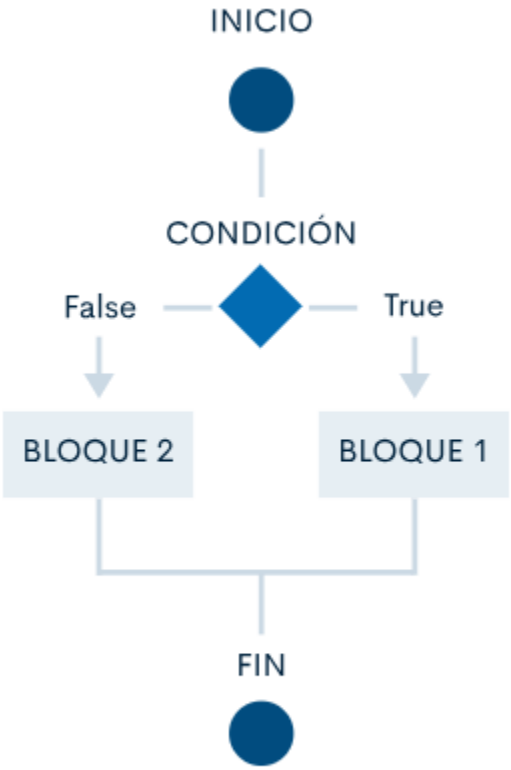


La sintaxis es la siguiente:



1.1 Uso de IF-ELSE

Una segunda forma de la sentencia condicional ***if*** es la ejecución con dos posibilidades. La condición, dependiendo si es verdadera o falsa, determina cuál de los dos bloques de instrucciones se ejecuta. El programa continúa después de la última instrucción con indentación.



La sintaxis es la siguiente:

```
if (condición):
    hacer esto solo para 'Verdadero'
    bloque de instrucciones
else:
    hacer esto solo para 'Falso'
    bloque de instrucciones
Siguiente instrucción después de la condicional
bloque de instrucciones
```

```
Ejemplo:

x=7

if x % 2 == 0:

print (x, "el número secreto es par")

else:

print (x, "el número secreto es impar")

print ("-------------------------")

print ("hemos terminado la verificación para este número")
```

La operación matemática (x % 2) calcula el valor del residuo de dividir un número por 2. El anterior código valida el valor del residuo, si el residuo es cero significa que se trata de un número par, de lo contrario, significa que es un número impar.

Este programa dará como resultado:

```
7 el número secreto es impar

-------------------------

hemos terminado la verificación para este número.
```

Ejemplo:

```
edad = int(input("¿Cuántos años tiene? "))
if edad < 99:
pass
else:
print("¡No creo que usted tenga esa edad!")
print("Usted dice que tiene {edad} años.")
```

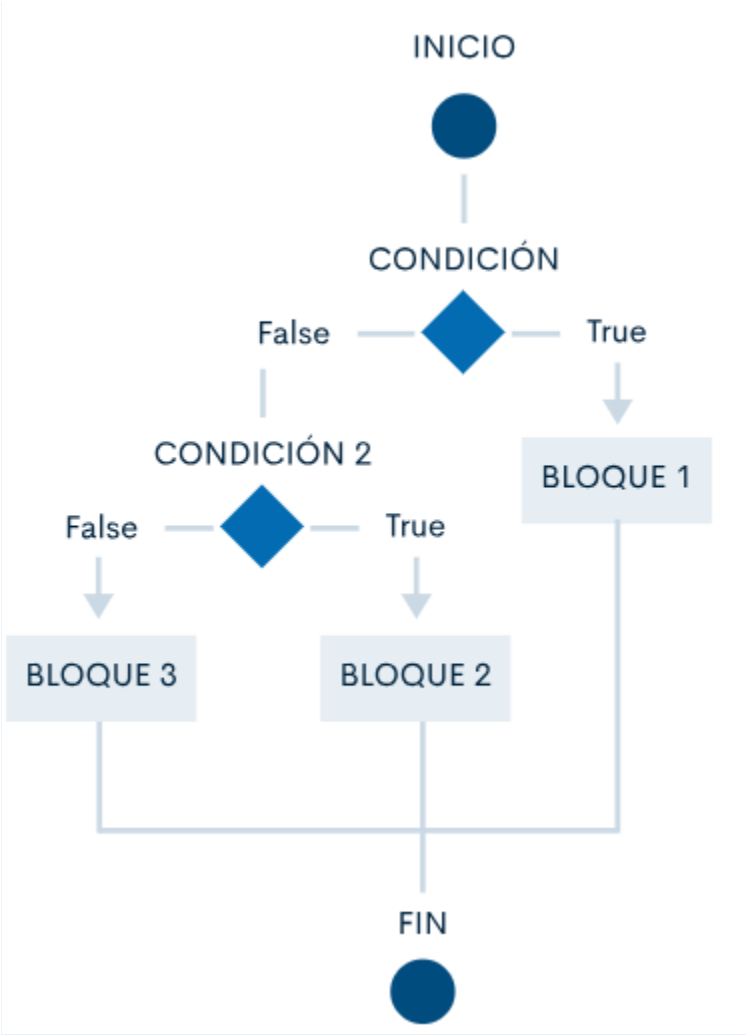
```
¿Cuántos años tiene? 130
¡No creo que usted tenga esa edad!
Usted dice que tiene 130 años.
```

2. Condicionales anidadas

En algunas ocasiones se requiere que después de una condicional se incluya una segunda condicional o quizás una tercera condicional. A continuación, se detallará la sintaxis para codificar varias condicionales anidadas en un solo bloque.

2.1 Concepto

Cervantes (2017) afirma que una sentencia condicional es anidada, si se observa que el bloque de código verdadero o el bloque de código falso, contiene otra sentencia condicional.


INICIO
CONDICIÓN
False
True
CONDICIÓN 2
BLOQUE 1
False
True
BLOQUE 3
BLOQUE 2
FIN

```
Ejemplo:
curso1="Requerimientos"
curso2="Algoritmos"

print("El curso1 es: ", curso1)
print("El curso2 es: ", curso2)
if curso1 == "Requerimientos":
   if curso2 == "Algoritmos":
      print ("Usted estudia Programación de Software")
   else:
      print ("Usted estudia otro programa diferente a Programación de Software")
```

La ejecución de este programa sería la siguiente:

```
El curso 1 es: Requerimientos
El curso 2 es: Algoritmos
Usted estudia Programación de Software
```

Aquí se observa una sentencia condicional anidada. La primera instrucción ***if*** contiene otra sentencia ***if***. Es valioso aplicar la indentación interna del código para que este sea legible y funcione correctamente.

2.1 Uso de ELIF

Existen también las llamadas **Condicionales encadenadas**. A veces hay más de dos posibilidades y necesitamos más de dos ramificaciones. Una forma de expresarlo es con una condicional encadenada, donde solo se puede cumplir una de las ramificaciones o ninguna de ellas.

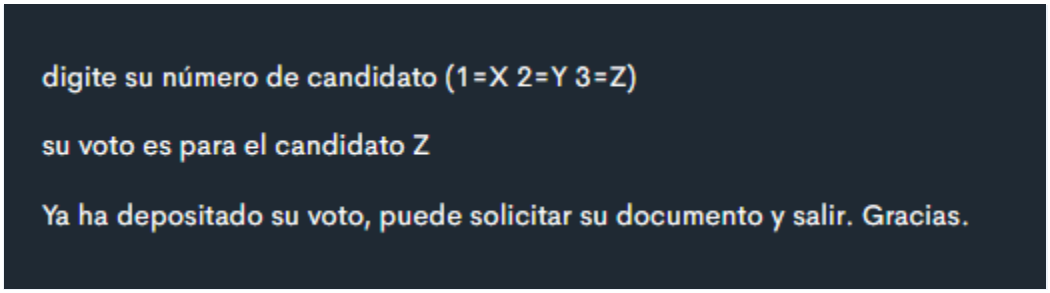
La sentencia **elif** es una abreviatura de ***"else if"***. No existe un límite definido al uso de sentencias **elif**, pero solo se permite una sentencia else (que es opcional) y debe ser la última rama de la sentencia:

```
if condición1:
    bloque de instrucciones para 'Verdadero' a la condición1
elif condición2:
    bloque de instrucciones para 'Verdadero' a la condición2
elif condición3:
    bloque de instrucciones para 'Verdadero' a la condición3
else:
    hacer esto solo para 'Falso' a todas las condiciones anteriores
    bloque de instrucciones
Siguiente instrucción después de la condicional
bloque de instrucciones
```

Aquí se presenta una aplicación de **elif**:

```
Ejemplo:
voto= int(input(" \n digite su número de candidato (1=X 2=Y 3=Z) "))
if voto==1 :
   print (" \n su voto es para el candidato X " )
elif voto==2 :
   print (" \n su voto es para el candidato Y " )
elif voto==3 :
   print (" \n su voto es para el candidato Z " )
else:
   print ("su voto es inválido.")
print (" \n Ya ha depositado su voto, puede solicitar su documento y salir. Gracias")
```

El resultado del anterior programa es:



Las condiciones se comprueban en orden. Si la primera es falsa, se comprueba la siguiente, y así se sigue con las demás. Si una de ellas es cierta, se ejecuta la rama correspondiente y termina la sentencia. Incluso, si es cierta más de una condición, sólo se ejecuta la primera rama verdadera.

Expresiones booleanas compuestas:

Fernández (2017) menciona que es posible tener más de una expresión booleana en la misma sentencia condicional. A esto se le llaman expresiones booleanas compuestas.

Un ejemplo: hallar el número más pequeño entre tres números enteros.

Se observa que el primer test es una expresión booleana compuesta. Todos los operandos tienen que ser ciertos para que la siguiente instrucción se ejecute.

```
Ejemplo:
a= int(input("Digita el primer numero entero "))
b= int(input("Digita el segundo numero entero "))
c= int(input("Digita el tercer numero entero "))
if a < b and a < c:
   print ("El menor numero entero es ", a)
elif b < c:
   print ("El menor numero entero es ", b)
else:
   print ("El menor numero entero es ", c)
print (" ******* END **********")
```

La ejecución del programa da como resultado:

```
Digita el primer número entero 7
Digita el segundo número entero 3
Digital el tercer número entero 9
El menor número entero es 3
******* END **********
```

Ciclos iterativos con Python

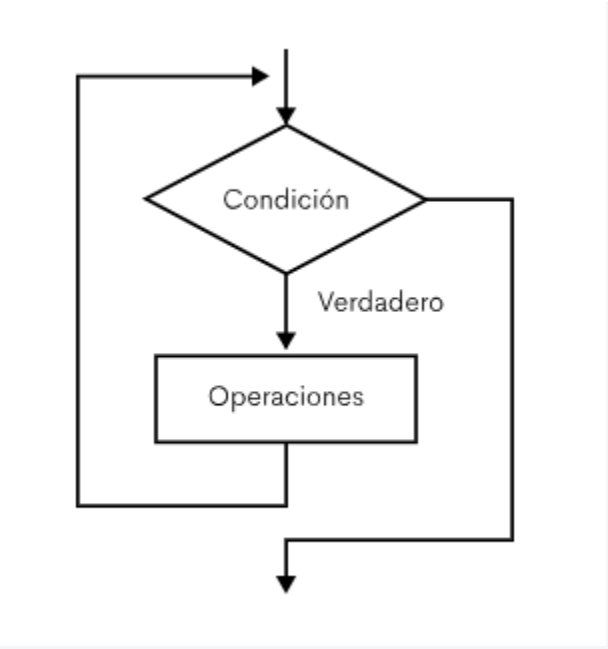
En la vida real, frecuentemente se requiere que un programa repita un bloque de instrucciones, hasta que, o mientras que, se cumpla una condición dada.

El aprendiz aprenderá a codificar estructuras de control iterativas, también llamados ciclos repetitivos o bucles, para resolver situaciones donde el usuario requiere que se realice, una y otra vez, una serie de instrucciones, hasta cumplir una determinada condición.

Introducción

Se denominan ciclos repetitivos o iterativos a aquellos que agrupan un bloque de instrucciones para ser ejecutado un determinado número de veces. Este número puede ser fijo (definido por el programador) o puede ser variable (dependiendo de algún valor o condición en el programa).

Se puede ver un ejemplo de un ciclo iterativo en el siguiente gráfico:



Mientras la condición sea Verdadera, se realizará el bloque de instrucciones, hasta que la condición sea falsa.

Para Marzal(2014), los ciclos iterativos en Python, usan dos instrucciones: el ciclo ***FOR*** y el ciclo ***WHILE***. Dependiendo de la situación que se deba resolver, se selecciona la instrucción mas adecuada a utilizar.

1. Ciclo FOR

La estructura ***for*** se usa en aquellos casos en los cuales se tiene certeza de la cantidad de veces que se desea ejecutar un bloque de instrucciones.

Ejemplo: leer 6 edades de estudiantes, ingresar 100 salarios de empleados.

El bloque de instrucciones que se repite se denomina cuerpo del bucle y cada repetición se denomina una iteración.

Esta estructura de control iterativa se utiliza en situaciones en las que se desea que una variable de control tenga un valor inicial y un valor final predefinidos. No es necesario definir la variable de control antes del bucle, aunque se puede utilizar una variable ya definida previamente en el programa.

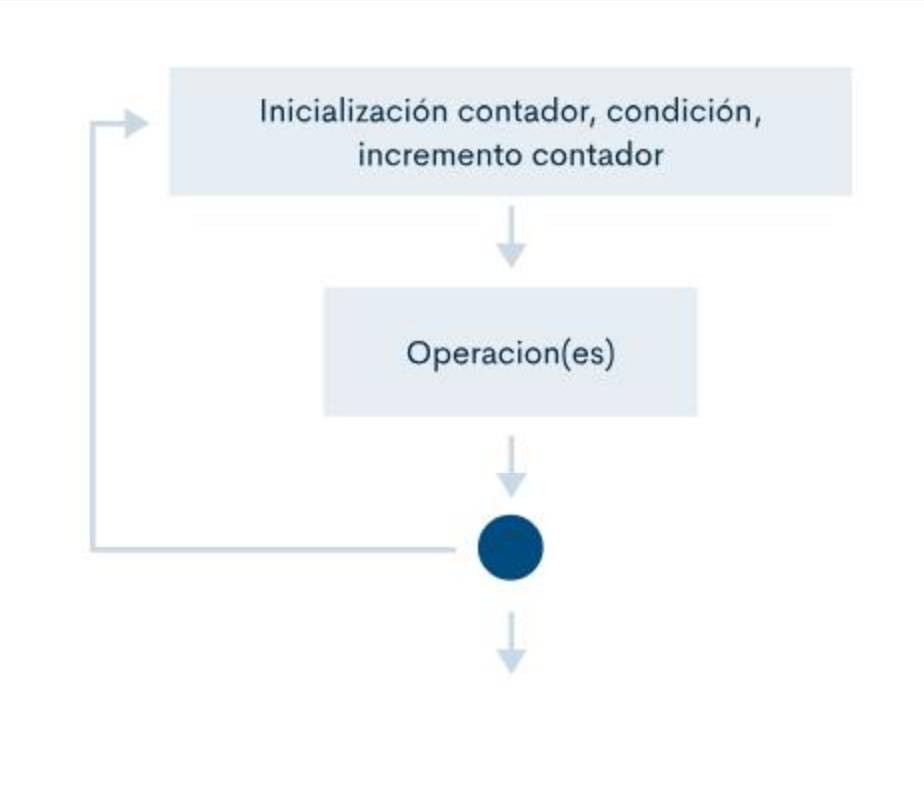
El bucle ***for*** se puede usar para recorrer los items de cualquier secuencia (cadena, lista, tupla, conjunto, diccionario) y ejecutar un bloque de código sobre ese item. En cada iteración se tiene acceso a un único elemento de la secuencia.

Ejemplo:

```
animales = ['leon', 'tigre', 'cocodrilo']
for animal in animales:
print ("El animal es: {0}, tamaño de palabra es: {1}".format(animal, len(animal)) )
```

```
El animall es: leon , tamaño de palabra es: 4
El animall es: tigre , tamaño de palabra es: 5
El animall es: cocodrilo , tamaño de palabra es: 9
```

La estructura del ciclo for es la siguiente:



1.1 Uso de RANGE

Según Pérez(2016) en el ciclo iterativo ***for*** se puede usar la función range, para indicar un rango de valores.

range(valor_inicial, valor_final)

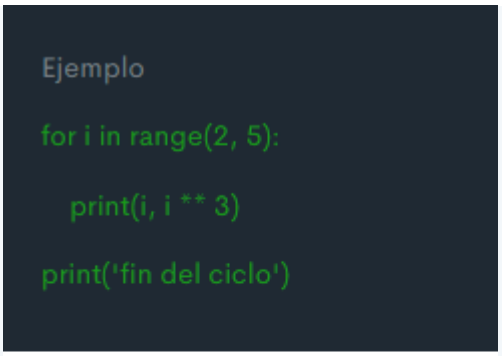
Es posible indicar solo un valor final, en caso tal, el ciclo inicia en 0.

Ejemplo

```
for i in range(5):
    print(i)
print('fin del ciclo')
```

```
0
1
2
3
4
fin del ciclo
```

Si se especifica el valor inicial y el valor final en la función range, el resultado imprimirá desde el valor inicial hasta el valor final - 1, así:




```
2 8
3 27
4 64
fin del ciclo
```

Para imprimir hasta el valor final, éste debe incrementarse en 1.

1.2 Conceptos: valor inicial, valor final, incremento.

Cuando se deseen especificar los valores inicial, final y el incremento o decremento de la variable de control en el ciclo iterativo for, se usa la función range con tres argumentos así:

range(valor_inicial, valor_final, incremento/decremento)

Cuando se omite, el incremento/decremento es implícitamente igual a 1.

El ciclo siempre incluye el **valor_inicial** y excluye el **valor_final** durante la iteración:

```
Ejemplo
for i in range(10, 0, -2):
    print(i)
print('fin del ciclo')
```

```
10
8
6
4
2
fin del ciclo
```

1.3 Usos y estructura

La sentencia **for** se puede usar para imprimir secuencias, tales como cadenas, listas, tuplas, conjuntos o diccionarios.

Uso del ciclo iterativo for en cadenas

```
Ejemplo
for character in "hola mundo":
    print(character)
print("fin del ciclo")
```

```
h

o

l

a


m

u

n

d

o

fin del ciclo
```

Uso de la estructura de control iterativa for en listas

```
Ejemplo:

nombre = "Alicia"

edad = 25

print(f"Me llamo {nombre} y tengo {edad} años.")


numeros = [0, 1, 2, 3, 4, 5]

for numero in numeros:

   print (numero, end=" " )

print("fin del ciclo")
```

```
Me llamo Alicia y tengo 25 años

0 1 2 3 4 5 fin del ciclo
```

Uso del bucle *for* en tuplas

```
Ejemplo

conexion_bd = ("115.16.10.5","root","7890","clientes")

print("la tupla es: ")

for parametro in conexion_bd:

   print (parametro)

print ("fin del ciclo")
```

```
la tupla es:

115.16.10.5

root

7890

clientes

fin del ciclo
```

Uso del ciclo iterativo *for* en conjuntos

Ejemplo

```
A={1, 2, 3, 5, 3, 6, 4, 2}
print ("Los elementos del conjunto A son:")
for elemento in A:
    print(elemento)
print ("fin del ciclo for")
```

```
Los elementos del conjunto A son:
1
2
3
4
5
6
fin del ciclo for
```

Uso del bucle *for* en diccionarios

```
Ejemplo
ensalada = {'Manzana':'verde', 'Patilla':'rosado', 'Banano':'amarillo', 'Papaya':'naranja'}
for nombre, color in ensalada.items():
   print (nombre, "es de color", color)
print ("fin del ciclo")
```

```
Manzana es de color verde
Patilla es de color rosado
Banano es de color amarillo
Papaya es de color naranja
fin del ciclo
```

Otra forma de hacerlo en un diccionario es:

```
Ejemplo
ensalada = {'Manzana':'verde', 'Patilla':'rosado', 'Banano':'amarillo', 'Papaya':'naranja'}
for elemento in ensalada:
   print (elemento, 'es de color', ensalada[elemento] )
print ("fin del ciclo")
```

```
Manzana es de color verde
Patilla es de color rosado
Banano es de color amarillo
Papaya es de color naranja
fin del ciclo
```

1.4 Sentencias BREAK, CONTINUE y PASS

Nolasco(2018) afirma que con frecuencia se requiere interrumpir un ciclo repetitivo, bien sea porque se cumple una condición y se desea salir inmediatamente de él o simplemente se requiere volver a realizar otra iteración sin ejecutar las demás instrucciones del bloque de código. Algunas veces el programador suspende la codificación de un ciclo iterativo y desea colocar un mensaje temporal. Para esto existen las sentencias *Break, Continue y Pass.*

BREAK

A veces los ciclos se vuelven infinitos, porque la condición siempre es verdadera. Existe una instrucción de Python, break, que permite salir de un ciclo en medio de su ejecución.

Ejemplo

```
Print ("Uso de la sentencia break")
for caracter in "PYTHON":
    if caracter == "N":
        break
    print ("El caracter actual es : " +caracter)
print ("fin del ciclo")
```

```
Uso de la sentencia break
el caracter actual es: P
el caracter acutal es: Y
el caracter actual es: T
el caracter actual es: H
el caracter actual es: O
fin del ciclo
```

CONTINUE

A veces se requiere en los ciclos iterativos que se ignoren las siguientes instrucciones y regresar al inicio del bucle, para continuar con una nueva iteración. Para solucionar esto existe la instrucción *continue*.

En el siguiente ejemplo, si la variable toma el valor de 9, salta al inicio e ejecuta otra iteración.

```
Ejemplo
print("Uso de la sentencia continue")
variable = 15
while variable > 1:
   variable = variable -2
   if variable == 9:
      continue
   print ("Valor actual de la variable : " + str(variable))
print ("fin del ciclo")
```

```
Uso de la sentencia continue
Valor actual de la variable: 13
Valor actual de la variable: 11
Valor actual de la variable: 7
Valor actual de la variable: 5
Valor actual de la variable: 3
Valor actual de la variable: 1
fin del ciclo
```

PASS

La sentencia pass, tal como lo dice su nombre (pasar), es una sentencia nula, o sea que no pasa nada cuando se ejecuta. Se utiliza pass cuando se requiere por sintaxis una instrucción pero no se quiere ejecutar ningún código. También se utiliza en programación donde donde el código irá a futuro, pero no ha sido escrito todavía, utilizándolo como un relleno temporal, es decir, se encuentra “en construcción”

Ejemplo

```
print("Uso de la sentencia pass")
for letra in "PYTHON SENA":
  if letra == "N":
    pass
  print ("El caracter actual es :" + letra)
print ("fin del ciclo")
```

```
Uso de la sentencia pass
El caracter actual es: P
El caracter actual es: Y
El caracter actual es: T
El caracter actual es: H
El caracter actual es: O
El caracter actual es: N
El caracter actual es:
El caracter actual es: S
El caracter actual es: E
El caracter actual es: N
El caracter actual es: A
fin del ciclo
```

1.5 Estructura *FOR - ELSE*

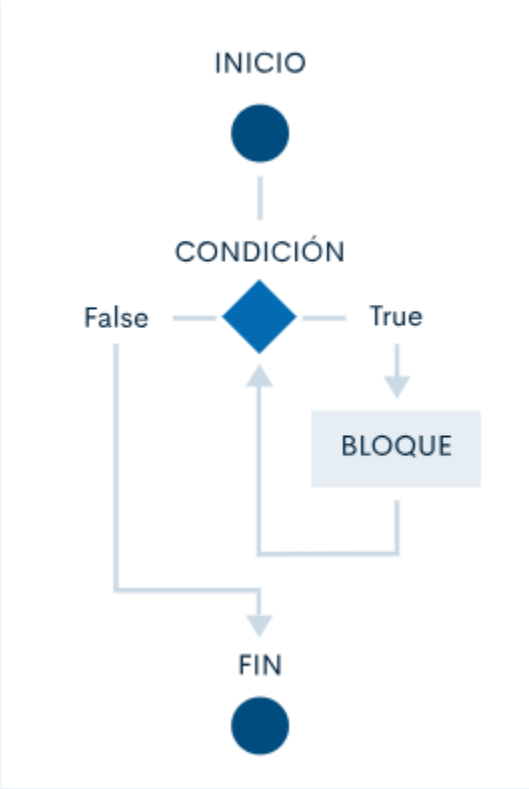
De forma similar a la sentencia ***if***, la estructura ***for*** también se puede combinar con una sentencia ***else***.

El bloque ***else*** se ejecutará cuando la expresión condicional del bucle for sea ***False***.

El siguiente ejemplo presenta una conexión a una base de datos de empleados:

```
Ejemplo
db_connection = "115.0.0.5","7890","root","empleados"
for parametro in db_connection:
    print parametro
else:
    print """El comando PostgreSQL es:
$ psql -h {server} -p {port} -U {user} -d {db_name}""".format(
    server=db_connection[0], port=db_connection[1],
    user=db_connection[2], db_name=db_connection[3])
```

2. Ciclo WHILE



Según Fernández(2016), un ciclo de control iterativo ***WHILE*** permite repetir la ejecución de un bloque de instrucciones mientras se cumpla una condición, es decir, mientras la condición tenga el valor ***True.***

2.1 Estructura WHILE

La estructura de la sentencia de control while es la siguiente:

```
while condicion:
    cuerpo del bucle
```

Las variables que se encuentran en la condición se denominan variables de control. Las variables de control se deben definir antes del ***while*** y modificarse dentro del ***while***.

Si el resultado es True se ejecuta el bloque de instrucciones del bucle. Una vez ejecutado el bucle, se repite el proceso una y otra vez mientras la condición sea verdadera.

Si el resultado es ***False***, no se ejecuta el bloque de instrucciones del bucle y continúa la ejecución en la siguiente instrucción después del bucle en el programa.

Una forma de incrementar la variable de control es usando el símbolo +=

```
Ejemplo
i = 1
while i <= 11:
    print(i)
    i += 2
print("Programa Finalizado")
```

Donde i += 2 significa que el valor de i se incrementa en 2 cada vez que se ejecuta el ciclo repetitivo.

```
1
3
5
7
9
11
Programa finalizado
```

Una ventaja adicional del bucle while es que el número de iteraciones lo puede definir el usuario durante el bucle. Por ejemplo, el programa solicita un número al usuario una y otra vez hasta que el usuario acierte.

```
Ejemplo
i = 1
numero1=100
numero2 = int(input("Digite un número de 1 a 100: "))
while numero2 != numero1:
   i += 1
   numero2 = int(input("Digite un número de 1 a 100: "))
print ("Acertaste en el intento No.", i)
```

```
Digite un número de 1 a 100: 5
Digite un número de 1 a 100: 8
Digite un número de 1 a 100: 10
Digite un número de 1 a 100: 100
Acertaste en el intento No: 4
```

2.2 Tipos de bucle WHILE

La sentencia ***while*** permite ejecutar ciclos, es decir, ejecutar un bloque de código múltiples veces.

El ciclo ***while*** realiza múltiples iteraciones con base en el resultado de una expresión lógica: *True o False*.

Para Zed(2014), existen 3 tipos de ciclos repetitivos con la sentencia while:

While controlado por contador.

While controlado por evento.

While con Else.
A continuación se detalla el uso de cada uno de ellos, dependiendo de la situación que deba resolver el programador.

Ciclo *while* controlado por contador
Es un ciclo donde existe una variable o contador con un valor inicial, el cual se incrementa / decrementa en cada iteración hasta que llega a un valor final definido en la condición incluida en la sentencia ***While***.

```
print ("Uso del ciclo while con contador")
total, valor = 0, 1
while valor <= 12:
   print (valor)
   total = total + valor
   print ("el total parcial es ",total)
   valor = valor + 1
print ("El total final es ",total)
print( "fin del ciclo while")
```

El resultado de este ciclo while controlado por contador es:

Uso del ciclo while con contador

1

el total parcial es 1

2

el total parcial es 3

3

el total parcial es 6

4

el total parcial es 10

5

el total parcial es 15

6

el total parcial es 21

7

el total parcial es 28

8

el total parcial es 36

9

el total parcial es 45

10

el total parcial es 55

11

el total parcial es 66

12

el total parcial es 78

El total final es 78

Fin del ciclo while

Ciclo *while* controlado por evento

Es un ciclo repetitivo ***while*** donde se activa un evento dentro del ciclo, el cual causa que el bucle se interrumpa.

```
print ("Uso del ciclo while controlado por evento")
promedio, total, contador = 0, 0, 0
print ("=== Software para parqueadero ===")
placa = input("Introduzca la placa del vehiculo (99 para salir): " )
while placa != "99":
   valor= float(input("Digite valor del parqueadero: "))
   total = total + valor
   contador= contador+1
   placa = input("Introduzca la placa del vehiculo (99 para salir): " )
promedio = round(total / contador)
print ("Promedio de vr. parqueadero por vehiculo: " + str(promedio))
print ("Total dinero recaudado: ", round(total))
print( "fin del ciclo while")
```

El resultado de este ciclo while controlado por evento es:

```
Uso del ciclo while controlado por evento
=== software para parqueadero ===
Introduzca la placa del vehiculo (99 para salir): XER45T
Digite valor del parqueadero: 2500
p Introduzca la placa del vehiculo (99 para salir): JEF56R
Digite valor del parqueadero: 1500
Introduzca la placa del vehiculo (99 para salir): 99
Promedio de vr. parqueadero por vehiculo: 2000
Total dinero recuadado: 4000
fin del ciclo while
```

Ciclo *while* controlado por evento

De manera similar a la sentencia ***if***, el ciclo de control iterativo ***while*** se puede combinar con la sentencia ***else***.

Pérez(2016), explica que la sentencia else ocurre cuando la expresión condicional del ***while*** es ***False***.

```
print ("Uso del ciclo while + else")
promedio, total, contador = 0, 0, 0
print ("=== Software para parqueadero ===")
placa = input("Introduzca la placa del vehiculo (99 para salir): " )
while placa != "99":
    valor= float(input("Digite valor del parqueadero: "))
    total = total + valor
    contador= contador+1
    placa = input("Introduzca la placa del vehiculo (99 para salir): " )
else:
    promedio = round(total / contador)
    print ("Promedio de vr. parqueadero por vehiculo: " + str(promedio))
    print ("Total dinero recaudado: ", round(total))
    print( "fin del ciclo while")
print("final del ciclo while + else")
```

El resultado de este ciclo ***while*** con ***else*** es:

```
Uso del ciclo while + else
=== software para parqueadero ===
Introduzca la placa del vehiculo (99 para salir): XER45T
Digite valor del parqueadero: 2500
p Introduzca la placa del vehiculo (99 para salir): JEF56R
Digite valor del parqueadero: 1500
Introduzca la placa del vehiculo (99 para salir): 99
Promedio de vr. parqueadero por vehiculo: 2000
Total dinero recuadado: 4000
fin del ciclo while
final del ciclo while + else
```

2.3 Sentencias BREAK, CONTINUE y PASS

Similar al ciclo iterativo ***for***, en el bucle con ***while*** existen existen las sentencias ***Break***, ***Continue*** y ***Pass***.

BREAK

Break termina el bucle actual y continúa con la ejecución de la siguiente instrucción después del ciclo.

```
print("Uso de la sentencia BREAK en while")
variable = 35
while variable > 1:
    variable = variable -5
    if variable == 10:
        break
    print ("Valor actual del caracter : " + str(variable))
print ("fin del ciclo")
```

```
Uso de la sentencia BREAK en while
Valor actual del caracter : 30
Valor actual del caracter : 25
Valor actual del caracter : 20
Valor actual del caracter : 15
fin del ciclo
```

CONTINUE

La sentencia ***Continue*** en Python, regresa el control de la ejecución del programa al inicio del bucle, desechando todas las instrucciones que quedan en la iteración actual del bucle e inicia una nueva iteración.

```
print("Uso de la sentencia CONTINUE en while")
variable = 35
while variable > 1:
    variable = variable -5
    if variable == 15:
        continue
    print ("Valor actual de la variable : " + str(variable))
print ("fin del ciclo")
```

```
Uso de la sentencia CONTINUE en while
Valor actual del caracter : 30
Valor actual del caracter : 25
Valor actual del caracter : 20
Valor actual del caracter : 10
Valor actual del caracter : 5
Valor actual del caracter : 0
fin del ciclo
```

PASS

La sentencia ***Pass*** se utiliza cuando se requiere por sintaxis una instrucción pero no se requiere ejecutar ningún comando o código.

```
print("Uso de la sentencia PASS en while")
variable = 35
while variable > 1:
    variable = variable -5
    if variable == 25:
        pass
    print ("Valor actual de la variable : " + str(variable))
print ("fin del ciclo")
```

```
Uso de la sentencia PASS en while
Valor actual del caracter : 30
Valor actual del caracter : 25
Valor actual del caracter : 20
Valor actual del caracter : 15
Valor actual del caracter : 10
Valor actual del caracter : 5
Valor actual del caracter : 0
fin del ciclo
```

```
from datetime import date
hoy = date.today()
print("Hoy es el día: ", hoy)
print()

print("EJERCICIO DE LAS CLASES DE TRIANGULOS")

a=int (input("digite el valor de A: "))
b=int (input("digite el valor de B: "))
c=int (input("digite el valor de C: "))

if a==b and a==c and b==c:
    print("EL TRIANGULO ES EQUILATERO")
else:
    if a==b or b==c or a==c:
        print("EL TRIANGULO ES ISOCELES")
    else:
            print("EL TRIANGULO ES ESCALENO")
print()
animal =input("digite un animal: ")
animal= animal.upper()
if animal== "PERRO":
    print ("Este animal es el mejor amigo del hombre:", animal)
elif animal =="GATO":
    print("Este animal persigue a los ratones:", animal)
elif animal=="OSO":
    print("Este animal vive en el polonorte: ", animal)
else:
    print("No es PERRO, no es GATO, ni es OSO... es  ", animal)
print()
print("FIN")
```

```
from datetime import date
hoy = date.today()
print("Hoy es el día: ", hoy)
print()

print("TALLER 5 CICLOS ITERATIVOS CON LA SENTENCIA FOR")
print()
num1=int(input("digite el primer número: "))
num2=int(input("digite el segundo número (mayor): "))
print("ciclo para el primer número")

for i in range (num1):
    print(i)
print('fin del clclo')

print()
print("ciclo dasde el primer número hasta el segundo número")
for i in range (num1,num2):
    print(i)
print('fin del ciclo')

print()
print("ciclo desde el primero hasta el segundo con incrementos de 2")

for i in range (num1,num2, 2):
    print(i)
print('fin del ciclo')

print()
empresa=input("digite nombre de una empresa:")
empresa= empresa.lower()
for character in empresa:
    print(character)
print("fin del ciclo")

print()
print("FIN")
```

```
from datetime import date
hoy = date.today()
print("Hoy es el día: ", hoy)
print()
print("TALLER 6 CICLOS ITERATIVOS CON LA SENTENCIA WHILE")
print()
num1=int(input("digite un número: "))
print("***Ciclo controlado por contador***")
i=1
while i<= num1:
    print(i)
    i+=1
print('fin del ciclo')

print()
print("***Ciclo controlado por evento***")
i=1
numero1=5
numero2=int(input("Digite un número de 1 a 10: "))
while numero2 != numero1:
    i+=1
    numero2 = int(input("Digite un número de 1 a 10: "))
print ("Acertaste en el intento No.", i)
print('fin del ciclo')

print()
print("***Ciclo abortado con la sentencia break***")
amistad=input("digite nombre de una amistad: ")
amistad= amistad.upper()
for character in amistad:
    print(character)
    if charracter=="A":
        break
print("fin del ciclo")
print()
print("FIN")
```

6. REFERENTES BILBIOGRÁFICOS Arias, A. (2019). Aprende a programar con Python. Columbia. Buttu, M. (2016). El gran libro de Python. España: Marcombo. Caballero, R. (2019). Big data con Python: recolección, almacenamiento y proceso. Bogotá: Madrid: Alfaomega Colombiana. Cervantes, O. (2017). Python con aplicaciones a las matemáticas, ingeniería y finanzas. México, Alfaomega. Cuevas, A. (2017). Python 3: Curso práctico. Bogotá: Ediciones de la U. Guzdial, B. y Vidal, A. (2013). Introducción a la Computación y programación con Python. México: Pearson educación. Hinojosa, A. (2016). Python paso a paso. Bogotá: Ediciones de la U. Marzal, A. y Gracia, I. (2009). Introducción a la programación con Python. Universitat Jaume I. Ortega, J. (2018). Hacking ético con herramientas Python. Madrid: Ra-Ma. Pérez, A. (2016). Python fácil. México: Alfaomega Grupo Editor. Salazar, P. (2019). Empezando a programar en Python. Bogotá: Editorial Escuela Colombiana de Ingeniería