PYTHON

Básicamente, Python es un lenguaje de programación de alto nivel, interpretado y multipropósito. En los últimos años su utilización ha ido constantemente creciendo y en la actualidad es uno de los lenguajes de programación más empleados para el desarrollo de software. Python puede ser utilizado en diversas plataformas y sistemas operativos.

¿Tiene Python un ámbito específico? Algunos lenguajes de programación sí que lo tienen. Por ejemplo, PHP fue ideado para desarrollar aplicaciones web. Sin embargo, este no es el caso de Python. Con este lenguaje podemos desarrollar software para aplicaciones científicas, para comunicaciones de red, para aplicaciones de escritorio con interfaz gráfica de usuario (GUI), para crear juegos, para smartphones y por supuesto, para aplicaciones web.

Por las características que tiene Python lo convierten en un lenguaje muy productivo. Entre esas características está el hecho de ser un lenguaje potente, flexible y con una sintaxis clara y concisa. Además, no requiere dedicar tiempo a su compilación debido a que es interpretado.

Python es open-source, cualquiera puede contribuir a su desarrollo y divulgación. Además, no es necesario pagar ninguna licencia para distribuir software desarrollado con este lenguaje.

➢ Características del lenguaje

- ✓ *Python es un lenguaje de programación interpretado y multiplataforma cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.*
- ✓ *Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación estructurada y, en menor medida, programación funcional.*
- ✓ *Se trata de un lenguaje de propósito general.*

➢ Ventajas

- ✓ Libre y gratuito (OpenSource).
- ✓ Fácil de leer, parecido a pseudocódigo.
- ✓ Aprendizaje relativamente fácil y rápido: claro, intuitivo
- ✓ Alto nivel.
- ✓ Alta Productividad: simple y rápido.
- ✓ Tiende a producir un buen código: orden, limpieza, elegancia, flexibilidad
- ✓ Multiplataforma. Portable.
- ✓ Multiparadigma: programación estructural, orientada a objetos, funcional.
- ✓ Interactivo, modular, dinámico.
- ✓ Librerías extensivas.
- ✓ Gran cantidad de librerías de terceros.
- ✓ Gran comunidad, amplio soporte.
- ✓ Tipado dinámico (las variables pueden alojar distintos tipos de datos)
- ✓ Fuertemente tipado (para realizar operaciones entre variables, los valores de las mismas deben ser del mismo tipo de datos)

- **Desventajas**

  - ✓ Interpretado (velocidad de ejecución, etc.).
  - ✓ Consumo de memoria.
  - ✓ Errores durante la ejecución.
  - ✓ Dos versiones mayores no del todo compatibles (v2 vs v3).
  - ✓ Documentación a veces dispersa e incompleta.
  - ✓ Varios módulos para la misma funcionalidad.
  - ✓ Librerías de terceros no siempre del todo maduras.

Comenzando a programar

Cuando vamos a trabajar con Python debemos tener instalado, como mínimo, un *intérprete* del lenguaje (para otros lenguajes sería un *compilador*). El intérprete nos permitirá ejecutar nuestro código para obtener los resultados deseados. La idea del intérprete es lanzar instrucciones "sueltas" para probar determinados aspectos.

Pero normalmente queremos ir un poco más allá y poder escribir programas algo más largos, por lo que también necesitaremos un editor. Un *editor* es un programa que nos permite crear archivos de código (en nuestro caso con extensión *.py), que luego son ejecutados por el intérprete.

Todo archivo que contenga código Python debe tener como extensión .py para que el intérprete pueda ejecutarlo correctamente.

Código fuente y bytecode

Hasta ahora solo hemos hablado de los archivos de código Python, que utilizan la extensión .py. También sabemos que este lenguaje es *interpretado* y no *compilado*.

Sin embargo, en realidad, internamente el intérprete Python se encarga de generar unos archivos binarios que son los que serán ejecutados. Este proceso se realiza de forma transparente, a partir de los archivos fuente (los archivos que contienen nuestro código de programación). Al código generado automáticamente se le llama *bytecode* y utiliza la extensión *.pyc*.

Así pues, al invocar al intérprete de Python, este se encarga de leer el archivo fuente, generar el *bytecode* correspondiente y ejecutarlo. ¿Por qué se realiza este proceso? Básicamente, por cuestiones de eficiencia. Una vez que el archivo .pyc esté generado, Python no vuelve a leer el archivo fuente, sino que lo ejecuta directamente, con el ahorro de tiempo que esto supone. La generación del *bytecode* es automática y el programador no debe preocuparse por este proceso. El intérprete es lo suficientemente inteligente para volver a generar el *bytecode* cuando es necesario, habitualmente, cuando el archivo de código correspondiente cambia.

Por otro lado, también es posible generar archivos binarios listos para su ejecución, sin necesidad de contar con el intérprete. Recordemos que los archivos Python requieren del intérprete para ser ejecutados. Sin embargo, en ocasiones necesitamos ejecutar nuestro código en máquinas que no disponen de este intérprete. Este caso suele darse en sistemas

Windows, ya que, por defecto, tanto Mac OS X, como la mayoría de las distribuciones de GNU/Linux, incorporan dicho intérprete.

Primer Programa, el clásico "Hola Mundo!!"

El primer programa que vamos a escribir en Python es el clásico Hola Mundo, y en este lenguaje es tan simple como:

```
print ("Hola Mundo")
```

Si abrimos el editor que hayamos elegido, creamos un archivo que se llame "mundo.py", pegamos el código anterior y utilizando el menú de ejecución por defecto del IDE en cuestión (en el caso de VSCode, menú "Ejecutar"-"Iniciar Depuracion"). También podemos ejecutar el archivo abriendo la línea de comando por defecto, nos paramos en la ruta física donde tengamos el archivo y con el comando: *python mundo.py* tendremos el mismo resultado.

Imprimir datos por pantalla

Es muy probable que nuestro programa deba imprimir información por pantalla, para ello se utiliza la función ***print*** la cual es la encargada de mostrar el contenido que le pasemos entre paréntesis. Veamos:

```
print("Esto es un mensaje")
edad = 30
print ("Mi edad es:", edad)
nombre = "Pedro"
apellido = "Perez"
print ("Mi nombre es:",nombre," y mi apellido es:", apellido)
```

Captura de datos desde teclado

Los programas se hacen para tener interacción con el usuario. Una de las formas de interacción es solicitar la entrada de datos por teclado. Como muchos otros lenguajes de programación, Python también nos ofrece la posibilidad de leer la información introducida por teclado. Para ello se utiliza la función ***input():***

```
nombre = input("Ingrese su nombre: ")
apellido = input("Ingrese su apellido: ")
edad = int(input("Ingrese su edad: "))
print ("Mi edad es:", edad)
print ("Mi nombre es:", nombre," y mi apellido es:", apellido)
```

Note que con la edad se está utilizando además una función que se llama **int**, aquí lo que se hace es "convertir" lo que se ingresa por teclado en un numero entero. Si lo que se ingresa no es eso, se muestra un error. Más adelante profundizaremos en este tema.

Datos

Los programas están formados por código y datos. Pero a nivel interno de la memoria del ordenador no son más que una secuencia de bits. La interpretación de estos bits depende del lenguaje de programación, que almacena en la memoria no sólo el dato puro sino distintos metadatos. En resumen los datos es toda aquella información con la que contamos para poder resolver el problema que se nos plantea, por ejemplo, si el problema es hacer un programa de facturación seguramente tendremos datos como producto, precio, cantidad, cliente, condición frente al IVA, etc, etc.

Cada dato dentro de nuestro programa tiene, al menos, los siguientes campos:

- Un *tipo* del dato almacenado.
- Un *identificador* único para distinguirlo de otros objetos.
- Un *valor* consistente con su tipo.

Variables

Los datos los representamos en nuestro programa a través de las variables, las cuales juegan un papel súper importante en nuestro programa porque nos permite definir nombres para los valores que tenemos en memoria y que vamos a usar en nuestro programa.

```
nombre = valor
```

- **Reglas para nombrar variables**

En Python existen una serie de reglas para los nombres de variables:

- ✓ Sólo pueden contener los siguientes caracteres:
  - Letras minúsculas.
  - Letras mayúsculas.
  - Dígitos.
  - Guiones bajos (_).
- ✓ Deben empezar con una letra o un guión bajo, nunca con un dígito.
- ✓ No pueden ser una palabra reservada del lenguaje ("keywords").

Podemos obtener un listado de las palabras reservadas del lenguaje de la siguiente forma, usando la sentencia:

help('keywords')

```
>>> help('keywords')

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

- **Asignación**

En Python se usa el símbolo = para asignar un valor a una variable:

Algunos ejemplos de asignaciones a variables:

nota = 9;
sueldo = 75000.84
nombre = “Juan Perez”
materia = “programacion”
otra_materia = materia #aqui le estoy asignando el valor de la variable *materia* a
#otra variable que se llama *otra_materia*

Python nos ofrece la posibilidad de hacer una *asignación múltiple* de la siguiente manera:

uno = dos = tres = 3

Aquí hay 3 variables (uno, dos y tres) a las cuales se les asigna en una sola sentencia el valor 3.

a, b, c = 'string', 15, True

En una sola instrucción, estamos declarando tres variables: a, b y c y asignándoles un valor concreto a cada una:

>>> print (a)
string
>>> print (b)
15
>>> print (c)
True

Sentencias

Una sentencia es una instrucción que puede ejecutar el intérprete de Python. Hemos visto dos tipos de sentencias: *print* y la *asignación*. Cuando usted escribe una sentencia en la línea de comandos, Python la ejecuta y muestra el resultado, si lo hay. El resultado de una sentencia *print* es un valor.

Las sentencias de asignación no entregan ningún resultado. Normalmente un guion contiene una secuencia de sentencias. Si hay más de una sentencia, los resultados aparecen de uno en uno tal como se van ejecutando las sentencias.

Por ejemplo, el guion
print (1)
x = 2
print (x)
presenta la salida
1
2
De nuevo, la sentencia de asignación no produce ninguna salida.

Expresiones

Una expresión es una porción de código Python que produce o calcula un valor (resultado). Desde otro punto de vista, una expresión es una combinación de valores, variables y operadores.

- Un valor es una expresión (de hecho es la expresión más sencilla). Por ejemplo el resultado de la expresión 111 es precisamente el número 111.

- Una variable es una expresión, y el valor que produce es el que tiene asociado en el estado (si x → 5 en el estado, entonces el resultado de la expresión x es el número 5).

- Usamos operaciones para combinar expresiones y construir expresiones más complejas:

    - Si x es como antes, x + 1 es una expresión cuyo resultado es 6.

    - Si en el estado millas → 1, pies → 0 y pulgadas → 0, entonces 1609.344 * millas + 0.3048 * pies + 0.0254 * pulgadas es una expresión cuyo resultado es 1609.344.

    - La exponenciación se representa con el símbolo **. Por ejemplo, x**3 significa x^3.

    - Se pueden usar paréntesis para indicar un orden de evaluación: ((b * b) - (4 * a * c))

- Igual que en la matemática, si no hay paréntesis en la expresión primero se agrupan las exponenciaciones, luego los productos y cocientes, y luego las sumas y restas.

- Sin embargo, hay que tener cuidado con lo que sucede con los cocientes. Si **x / y** se calcula como la división entera entre **x** e **y**, entonces si **x** se refiere al valor 12 e **y** se refiere al valor 9, entonces **x / y** se refiere al valor 1.

- Si x e y son números enteros, entonces x % y se calcula como el resto de la división entera entre x e y. Si x se refiere al valor 12 e y se refiere al valor 9, entonces x % y se refiere al valor 3.

Más ejemplos:

```
A = 10
B = 20
C = A + B
print (C)
```

Como se puede apreciar, es un tanto confuso la identificación entre sentencias y expresiones, a mi entender una expresión es una combinación de operadores y operando donde se refleja una intención de calcular algo (por ej.: resultado = a+(b*c)/z), mientras que una sentencia es algo "atómico" algo más simple, donde no necesariamente deben existir operando ni operadores (por ej.: print (resultado)).

Generalidades para tener en cuenta

A continuación veremos algunas cuestiones para tener en cuenta a la hora de comenzar a generar código:

➢ Definición de bloques

A diferencia de otros lenguajes que utilizan llaves para definir los bloques de código, cuando se creó el lenguaje Python se quiso evitar estos caracteres por considerarlos innecesarios. Es por ello que en Python los bloques de código se definen a través de **espacios en blanco**, preferiblemente **4**. En términos técnicos se habla del tamaño de ***indentación***.

```
Linea 1 - if pwd == 'manzana':
Linea 2 -     print('Iniciando sesión ')
Linea 3 - else:
Linea 4 -    print('Contraseña incorrecta.')

Linea 5 - print('¡Todo terminado!')
```

En el pequeño programa que se muestra, se encuentran 3 bloques, uno formado por la linea 1 y 2, otro con la linea 3 y 4 y finalmente el bloque "general" formado por la linea 5. Aquí se ve que la linea 2 pertenece al bloque que inicio la linea 1, la linea 4 pertenece al

bloque que inicio la linea 3 y la linea 5 es independiente ya que no forma parte de ninguno de los bloques anteriores.

Consejo: *Esto puede resultar extraño e incómodo a personas que vienen de otros lenguajes de programación pero desaparece rápido y se siente natural a medida que se escribe código.*

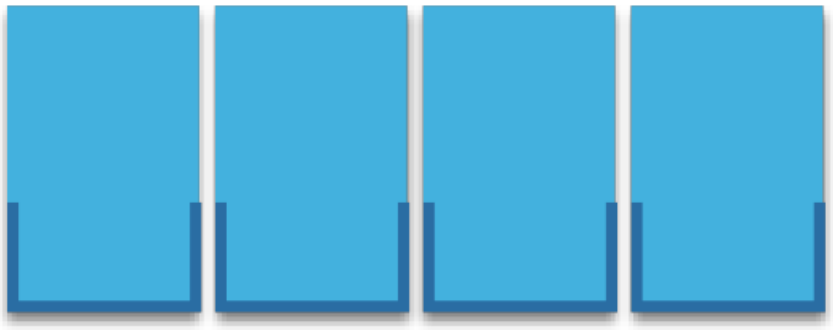

Figura 1: Python recomienda 4 espacios en blanco para indentar

➢ Comentarios

Los comentarios son anotaciones que podemos incluir en nuestro programa y que nos permiten aclarar ciertos aspectos del código. Estas indicaciones son ignoradas por el intérprete de Python.

Los comentarios se incluyen usando el símbolo almohadilla # y comprenden hasta el final de la línea.

```
#esta es una linea de comentario
print (10)
```

Si queremos comentar mas de una línea podemos hacerlo línea por línea o un conjunto se líneas. Por ejemplo:

```
#linea 1 comentada individualmente
#linea 2 comentada individualmente
#linea 3 comentada individualmente
"""linea 4 comentada en bloque
linea 5 comentada en bloque
linea 6 comentada en bloque
linea 7 comentada en bloque"""
print ("Esta línea no esta comentada")
```

Control de Flujo

Todo programa informático está formado por instrucciones que se ejecutan en forma secuencial de «arriba» a «abajo», de igual manera que leeríamos un libro. Este orden constituye el llamado *flujo* del programa. Es posible modificar este flujo secuencial para

que tome *bifurcaciones* o *repita* ciertas instrucciones. Las sentencias que nos permiten hacer estas modificaciones se engloban en el *control de flujo*.

➢ Estructura condicional "if"

La sentencia condicional en Python (al igual que en muchos otros lenguajes de programación) es ***if***. En su escritura debemos añadir una expresión de comparación terminando con dos puntos al final de la línea. Veamos un ejemplo:

```
temperatura = 40
if temperatura > 35:
   print("Aviso por alta temperatura")
```

Nota: *Nótese que en Python no es necesario incluir paréntesis (y) al escribir condiciones. Hay veces que es recomendable por claridad o por establecer prioridades.*

En el caso anterior se puede ver claramente que la condición se cumple y por tanto se ejecuta la instrucción que tenemos dentro del cuerpo de la condición. Pero podría no ser así. Para controlar ese caso existe la sentencia *else*. Veamos el mismo ejemplo anterior pero añadiendo esta variante:

```
temperatura= 20
if temperatura> 35:
   print("Aviso por alta temperatura")
else:
   print("Parámetros normales")
```

Podríamos tener incluso condiciones dentro de condiciones, lo que se viene a llamar técnicamente **condiciones anidadas**. Veamos un ejemplo ampliando el caso anterior:

```
temperatura= 28
if temperatura< 20:
   if temperatura< 10:
      print("Nivel azul")
   else:
      print("Nivel verde")
else:
   if temperatura< 30:
      print("Nivel naranja")
   else:
      print("Nivel rojo")
```

Python nos ofrece una mejora en la escritura de condiciones anidadas cuando aparecen consecutivamente un **else** y un **if**. Podemos sustituirlos por la sentencia **elif**:

Figura 2: Construcción de la sentencia `elif`

Apliquemos esta mejora al código del ejemplo anterior:

```
temperatura= 28
if temperatura< 20:
    if temperatura< 10:
        print("Nivel azul")
    else:
        print("Nivel verde")
elif temperatura< 30:
    print("Nivel naranja")
else:
    print("Nivel rojo")
```

Cuando escribimos condiciones debemos incluir alguna expresión de comparación. Estas expresiones se detallan en el apartado de tipos de datos booleanos. En ese mismo apartado vimos los operadores condicionales ***and, or y not***, los cuales también se utilizan en las condiciones de la estructura de control ***if***.

Vamos algunos ejemplos:

```
temperatura = 40
humedad = 80
if (temperatura > 30 and humedad > 60):
  print("Dia con calor y humedad")
if (temperatura < 42 or humedad < 85):
  print("Dia soportable, puede ser peor")
```

Cuando queremos preguntar por la veracidad de una determinada variable «***booleana***» en una condición, la primera aproximación que parece razonable es la siguiente:

```
is_cold = True
if is_cold == True:
    print("Usa campera")
else:
```

print(“No usa remera”)

Aquí la salida seria: *Usa campera*

➢ Sentencia “match-case”

Ésta se podría asemejar a la sentencia «switch» que ya existe en otros lenguajes de programación.

Comparando valores

En su versión más simple, el «pattern matching» permite comparar un valor de entrada con una serie de literales. Algo así como un conjunto de sentencias «if» encadenadas. Veamos esta aproximación mediante un ejemplo:

```
numero = 10
match numero:
   case 10:
      print("El numero es 10")
   case 11:
      print("El numero es 11")
   case 12:
      print("El numero es 12")
```

¿Qué ocurre si el valor que comparamos no existe entre las opciones disponibles? Pues en principio, nada, ya que este caso no está cubierto. Si lo queremos controlar, hay que añadir una nueva regla utilizando el subguión _ como patrón:

```
numero = 15
match numero:
   case 10:
      print("El numero es 10")
   case 11:
      print("El numero es 11")
   case 12:
      print("El numero es 12")
   case _ :
      print("El numero es OTRO")
```

Otra forma de utilizar la estructura de control es analizando un poco el siguiente ejemplo: *veremos un código que nos indica si, dada la edad de una persona, puede tomar alcohol:*

```
edad = 14
match edad:
   case 0 | None:   #aquí hay un or, lo que significa que puede ser una opción u otra
      print('No es una edad valida')
   case n if n < 17:
      print('No puede Beber')
   case n if n < 22:
      print('Puede beber con precaucion')
   case _:
      print('Si puede beber')
```

Ejercicio para Practicar

Realizar un programa donde se pida por teclado tres números; si el primero es negativo, debe realizar el producto de los tres y si no lo es, la suma. Mostrar los resultados.

Bucles

Cuando queremos hacer algo más de una vez, necesitamos recurrir a un bucle. En esta sección veremos las distintas sentencias en Python que nos permiten repetir un bloque de código.

➢ La sentencia while

El primer mecanismo que existe en Python para repetir instrucciones es usar la sentencia while. La semántica tras esta sentencia es: «Mientras se cumpla la condición haz algo».

Veamos un sencillo bucle que muestra por pantalla los números del 1 al 4:

```
value = 1
        while value <= 4:
                print(value)
                value += 1
```

El programa imprime: *1 2 3 4*

Interrumpir un bucle while

Python ofrece la posibilidad de romper o finalizar un bucle antes de que se cumpla la condición de parada. Supongamos un ejemplo en el que estamos buscando el primer número múltiplo de 3 yendo desde 20 hasta 1:

```
num = 20
while num >= 1:
   if num % 3 == 0:
      print(num)
      break
   num -= 1
```

El programa imprime: 18

Como hemos visto en este ejemplo, break nos permite finalizar el bucle una vez que hemos encontrado nuestro objetivo: el primer múltiplo de 3. Pero si no lo hubiéramos encontrado, el bucle habría seguido decrementando la variable num hasta valer 0, y la condición del bucle while hubiera resultado falsa.

Continuar un bucle

Hay situaciones en las que, en vez de romper un bucle, nos interesa saltar adelante hacia la siguiente repetición. Para ello Python nos ofrece la sentencia ***continue*** que hace precisamente eso, descartar el resto del código del bucle y saltar a la siguiente iteración.

```
num = 21
while num >= 1:
    num -= 1
    if num % 3 == 0:
        continue
    print(num, end=", ") # Evitar salto de línea
```

El programa imprime: 20, 19, 17, 16, 14, 13, 11, 10, 8, 7, 5, 4, 2, 1,

Como se puede apreciar, se imprimen todos los números comprendidos entre el 1 y el 21, exceptos los que son múltiplos de 3.

➢ La sentencia for

Python permite recorrer aquellos tipos de datos que sean iterables, es decir, que admitan iterar sobre ellos. Algunos ejemplos de tipos y estructuras de datos que permiten ser iteradas (recorridas) son: cadenas de texto, listas, diccionarios, ficheros, etc.
La sentencia ***for*** nos permite realizar esta acción.
A continuación se plantea un ejemplo en el que vamos a recorrer (iterar) una cadena de texto:

```
word = 'Python'
for letter in word:
    print(letter)
```

El programa imprime: P y t h o n

La clave aquí está en darse cuenta que el bucle va tomando, en cada iteración, cada uno de los elementos de la variable que especifiquemos. En este caso concreto ***letter*** va tomando cada una de las letras que existen en ***word***, porque una cadena de texto está formada por elementos que son caracteres.

Nota: La variable que utilizamos en el bucle for para ir tomando los valores puede tener cualquier nombre. Al fin y al cabo es una variable que definimos según nuestras necesidades. Tener en cuenta que se suele usar un nombre en singular.

Otro ejemplo, esta vez con tuplas:

```
mi_lista = ['Juan', 'Antonio', 'Pedro', 'Herminio']
for nombre in mi_lista:
   print (nombre)
```

Aquí el programa va a imprimir cada uno de los nombres de manera individual.

Otro ejemplo más:

```
for anio in range(2001, 2013):
   print (anio)
```

En el ejemplo se van a imprimir los años comprendidos entre el 2001 y el 2013. Note la particularidad de la función "***range***".

La función ***range*** tiene como objetivo generar numero comprendidos entre un rango, siguiendo la siguiente estructura ***range(start, stop, step):***

- **start**: Es opcional y tiene valor por defecto 0.
- **stop**: es obligatorio (siempre se llega a 1 menos que este valor).
- **step**: es opcional y tiene valor por defecto 1.

Ejemplos:

```
for i in range(1, 6, 2):
        print(i)
```

```
1
3
5
```

```
for i in range(2, -1, -1):
        print(i)
```

```
2
1
0
```

Romper un bucle for

Una sentencia ***break*** dentro de un for rompe el bucle, igual que veíamos para los bucles while. Veamos uno de los ejemplos anteriores, donde vamos a recorrer una cadena de texto y pararemos el bucle cuando encontremos una letra t minúscula:

```
word = 'Python'
for letter in word:
```

```
if letter == 't':
    break
print(letter)
```

Aquí se imprimirán solo las letras: P y

Usando el guión bajo

Hay situaciones en las que no necesitamos usar la variable que toma valores en el rango, sino que únicamente queremos repetir una acción un número determinado de veces.
Para estos casos se suele recomendar usar el **guión bajo _** como nombre de variable, que da a entender que no estamos usando esta variable de forma explícita:

```
for _ in range(10):
    print('Se imprime 10 veces!')
```

En este caso el programa va a imprimir **10** veces el mismo mensaje.

Bucles anidados

También podemos hacer uso de la anidación de bucles, lo que me permite hacer en un código reducido un procesamiento complejo y repetitivo, por ejemplo:

```
for i in range(1, 10):
    for j in range(1, 10):
        result = i * j
        print(f'{i} * {j} = {result}')
```

Lo que está ocurriendo en este código es que, para cada valor que toma la variable i, la otra variable j toma todos sus valores. Como resultado tenemos una combinación completa de los valores en el rango especificado.

Nota:

- Podemos añadir todos los niveles de anidamiento que queramos. Eso sí, hay que tener en cuenta que cada nuevo nivel de anidamiento supone un importante aumento de la ***complejidad ciclomática*** de nuestro código, lo que se traduce en mayores tiempos de ejecución.
- Los bucles anidados también se pueden aplicar en la sentencia while.

Ejercicio para Practicar

Escribir un programa que pida al usuario un número entero positivo y muestre por pantalla todos los números comprendidos entre el cero y el número ingresado separados por comas. Le agregamos un detalle más: mostrar la suma de todos los números.

Ejercicio para Practicar

Escribir un programa que pregunte al usuario una cantidad de dinero ($) a invertir, el interés anual y el número de años, y muestre por pantalla el capital obtenido en la inversión cada año que dura la inversión. Nota: el valor inicial de cada año depende del capital + interés obtenido en el año anterior.