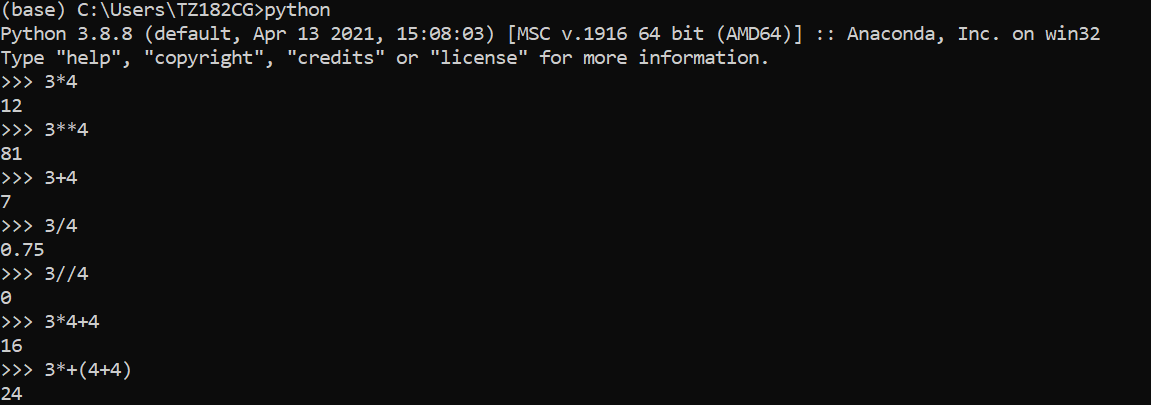
**Operaciones aritméticas en Python**

**Ejemplo 1:**

Escriba las siguientes operaciones en el prompt de Python y verifique los resultados:

1. 3\*4
2. 3\*\*4
3. 3+4
4. ¾
5. 3//4
6. 3\*4+4
7. 3\*(4+4)

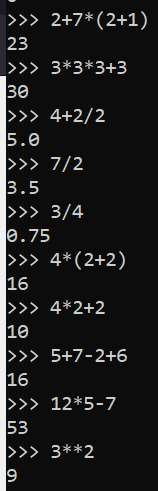
Resultado:

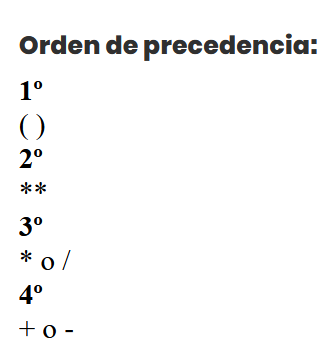


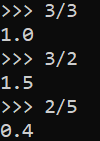
¿Por qué las diferencias en las dos últimas?

Por el orden de precedencia. Primero se resuelve el paréntesis y luego la potenciación, luego la división o multiplicación, y por último la suma y resta. Es importante destacar que multiplicación y división están en el mismo nivel, así que como se tome no modificara el resultado.

**Ejemplo 2:**

Resuelva las siguientes operaciones, primero a mano y luego en la consola de Python:

1. 2+7\*(2+1)= 23
2. 3\*3\*3+3= 30
3. 4+2/2= 5
4. 7/2 = 3.5
5. ¾=0.75
6. 4\*(2+2)=16
7. 4\*2+2=10
8. 5+7-2+6=16
9. 12\*5-7=53
10. 3\*\*2=9

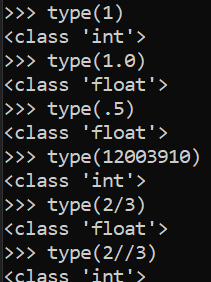


Numero entero int()

Numero decimal float()

**Tipo de datos**

Con la función type() podremos saber con qué tipo de dato estamos trabajando:



**Ejemplo:**

>>>type(1)

>>>type(1.5)

>>>type(.5)

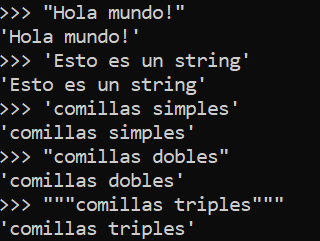
>>>type(12003910)

>>>type(2/3)

>>>type(2//3)

**Tipo de datos - Strings**

Los **strings(str)** son cadenas de letras, símbolos y números en formato de texto.

**String(str)** es el tipo de datos que usamos para trabajar con textos en Python. En Python, los textos (o strings) son delimitados por comillas, simples (‘’) o dobles(“”). Sin embargo, es necesario ser consistentes con nuestro código con las comillas para que el intérprete Python que utilicemos sepa dónde empieza un texto y donde termina.

**Ejemplos:**

>>> ‘Hola mundo’

>>>’Esto es un string’

>>>’Comillas simples’

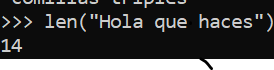
>>>”Comillas dobles”

>>>”””Comillas triples”””

Para descubrir el tamaño de un string solo basta con utilizar la función **len()**

Esta función nos devolverá el número de caracteres que contiene un string.

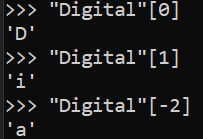
Ejemplo:



Esta función es muy útil para poder trabajar con textos.

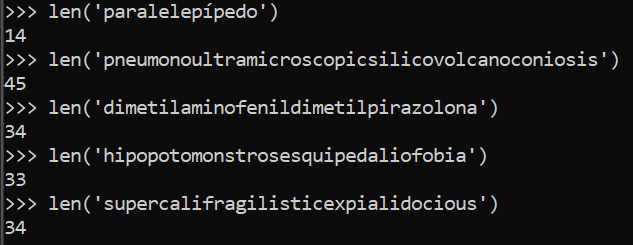
A parte de contar cuantas letras hay en un string, también se puede acceder a los caracteres específicos del string.

Para acceder a los caracteres, hay que utilizar un índice, que es colocado dentro de corchetes([]) al lado del string, indicando la ubicación de la letras que queremos acceder.



**Importante:** En Python, los índices comienzan desde el 0, es decir, que la primer letra de un string se encontraría en el índice 0, y no en el índice de número 1.

Ejercicio:

Descubra la longitud de las siguientes palabras:

1. 'paralelepípedo'
2. 'pneumonoultramicroscopicsilicovolcanoconiosis'
3. 'dimetilaminofenildimetilpirazolona'
4. 'hipopotomonstrosesquipedaliofobia'
5. 'supercalifragilisticexpialidocious'

**Interpolación - Strings**

Para complementar nuestro conocimiento de strings, ahora vamos a hablar sobre la interpolación de strings. En otras palabras, esto solamente quiere decir “todo lo que nos permita cambiar, modificar o alterar” un string.

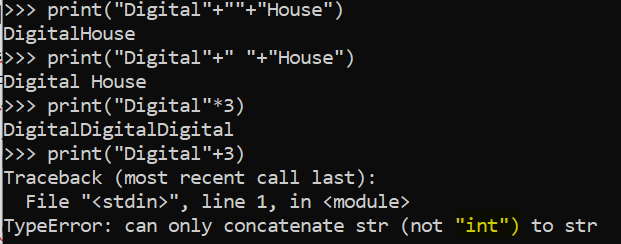
La interpolación de cadenas de texto se refiere al proceso de evaluar el valor de una string en uno o más marcadores de posición. En poca palabras ayuda a los desarrolladores con el formato y concatenación de strings.

**Concatenación de strings**

En Python existe más de una forma de concatenar so unir strings. La string resultante de la concatenación también es tratada como un objeto de tipo string por Python. Cuando buscamos combinar dos strings en una única cadena de texto debemos utilizar el operador “+”.

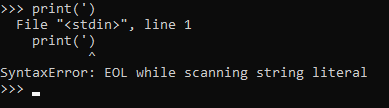
No obstante no solo podemos unir dos strings, sino que podemos hacer que se multipliquen usando el comando “\*”.

Ejemplos:



**ERRORES Y COMO LEERLOS**

Parte de la vida de un programador se le pasa leyendo código. También leyendo errores. Es de los errores que aprendemos más y descubrimos que esta errado en nuestro código. Los errores en Python se ven de las siguiente forma:



La primera línea nos indica **en qué archivo y qué línea es la que provoca el error**.

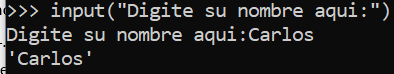
La segunda línea intenta mostramos con una flecha **donde ocurre el error**. Hay veces donde esta línea no puede indicar exactamente donde ocurre el error, pero suele estar muy cerca.

Al final vamos a encontrar un **tipo de error** con un mensaje indicado qué tipo de error cometimos, esto es fundamental para ayudarnos a encontrar la solución.

Tomando el mensaje después del tipo de error que es **“EOL while scanning string literal”** y buscando en Google, encontrara una gran variedad de resultados. La herramienta más popular para solucionar estos errores y buscar la solución a este tipos de mensajes es *Stack overflow*.

En el ejemplo anterior, el error es del tipo SyntaxError (error de sintaxis) y su descripción es: "EOL while scanning string literal". En la línea inmediatamente superior, Python señala dónde percibió el error, lo que ocurre muy cerca de donde realmente se encuentra el error. Es decir, aquí es donde debe buscar primero. Finalmente, en la línea inmediatamente superior indica qué archivo y la línea donde ocurrió. En el ejemplo anterior, el "archivo" es "stdin", que en realidad se refiere al Prompt interactivo de Python.

**VARIABLES**



Las variables sirven para guardar información y acceder a esta más tarde, esta información se debe almacenar en algún lugar. Las variables son nombres que usamos para almacenar información específica como: edad, dirección, apodo, apellido, etc.

Sin variables sería prácticamente imposible crear programas. En la programación se depende mucho de estas.

**DEFINIENDO VARIABLES**

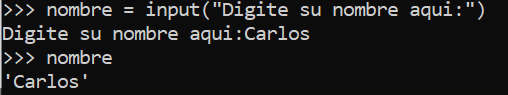
Python hace que definir una variables (reservar una letra / palabra específica y asignarle un valor) sea relativamente sencillo. Simplemente asta con escribir el nombre de la variables, seguido de un símbolo igual (=). Lo que venga después del igual es el valor que intentaremos asignarle a nuestra variable. La sintaxis para definir una variable entonces será la siguiente:

**Variable = valor**

Cabe mencionar que, como muchas cosas en Python, las variables distinguen entre mayúsculas y minúsculas. Es decir, si alguna letra está en mayúsculas o minúsculas, eso hace una diferencia para el lenguaje Python. Por lo tanto la variable “a” es considerada diferente de la variable ”A”.

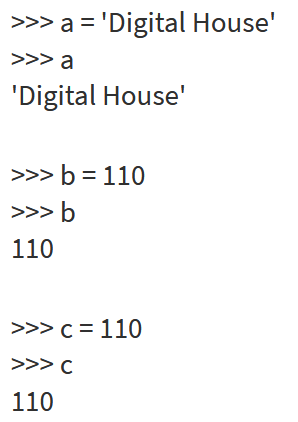
**LLAMANADO UNA VARIABLE**

Para llamar a una variable y acceder a su valor asignado, simplemente vas con escribir el nombre con el que definimos esa variable.



No es una regla, pero cuanto más descriptivo sea el nombre de su variable, más fácil podrán identificar su código más adelante.

A la variables, también se les puede asignar un valor fijo.

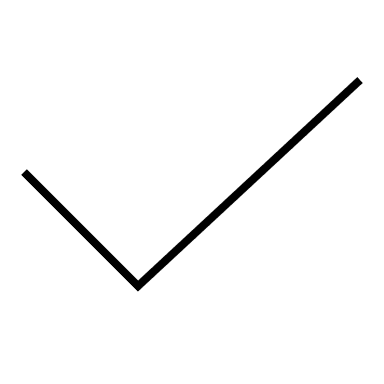


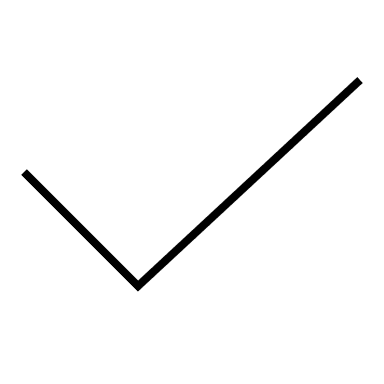
**BEST PRACTICES PARA TRABAJAR CON VARIABLES**

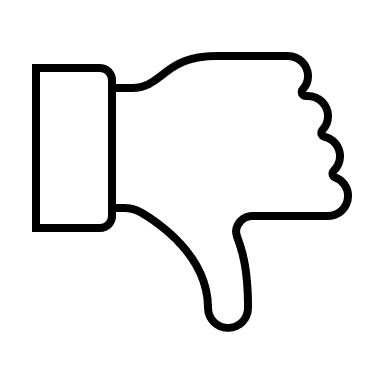
Por último, siempre es importante revisar las buenas prácticas al trabajar con distintos lenguajes, ya que esto permite que nuestro código sea más fácil de leer por otras personas.

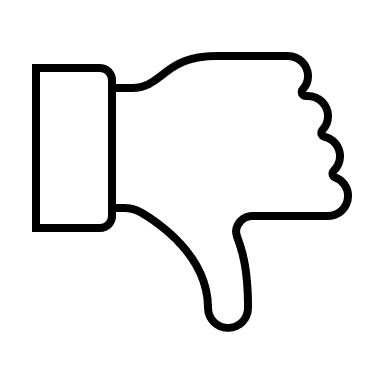
La Guía de estilo oficial de Python (PEP 8 o "Python Enhancement Proposal #8") establece que los nombres de las variables deben estar siempre en minúsculas, y de ser necesario, pueden estar separados por guiones bajos (\_) para mejorar su legibilidad. Esta guía establece que los nombres de las funciones también deberían seguir esta lógica.

**Veamos algunos ejemplos:**

Variable

otra\_variable

NombrE\_PosiblE

NoMbRe\_PoSiBlE\_2

Existen varias formas de nombrar una variable, mezclando mayúsculas y minúsculas, o también incluyendo números. Pero según las buenas prácticas, es recomendable nombrarlas como los primeros 2 ejemplos, para facilitar su legibilidad.

**TIPOS DE DATOS**

En Python tenemos algunos tipos de datos básicos (“Basic Data Types” o BDT) que son: números enteros (int), numero de punto flotante (float), cadenas de texto (str) y booleanos (bool).

Lo primero que tenemos que entender que los tipos de datos son objetos. Objetos como un auto, una persona, un sentimiento o, un horario apuntado con el dentista. ¿Las personas son objetos? ¡Si! Sin embargo, no en el sentido común y amplio que generalmente pensamos. Nos referimos a que los objetos tienen propiedades y características.

Por ejemplo, un VW Beetle (comúnmente llamado “escarabajo”) es un auto todo curvo. No hay partes cuadradas en su diseño. Esto habla de algunas de sus características. El Beetle puede andar, tocar la bocina, estacionarse, llevar a la gente de un punto a otro, y mucho más. Esto habla de las acciones que puede realizar. Una persona puede hablar, un perro ladrar, y así sucesivamente. Todos los objetos tienen sus características y sus acciones son lo que llamamos propiedades.

Otro nombre que los programadores les gusta usar para referirnos a objetos es una **clase**. Un objeto es una clase. Pero, ¿Cómo nos enteramos de eso? Simplemente podemos usar la función type() para averiguar a qué tipo de dato pertenece cualquier cosa que queramos analizar.

***type() devuelve el tipo de datos de aquello que le proporcionamos.***

Ejemplo:

>>>type(2)

<class ‘int’>

Existen un montón de características y propiedades de cada tipo de dato dentro de Python. A continuación veremos las más básicos ya que la gran mayoría son de utilidad muy específica.

**Strings:**

Son cadenas de texto. Sus características pueden ser:

1. Contener una letra
2. Contener múltiples palabras
3. Podemos concatenar string (acción)

Sin embargo, no podemos realizar operaciones aritméticas entre ellas. En otras palabras, ‘2’+’2’ es ‘22’ y no 4. Solamente podemos colocar cadenas de texto una al lado de la otra.

**Booleanos:**

Los booleanos son tipos de datos bastante simples. Solamente existen 2: Verdadero (TRUE) y Falso (FALSE). Deben estar escritos con la primer letra en mayúsculas, Estos valores booleanos se utilizan para validar situaciones y su estado, como si fuesen ‘si’ y ‘no’. True seria ‘si’ y False seria ‘no’. Vea los ejemplos a continuación:

* ¿el cielos es azul? **Si**
* ¿el sol es verde? **No**
* ¿Los peces respiran? **True**
* ¿Los peces caminan? **False**

Los booleanos solo se puede utilizar para validar situaciones y su estado. Esto significa que podemos eventualmente hacer varias comparaciones lógicas y evaluar su estado final como un True o un False ¿Qué queremos decir con esto? Empecemos primero por ver cuáles son los operadores lógicos que nos permitirán comparar distintas situaciones.

Existen 3 **operadores lógicos** **and**, **or** y **not**. Por ahora, solo trabajaremos en los primeros dos:

“**AND**” es un operador lógico que va a devolver como resultado True si y solo si sus dos operandos (lo que le pedimos evaluar) son True. Sería la traducción de decir “que A **Y** B sean verdaderos”.

“**OR”** es un operador lógico que va a devolver como resultado True si por lo menos uno de sus operandos es True. Sería la traducción de decir **“**que A **O** B sea verdadero**”**

**Ejemplos:**

>>> True and True

True

**Nota:** Siempre escribir “True” con la primera letra en mayúscula.

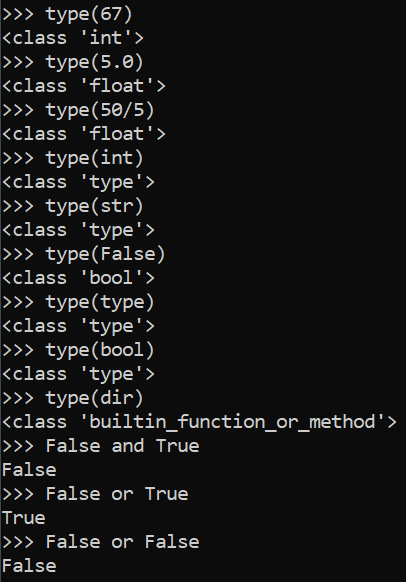
**Ejemplo:**

>>>3>2 and 1>0

True

Cuando utilizamos el operador **and** le estamos preguntando a Python si ambos resultados a esas preguntas son verdaderos. En cambio, si estuviéramos usando el operador **or** solo necesitaríamos que uno sea verdadero para que Python nos devuelva como resultado True.

**Ejemplos:**



**FUNCIONES**

Las funciones integrados (o “Built-In”), es decir, que ya se encuentran integradas dentro de nuestra versión de Python y tenemos acceso a ellas en cualquier archivo o programas que creemos. Por ahora, vamos a limitarnos a usar funciones integrado o a importar funcionas ya creados en diferentes módulos, pero también podemos crear nuestras propias funciones.

Por ahora, existen 69 funciones integradas que están detalladas en la documentación oficial ***Funciones Built-In documentación de Python – 3.9.5***

**Nota:**

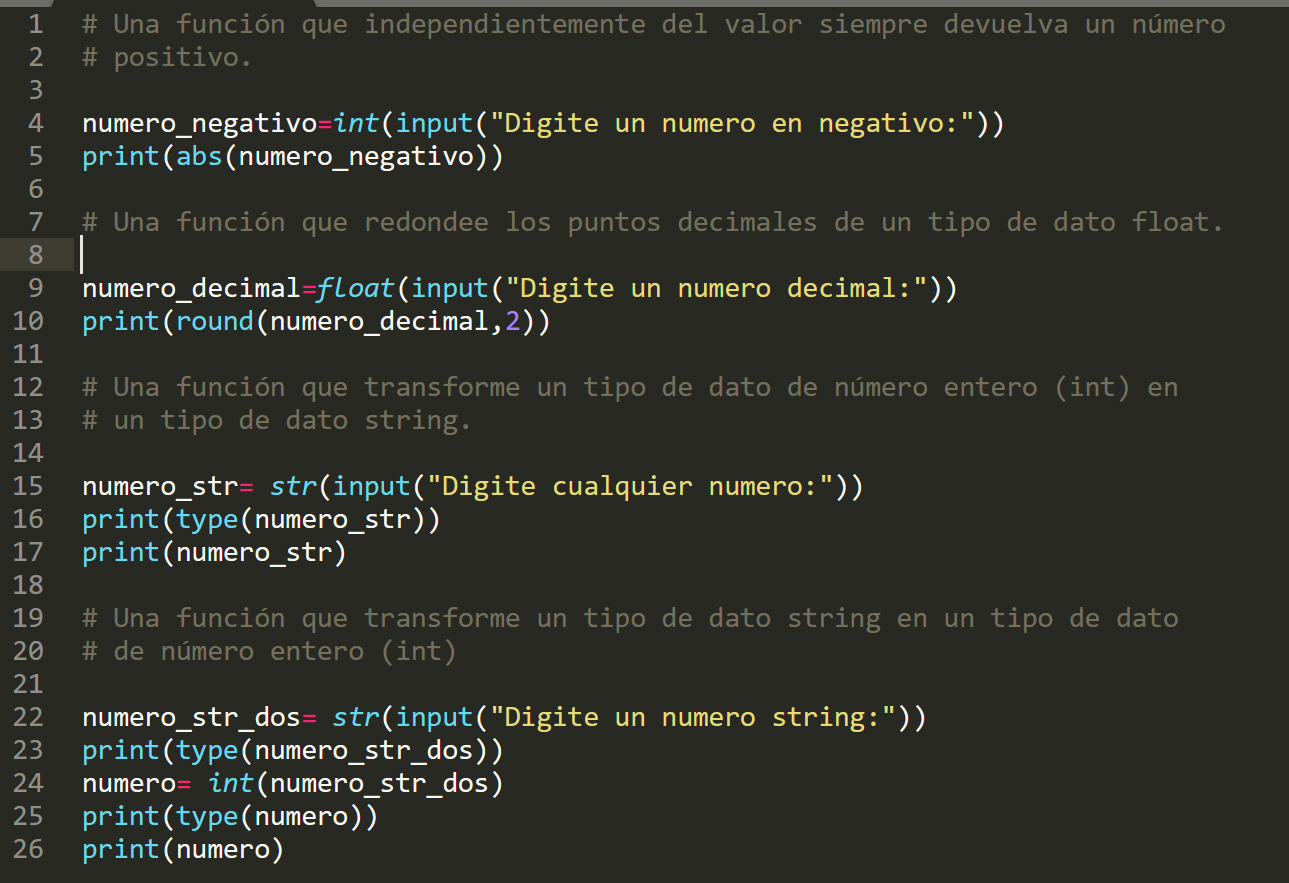
Hay que recordar que Python y su librerías están en constante actualización. Si se revisan versiones anteriores seguramente encontremos que algunas de las funciones no existían para esas versiones, o que incluso tenían funcionalidades diferentes.

Sin embargo, existen otras funciones que no se encuentran en esa lista. Estas son las funciones que debemos importar primero para poder usarlas como la función randint(). Estas funciones existen dentro de módulos integrados que básicamente son “librerías” de código que no está automáticamente disponibles para el usuario por cuestiones de performance. Python ya nos brinda casi 70 funciones, pero podemos acceder a través de estas librerías a cientos, sino, a miles de otras funciones. Sería poco eficiente tener la disponibilidad de todo este código en cada proyecto, por eso siempre trabajaremos trayendo a nuestro documento lo estrictamente necesario.

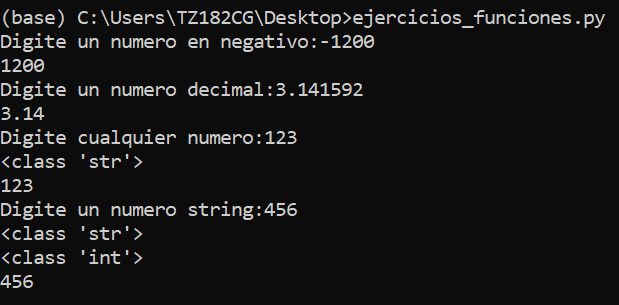
¿Como se identifican las funciones? Por defecto, tienen su nombre en minúscula por ej: (“print”) y deben tener paréntesis al final (“()”). Cuando lo combinamos tenemos nuestra función print().

Como ya mencionamos anteriormente, existen infinidad de funciones, por lo tanto, entra en nuestra responsabilidad leer sobre los diferentes objetos, clases y funciones, principalmente en la documentación de Python. A medida que vayamos encontrando funciones que no conozcamos.

**Ejercicios:**



**Consola:**



**CLASE 3 – CURSO DE ADMISIONN DATA SCIENCE**

1. Declaración de variables
2. Slicing
3. Booleanos
4. Operadores lógicos
5. Funciones e importar funciones
6. Función input y format

**INDEXACION Y SLICING**

En Python, los caracteres de una cadena de texto se obtienen mediante indexación, proporcionado el desplazamiento número del componente deseado entre corchetes después de la cadena. De tal manera se obtiene de la cadena un carácter en a posición especificada.

Veamos la siguiente interacción:

>>> S = ’spam’

>>> S[0], S[−2] #Indexing

--- (‘s’,’a’)

>>> S[1:3], S[1:], S[:-1] #Slicing: extraer una porción de la cadena de texto

(‘pa’, ‘pam, ‘spa’)

La indexación (S[i]) captura los componentes en los desplazamientos:

* + - El primer elemento está en el desplazamiento 0.
    - Los índices negativos significan contar hacia atrás desde el final.
    - S[0] obtiene el primer elemento
    - S[-2] obtiene el Segundo elemento del final (como S[len(S)-2]

**Sintaxis**

S[i:j] --- extra secciones contiguas de secuencias:

* + - El límite superior no es inclusivo
    - El valor predeterminado de los límites de los sectores e 0 y la longitud de la secuencia, si se omite.
    - S[1:3] obtiene elementos en los desplazamientos 1 hasta 3 , pero sin incluir 3.
    - S[1:] captura los elementos en el desplazamiento 1 hasta el final (la longitud de la secuencia).
    - S[:3] captura los elementos en el desplazamiento 0 hasta 3, pero sin incluirlo.
    - S[:-1] captura los elementos en el desplazamiento 0 hasta el último elemento, pero sin incluirlo.
    - S[:] obtiene elementos en los desplazamientos de 0 al final, lo que hace una copia de nivel superior de S.

**Invocar un método**

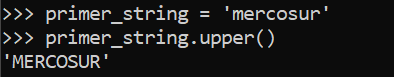
* Los métodos son simplemente funciones que están asociadas con objetos particulares y actúan sobre estos.
* Los métodos se invocan con la notación punto (.)
* Un método es una acción que Python pude realizar en un dato.
* Cada método va seguido de un paréntesis, porque los métodos a menudo necesitan información adicional para hacer su trabajo
* Esa información se proporciona entre paréntesis

***objeto.metodo (argumentos)***

La expresión de llamada al método significa esto: **llame al método para procesa el objeto con argumentos**

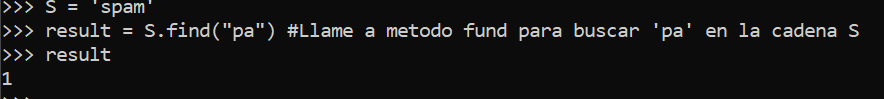
* El punto (.) Después del nombre en name.upper() le dice a Python que haga el método upper() actúe sobre el nombre de la variable.
* La función upper() no necesita información adicional, por lo que sus paréntesis están vacíos.

**Ejemplo:**



El método **find** devuelve un entero representando la posición donde inicia la subcadena dentro de cadena. Si no la encuentra, retorna -1.

**Ejemplo:**



**Importar un módulo:**

* Python también tiene un módulo incorporado llamado math, que amplía la lista de funciones matemáticas.
* Para usarlo debe importar el módulo matemático:

Sintaxis:

**import** math

El método math.floor() redondea un numero HACIA ABAJO al entero más cercano y devuelve el resultado.

Sugerencia para redondear un numero hacia arriba al entero más cercano, observe el método math.cell()

* También puede importar una función específica desde un módulo. Aquí está la sintaxis general de este enfoque:

**from** nombre\_modulo **import** nombre\_funcion

* Puede importar tantas funciones como deseemos de un módulo separando el nombre de cada función con una coma:

**from** modulo\_name **import** function\_0, function\_1. Function\_2

* Con esta sintaxis, no necesita usar la notación de puntos cuando llama una función.
* Veamos con un ejemplo:

**from** nombre\_modulo **import** nombre\_funcion

**from** math **import** floor

* Vemos con un ejemplo:

**from** math **import** floor, ceil, sqrt

* Con esta sintaxis, no necesita usar la notación de puntos cuando llama a una función.

**Función Input:**

Escribir mensajes claros

* Cada vez que utilice la función input(), debe incluir un mensaje claro y fácil de seguir que le diga al usuario exactamente qué tipo de información está buscando. Cualquier instrucción que indique al usuario que debe escribir debería funcionar. Por ejemplo:

**nombre =** input(‘Introduzca su nombre’)

* Agregue un espacio al final de los mensajes (después de los puntos en el ejemplo anterior) para separar el mensaje de la respuesta del usuario y para dejar claro al usuario donde escribir su texto.

Uso de int() para aceptar entradas numéricas.

* + - Cuando se utiliza la función input(), Python interpreta todo lo que el usuario introduce como un string.

Considere el siguiente ejemplo que pregunta por la edad del usuario:

>>> edad = input (‘¿Cuántos años tiene?’)

¿Cuántos años tienes? 21.

Edad >>> ‘21’ --- string

El usuario introduce el numero 21 pero cuando le pedimos a Python el valor de edad, devuelve ‘21’, la representación de cadena del valor numérico introducido.

Sabemos que Python interpreto la entrada como una cadena porque el número ahora está entre comillas. Si todo lo que desea hacer es imprimir la entrada, esto funciona bien. Pero si intentas usar la entrada como un número, obtendrás un error:

>>> edad = input (‘Cuantos años tiene?’)

¿Cuántos años tienes? 21

1. >>>edad >=18

Cuando se intenta utilizar la entrada para hacer una comparación numérica en (1) Python produce un error porque no puede comparar una cadena con un entero: la cadena ‘21’ que se asigna no se puede comparar con el valor numero 18

¿Cuántos años tienes? 21

1. >>>edad = int(edad)
2. Podemos resolver este problema usando la función int(), que le dice a Python que trate la entrada como un valor numérico. La función int() convierte una representación de cadena de un numero en una representación numérica, como se muestra a continuación:

>>>edad = input(‘¿Cuántos años tiene?’)

¿Cuántos años tienes? 21

1. Edad = int(edad)

En algunas situaciones, querrá usar el valor de una variable dentro de un string. Por ejemplo, es posible que nos gustaría dos variables representen un nombre y un apellido respectivamente y, a continuación desee combinar esos valores para mostrar el nombre completo de alguien:

first\_name = ‘maria’

second\_name = ‘ramirez’

1. full\_name = f’{first\_name}{last\_name}

print (full\_name)

Para insertar el valor de una variable en un string.

* Colocamos la letra f inmediatamente antes de las comillas de apertura (a). Coloque llaves alrededor del nombre o nombre de cualquier variable que deseemos utilizar dentro de la cadena. Python reemplazara cada variable con su valor cuando se muestre el string.
* Estos strings se denominas f-strings. La f es por format, porque Python da formato a la cadena reemplazando el nombre de cualquier variable entre llaves por su valor.

También podemos utilizar f-strings para redactar mensajes completos utilizando la información asociada una variable como se muestra a continuación:

first\_name = "maria"

last\_name = "ramirez"

full\_name = f"{first\_name} {last\_name}"

➊ print(f"Hola, {full\_name.title()}!")

El nombre complete se utiliza en una oración que Saluda al usuario ➊, y el método title() cambia el nombre a la primer letra de cada variable en mayúscula. Este código devuelve un saludo simple pero con un formato agradable.

Hola, Maria Ramirez!

**Operadores racionales**

Python tiene varios tipos de operadores. En este caso veremos los operadores relacionales (o de comparación), son similares a los que ya conocemos de matemáticas:

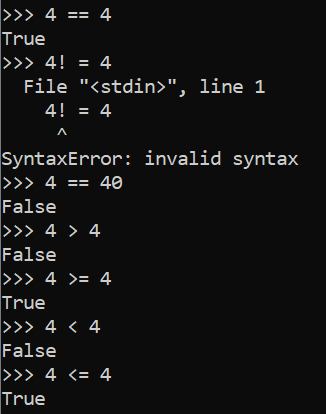
**<,>,<=,>=, == ,!=.**

La siguiente tabla resume el funcionamiento de los operadores relacionales en Python. En el asumimos que estamos haciendo ‘**a (operador) b**’, es decir, que estamos aplicando un operador a dos operandos, **a** y **b**.

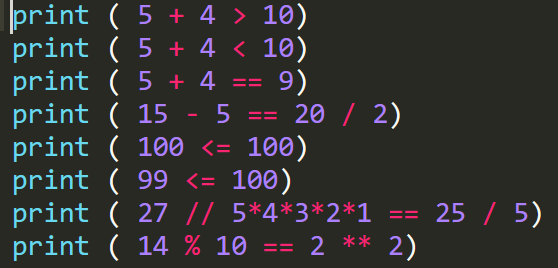
|  |  |  |
| --- | --- | --- |
| **Operador** | **Descripción** | **Ejemplo** |
| **<** | **True** si **a** es menor que **b**. | 5 < 2 resulta en **False**. |
| **>** | **True** si **a** es mayor que **b**. | 5 > 2 resulta en **True**. |
| **>=** | **True** se **a** es mayor o igual a **b**. | 5 >= 2 resulta en **True**. |
| **<=** | **True** si **a** es menor o igual a **b**. | 5 <= 2 resulta en **False**. |
| **==** | **True** si **a** y **b** son iguales. | 5 == 2 resulta en **False**. |
| **!=** | **True** si **a** u **b** son diferentes. | 5 != 2 resulta en **True**. |

***Nota:*** *recuerde que el operador de igual (=) era el que usábamos para definir variables y otros objetos. Es por esto que el operador relacional, para realizar comparaciones de igual siempre va a ser con el doble símbolo de igual (==)*

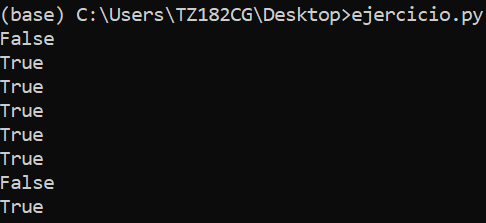
Ejemplos:



Ejercicios:



Consola:



**CONDICIONALES (IF, ELSE)**

Imaginemos que hemos creado un sitio web de acceso de usuarios. Y en este programa solo las personas mayores de 16 años pueden crear cuentas. Debemos encontrar la forma de aplicar una **condición** en nuestro código para asegurarnos que solo aquellas personas mayores de 16 años puedan crearse una cuenta.

Aquí es donde entran **condicionales**. Veamos el siguiente ejemplo para diagramar nuestro problema

**Si** edad es mayor que 16 años:

Imprimir “¡Cuenta creada con éxito!”

**Caso contrario.**

Imprimir “Lo siento, aun no puede registrarse.”

Veamos cómo podríamos escribir un script que resuelta esta situación. Primero tenemos que familiarizarnos con los condicionales **if, elif y else**. Los condicionales sirven para ejecutar determinadas líneas de código **cuando cumpla la condición** que definimos.

**Todos los condicionales** deben finalizar con un símbolo de dos puntos (:), para indicar que estamos ingresando a un bloque de código especifico.

**IF:**

El operador **if**, nos sirve para determinar una condición específica, y que solamente queremos ejecutar el código que le sigue cando se cumpla esa condición. Es el primer paso.

**if se utiliza al principio de la línea, seguido por nuestra condición, y finalizado por el símbolo :.**

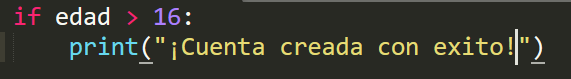
En nuestro ejemplo, debemos indicar que queremos imprimir un mensaje especifico cuando se cumpla la condición que Edad > 16.

Entonces:

**Si** edad mayor que 16 años:

Imprimir “¡Cuenta creada con éxito!”

Se convierte en:



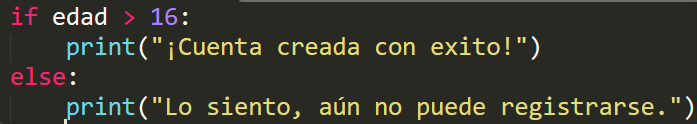
**ELSE:**

**else** es el operador que vamos a usar para capturar todos los casos restantes. Es decir, el famoso “caso contrario”. **Else NO necesita que agreguemos ninguna condición.** Las líneas de código dentro del bloque del **else**, se van a aplicar en **aquellos casos en los cuales no se cumpla ninguna condición planteada anteriormente.** Funciona como una red de seguridad.

Imagínense en nuestro ejemplo, donde por ahora solamente definimos el código se va a ejecutar cuando se cumpla la condición que la edad del usuario es mayor a 16 años. ¿Qué código se ejecuta si el usuario nos es mayor a 16 años? ¿Debemos crear una condición para cada edad? Eso llevaría mucho tiempo…

Aquí es donde **else** viene al rescate. **Si ya no tenemos más condiciones que agregar**, es una buena práctica agregar una condición **else**, para ejecutar algún código en caso de que no se cumpla ninguna condición. De no agregarla, nuestro código devolverá **None** (es decir, nada)

Volvamos a nuestro ejemplo:

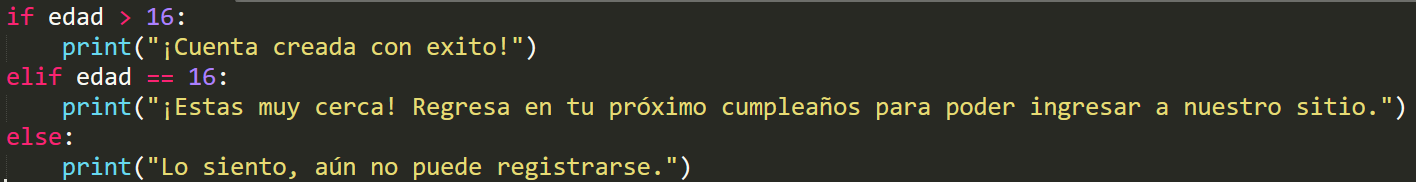


**ELIF:**

**elif** es la forma que tiene Python de decir **“si las condiciones anteriores no son verdaderas, entonces probemos la siguiente condición:”**. En otras palabras, es la forma de añadir más condiciones, en conjunción con las planteadas anteriormente (en un **if** o en otros **elif**).

Supongamos que queremos agregar una condición adicional, para enviar el un mensaje especial motivando a aquellos usuarios que tienen exactamente 16 años, para que no se desmotiven y vuelvan al sitio luego de su próximo cumpleaños.

Entonces nuestro código quedara de la siguiente forma:



**¡CUIDADO CON LA INDENTACION!**

La indentación es **crítica** para el funcionamiento correcto de nuestro código en Python.

Tomemos el ejemplo de nuestra primera condición, el bloque **if**. La indentación le indica a Python que el código se encuentra dentro del **bloque de condición** if. Por lo general, esta indentación contiene cuatro espacios ("\s\s\s\s") o una tabulación ("\t"). Traducido sería algo como: "Si la variable de edad es mayor que 16, ejecute la impresión que está asociada con el if, de no cumplirse, ejecute la impresión asociada con el elif, de lo contrario, ejecute el código que está contenido dentro del else".

**Ejemplo:**

1. **Cree un archivo "mi\_sitio.py" con el siguiente código:**

**print("Bienvenido a Redes Sociales Inc.")**

**print("Para crear una cuenta, basta con ingresar su edad.")**

**edad = int(input("Por favor, ingrese su edad: "))**

**if edad > 16 :**

**print("¡Cuenta creada con éxito!")**

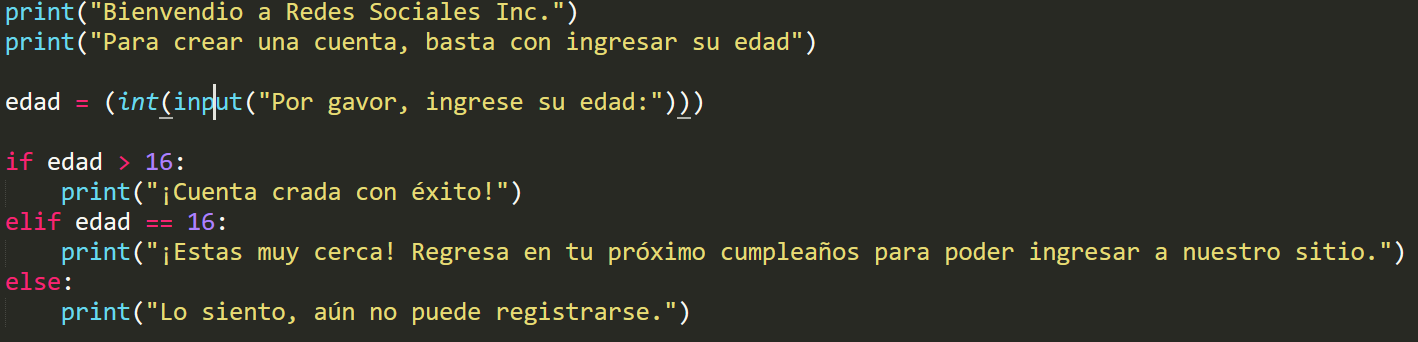
**elif edad == 16 :**

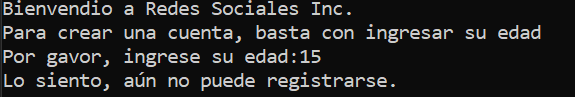
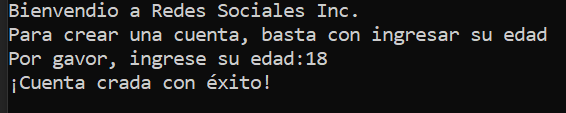
**print("¡Estas muy cerca! Regresa en tu próximo cumpleaños para poder ingresar a nuestro sitio.")**

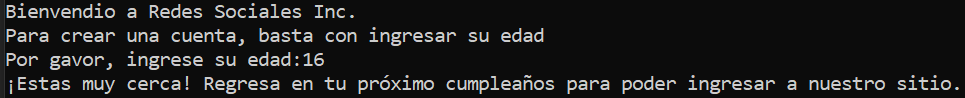
**else:**

**print("Lo siento, aún no puede registrarse.")**

**RESOLUCION:**





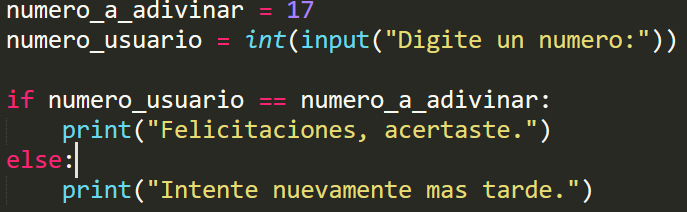
1. **Hagamos algo diferente, ahora creen un juego (un programa de Python) con los conceptos aprendidos, donde el usuario tenga que adivinar un número. El programa debe:**

• Definir un número para adivinar;

• Recibir un valor de entrada el usuario;

• Comparar ese valor con el número inicial e imprimir "¡Felicitaciones, acertaste!", en caso de que el valor sea igual al número inicial;

• Caso contrario, imprimir "Intente nuevamente más tarde.".



1. Ahora creen otro programa. El programa debe:

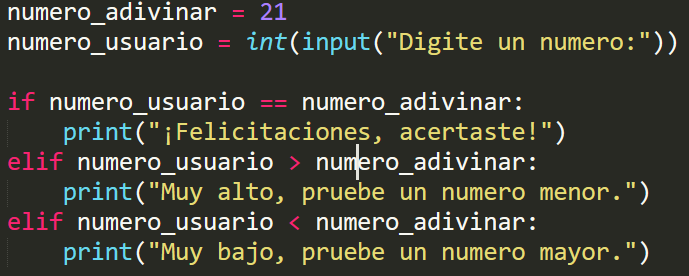
• Definir un número para adivinar;

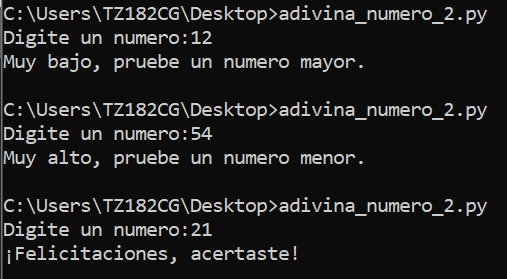
• Recibir un valor de entrada el usuario;

• Comparar ese valor con el número inicial e imprimir "¡Felicitaciones, acertaste!", en caso de que el valor sea igual al número inicial;

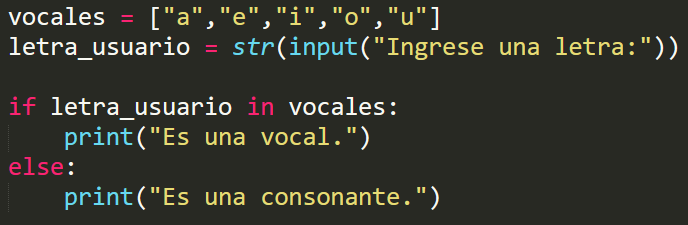
• Imprimir "Muy alto, pruebe un número menor.", en caso de que el valor ingresado sea mayor al número inicial, o imprimir "Muy bajo, pruebe un número mayor." en caso de que el valor ingresado sea menor al número inicial.

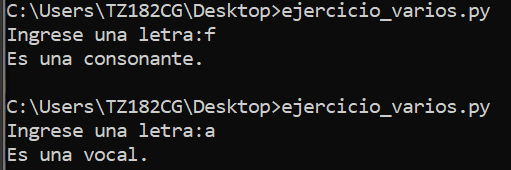
Sugerencia: Anidar condicionales if.





1. Cree un programa que verifique si una letra ingresada es vocal o consonante.



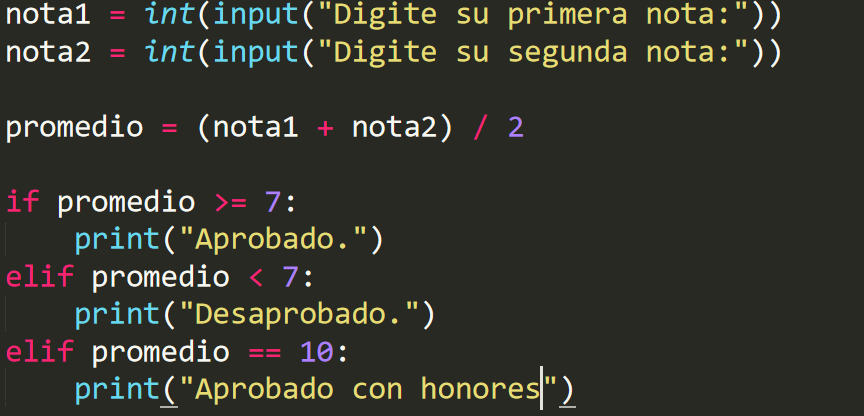


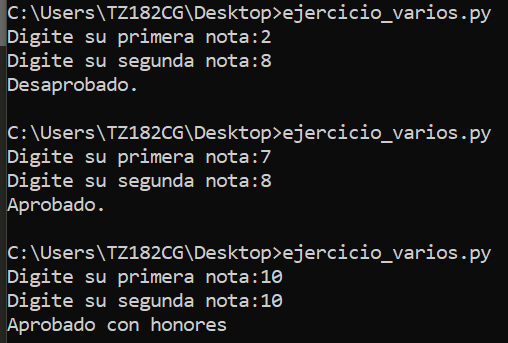
1. • Cree un programa que le pida dos calificaciones a un estudiante. Luego debe calcular el promedio del estudiante y dar el siguiente resultado:

•El mensaje "Aprobado", si el promedio alcanzado es mayor o igual a siete;

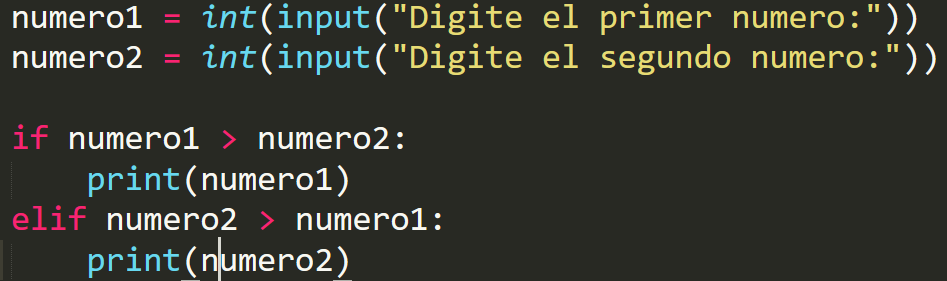
•El mensaje "Desaprobado", si el promedio alcanzado es menor a siete;

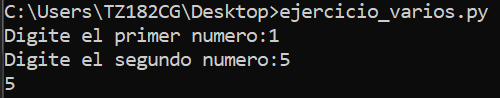
•El mensaje "Aprobado con distinción", si el promedio alcanzado es igual a diez.;



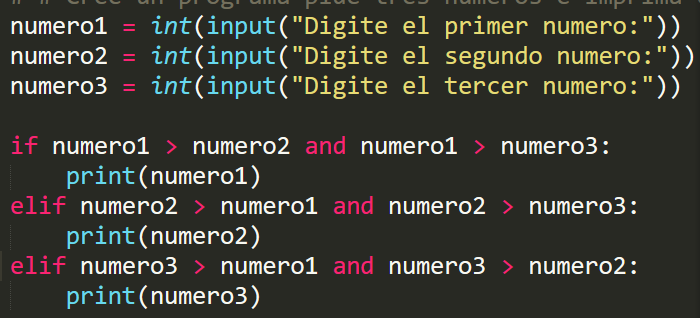


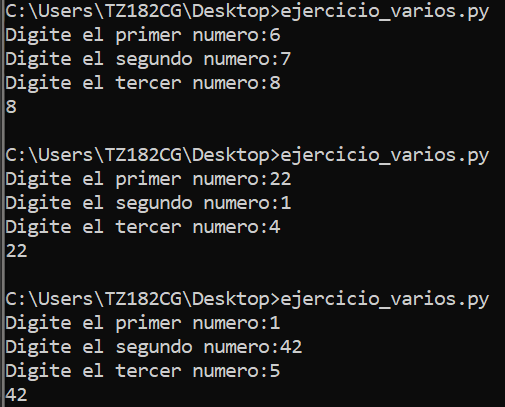
1. Cree un programa pide dos números e imprima el más grande.





1. Cree un programa pide tres números e imprima el más grande.





**SINTAXIS EN PYTHON**

**INDENTACION**

La indentación es un concepto crítico para el desarrollo de un buen código Python.

La indentación o sangra es el espacio del texto en relación a su margen, es decir, el espacio que existe en una línea entre su comienzo y el comienzo del texto, si antes de escribir una instrucción usamos cuatro (4) espacios desde el margen izquierdo hasta la instrucción, podemos decir que la sangría utilizada tiene 4 espacios.

La palabra indentación proviene del inglés *indentation* y es muy común su uso en el código de programación, independientemente del lenguaje utilizado.

En Python, la sangría tiene una función especial, porque **los bloques de instrucción están delimitados por la profundidad de la sangría**, es decir, los códigos que se encuentran pegados al margen izquierdo, serán parte del primer nivel jerárquico. Luego, los códigos que se encuentren a 4 espacios del margen izquierdo, estarán en el segundo nivel jerárquico. Los que estén a 8 espacios, estarán en el tercer nivel, y así sucesivamente.

Todo los **bloques** esta delimitados por la profundidad de la sangría y, por lo tanto, su importancia es vital para un programa de Python. El mal uso de la indentación, por ejemplo, usar 4 espacio en vez de 8 espacios, dará como resultado la no ejecución del código o, peor aún, el mal funcionamiento del mismo.

**BLOQUE**

Los **bloques** son una o más instrucciones que deben ejecutarse una tras otra, de arriba hacia abajo, de izquierda a derecha. Existen varios tipos de bloques, los más comunes son los bloques de código, es decir, bloques que contienen instrucciones de Python. Otro tipo común de bloques son los **bloques de comentarios**, es decir, un conjunto de caracteres que ocupan una o más líneas de código y están delimitados por una notación que ha definido el leguaje de programación. En Python, podemos “comentar” el texto utilizado el símbolo número (#).

Ejemplo:

#Primera condición para revisar si la edad es estrictamente mayor a 16.

if edad > 16:

print(“Felicitaciones, la edad es mayor a 16 años”)

En este caso, la primera línea (precedida con un #) **no será ejecutada** por el intérprete de Python y no será leída como una instrucción. Esto sirve para organizar el código, los objetivos del mismo y para que sea reutilizable por otras personas.

BLOQUES DE CODIGO

El **bloque de código** es una instrucción o un conjunto de instrucción que se encuentran a la misma distancia del margen izquierdo. A continuación, tenemos un fragmento de código Python que muestre el uso de 2 bloques de código. Estos fueron definidos por el número de espacios, es decir, por la distancia que tienen desde el margen izquierdo.

print(“Este es el nivel 0”) #Primer nivel jerárquico.

if True:

print(“Este es el nivel 1”) #Segundo nivel jerárquico.

Es común utilizar 2 espacios o 4 espacios, o incluso 1 tabulación (“*tab*”) en el lado izquierdo de la instrucción para definir en qué bloque está contenida la instrucción. El primer nivel es el nivel 0, es decir, el nivel que no contiene espacios. El idioma no nos obliga a utilizar cierta cantidad de espacios o tabulaciones. Sin embargo, si usamos 4 espacios para definir el primer bloque, el intérprete asumirá que las siguientes instrucciones están inventadas utilizando múltiplos de 4.

Una recomendación es utilizar 1 tabulación en vez de los 4 espacios. En lugar de trabajar con grandes cantidades de espacios, podemos usar una cierta cantidad de tabulaciones, y de esta forma mitigar el posible error por un espacio faltante. El primer nivel jerárquico seria nivel 0, es decir, las instrucciones que no tienen tabulaciones a su izquierda. El segundo nivel usaría una sola tabulación, el tercer nivel usaría 2 tabulaciones y así sucesivamente.

Lo más importante es saber que en Python la **indentación no es opciona**l, es decir, no lo colocamos para que se ve más bonito, sino que estamos obligados a trabajar con algún sistema de indentación para poder definir bloques de código aislados.

Los beneficios de utilizar la tabulación se obtienen rápidamente, porque el código Python siempre está organizado; de lo contrario, no funcionara. Además, los códigos bien indentados se vuelven más legibles y es un estándar utilizado por prácticamente todos los programadores , independientemente del lenguaje en el que están trabajando.

**WHILE**

while es una palabra clave similar a los condicionales pero que representa un ciclo de repetición. Dentro de este “ciclo”, y después de la indentación obligatoria tendremos un bloque de repetición (el código a ejecutar). En particular, **while siempre va a estar acompañada por alguna condición**, donde “ **mientras se siga cumpliendo esa condición**” el bloque de código va a repetirse de principio a fin, hasta que la condición eventualmente deje de cumplirse.

**Nota:** en estos primero ejemplos usaremos solamente “**True**” como nuestra condición (que siempre se va a cumplir) para enfocarnos en la estructura general.

Veamos el siguiente fragmento de código

while True:

pass

Esta es la estructura del while. Observemos la indentación de una tabulación que agregamos después del True y los dos puntos (:). Algunos de los entornos de desarrollo ya agregan las tabulaciones automáticamente para nosotros, lo que hace que la construcción del código sea mucho más sencilla para nosotros. Sin embargo, teneos que familiarizarnos con el concepto para evitar posibles errores y saber que parte de código se debe ejecutar y en qué momento:

Veamos otro ejemplo:

while True:

print(“Bucle de repetición”)

break

break significa “romper”. Esto nos va a permitir escapar del bucle donde se utilice. Como mencionamos, siempre vamos a necesitar una condición para que funcione el bucle. En este caso , el booleano True se utiliza como condición.

Hay que tener cuidado, ya que sin la interrupción (la palabra clave break), el bucle se ejecutara infinitamente o hasta que la computadora crashee, fallara o incluso dejara de funcionar. Esto puede traer problemas en nuestro equipo, pero desde hace un tiempo los IDE’s ya tienen forma de identificar este tipo de bucles y arrojan errores cuando sucede un “bucle infinito”

INCREMENTO Y OTRAS OPERACIONES DE ASIGNACION:

El incremento nos va a ayudar a validar los bucles. ¿Qué es un incremento de hecho? Un incremento es una variable que tiene su valor agregado después de una repetición. Veamos el siguiente ejemplo de en nuestro Prompt de Python.

>>> a = 0

>>> a = a + 1

>>> a = a + 1

>>> a = a + 1

>>> a = a + 1

>>> a

4

Revisemos qué sucedió aquí paso por paso:

1. En la primera línea: definimos la variable a y le asignamos el valor 0.

2. En la segunda línea: a la variable a le asignamos el valor de a (en este caso 0) y le sumamos 1. El valor de a ahora queda como 1.

3. En la tercera línea: a la variable a le asignamos el valor de a (en este caso 1) y le sumamos 1. El valor de a ahora queda como 2.

4. En la cuarta línea: a la variable a le asignamos el valor de a (en este caso 2) y le sumamos 1. El valor de a ahora queda como 3.

5. En la quinta línea: a la variable a le asignamos el valor de a (en este caso 3) y le sumamos 1. El valor de a ahora queda como 4.

6. En la última línea: llamamos a la variable a, que nos devuelve su valor, en este caso, 4.

Parece muy tedioso, pero de hecho, existe un atajo para realizar un incremento, o cualquier operación de asignación. Esto es agregando un operador aritmético seguido de un símbolo igual (=). Veamos el siguiente ejemplo:

>>> b = 0

>>> b += 3

>>> b += 3

>>> b += 3

>>> b += 3

>>> b

12

Entonces un incremento como "a = a + 1", se convertiría en "a +=1"

Los incrementos tendrán un papel fundamental en diversos programas de Python en combinación con bucles. A medida que los vaya utilizando va a percibir sus enormes beneficios

VALIDADNDO BUCLES WHILE

Volviendo al uso de bucle while, hasta l momento teníamos un bucle de repetición que funcionaria hasta el infinito, o hasta que utilicemos una palabra clave como break para romper la repetición y finalizar el código. Pero esto no seria demasiado útil. Es por esto que vamos a utilizar **contadores** con incrementos o asignaciones para manipular en que momento funcionan los bucles while.

Vemos el siguiente ejemplo:

a = 0

while a < 5:

a +=1

print("Sumando 1...")

print(a)

En este caso, sumamos 2 elementos a nuestro código. Por un lado, en la primera línea definimos la variable **a** y le asignamos el valor cero (0). Esto lo vamos a utilizar como un contador. Por otro lado, ahora cambiamos la condición en nuestro **while** para que el código se repita **mientras se cumpla que a es menor a 5.**

Luego, dentro del bucle de repetición, con la indentación obligatoria, definimos un incremento. Esto logra que a medida que el ciclo vaya haciendo repeticiones, vamos a incrementar el valor del contador (la variable a) en 1 en cada repetición. Este ciclo se va a repetir, **siempre y cuando a sea menor que 5**.

En otras palabras, el ciclo se va a ejecutar 5 veces, hasta que el valor de a deje de ser estrictamente menor que 5, se corte el ciclo de repeticiones y continuemos con el código siguiente.

Por último, de nuevo en el nivel 0 de indentación, imprimimos el valor final de a para ver en qué momento se terminó el bucle.

De esta forma podemos combinar contadores y utilizar los bucles while para repetir una serie de instrucciones siempre que se mantenga una condición, y podemos manipular esa condición con un contador.

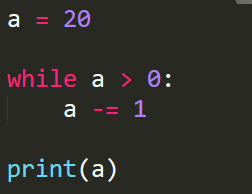
Ejercicios:

1. Cree un programa que contenga un bucle while, y realice las siguientes operaciones:

• Que disminuya el valor de una variable de uno en uno;

• Que rompa el bucle cuando la variable sea menor que cero;

• Asignar el valor inicial de la variable en 20.



2. Cree un programa que contenga un bucle while, y realice las siguientes operaciones:

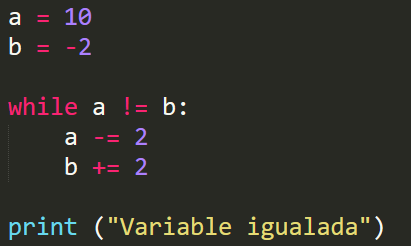
• Que disminuya el valor de una variable de dos en dos;

• Que aumente el valor de otra variable de dos en dos;

• Que imprima en la consola "Variables igualadas.";

• Que rompa el bucle cuando el valor de ambas variables sea igual a 4;

• Asignar el valor inicial de la variable mayor en 10 y el valor inicial de la variable menor en -2.



**FUNICON BOOL (), Truthys y Falsys**

Tenemos dos tipos de booleanos, **True** y **False**. Pero también existen lo que llamamos “**Truthy**” y “**Falsy**”. Traducido seria “cercano a verdadero” y “cercano a falso”. Estos son los otros valores (o tipos de datos) que podemos usar como verdadero y falto. Para eso usaremos la función bool()m que nos va a ayudar a convertir algún tipo de dato a un booleano.

Veamos un poco más en detalle:

El primer código ">>> bool('')", contiene una string vacía. Una string vacía es considerada un Falsy, que cuando lo llevamos al booleano, es False.

El segundo código ">>> bool('#')", contiene una string con un símbolo numeral. Podría ser un texto o un espacio en blanco. Siempre que la cadena contenga algún carácter, se considerará Truthy, con su valor booleano True.

Cuando se trata de números enteros y números de punto flotante, tanto 0 como 0.0 se consideran como False y los demás números son Truthy. Esto sucede porque computacionalmente True es igual a 1, False es igual a 0, y viceversa.

None también es considerado falso.

Ejercicios:

• Cree un bucle while infinito que no utilice la palabra True.

• Cree un bucle while utilizando un carácter considerado Falsy e imprima "Bucle de 1 repetición." y salga del bucle.

• Cree un programa condicional donde el usuario tenga que ingresar un texto. En el caso donde el usuario no coloque ningún texto, imprima "VACÍO", caso contrario, imprimir "LLENO".

• Cree un bucle infinito donde ese bucle tome un texto ingresado por el usuario. En el caso donde el usuario ingrese un texto, rompa el bucle. Caso contrario, continúe repitiendo el bucle.

**LISTAS**

Una lista es una colección de elementos en un orden determinado. Podemos hacer una lista que incluya por ejemplo:

* Las letras del alfabeto;
* Los dígitos del 0 al 9;
* Los nombre de todas las personas de su familia.

Asimismo, los elementos de tu lista no tienen que estar relacionados de ninguna manera particular.

Dado que una lista normalmente contiene más de un elemento, es una buena idea hacer que el nombre de la lista sea plural, como letras, dígitos o nombres.

En Python, los corchetes (**[])** indican una lista y elementos individuales de la lista sestan separados por comas. Veamos el siguiente ejemplo de una lista que contiene algunos tipos de bicicletas:

bicicletas = [‘urbanas’, ‘plegables’, ‘hibridas’, ‘cycloross’]

print (bicicletas)

Si le pedimos a Python que imprima una lista, Python devuelve su representación de la lista incluidos los corchetes:

[‘urbanas’, ‘plegables’, ‘hibridas’, ‘cycloross’]

ACCERDER A LOS ELEMENTOS DE UNA LISTA

Dado que esta es la salida que desean ver los usuarios, veamos como acceder a los elementos individuales de una lista.

Como vimos, las listas son colecciones ordenadas, por lo que podemos acceder a cualquier elemento de una lista indicando a Python la posición, o índice del elemento deseado.

Para tener acceso a un elemento de una lista, escribimos el nombre la lista segundo del índice del elemento entre corchetes.

Por ejemplo si queremos conocer el primer elemento de la lista de bicicletas:

bicicletas = [‘urbanas’, ‘plegables’, ‘hibridas’, ‘cycloross’]

1. Print(bicicletas[0])

La sintaxis para esto se muestra en (1). Cuando pedimos un solo elemento de una lista, Python solo ese elemento sin corchetes:

“urbanas”

Observemos que las posiciones del índice comienzan en 0, no en 1. Esto es así, porque Python considera que el primer elemento de una lista está en la posición 0, no en la posición 1.

Por ejemplo, para tener acceso al cuarto elemento de una lista, debe acceder al elemento en el índice 3. Veamos los siguientes casos para acceder al índice 1 y el índice 3:

bicicletas = [‘urbanas’, ‘plegables’, ‘hibridas’, ‘cycloross’]

Print(bicicletas[1])

Print(bicicletas[3])

Este código devuelve la segunda y cuarta bicicleta de la lista:

Plegables

Cyclocross

Python tiene una sintaxis especial para acceder al ultimo elemento de una lista. Al acceder al elemento en el índice -1, Python siempre devuelve el ultimo elemento de la lista.

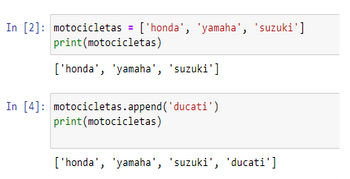
bicicletas = [‘urbanas’, ‘plegables’, ‘hibridas’, ‘cycloross’]

Print(bicicletas[-1])

Este código devuelve el valor ‘cyclocross’. Esta sintaxis es bastante útil, para tener acceso a los últimos elementos de una lista sin saber exactamente cuánto dura la lista.

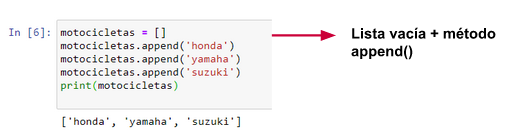
AGREGAR ELEMENTOS A UNA LISTA

Anexar elementos al final de una lista la forma mas sencilla de agregar un nuevo elemento a una lista es anexar el elemento a la lista. Cuando se anexa un elemento a una lista, el nuevo elemento se agrega al final de la lista. Veamos el siguiente ejemplo con una lista de motocicletas, agregamos el nuevo elemento ‘ducati’ al final de la lista, utilizando el método append:



El método append() añade ‘ducati’ al final de la lista sin afectar a ninguno de los otros elementos de la lista [‘honda’, ‘yamaha’, ‘suzuki’]

El método append() facilita la creación dinámicas de listas. Por ejemplo, podríamos comenzar con una lista vacía y, a continuación, agregar elementos a ala lista mediante una seria de llamada a append(). Usando una lista vacía, agreguemos los elementos ‘honda’, ‘yamaha’ y ‘suzuki’ a la lista:



Observe que la lista resultante tiene exactamente el mismo aspecto que la listas de los ejemplos anteriores.

**FUNCION RANGE()**

**Sintaxis**



**Start**: Es la posición de inicio de la secuencia. El valor predeterminado es 0 si no se especifica. Por ejemplo, rango (0,10). Entonces, start = 0 y stop = 10

**Stop**: Genera números hasta un número entero que especifica en que posición detenerse. El rango () nunca incluye el numero de stop en su resultado.

**Step**: especifica el valor de incremento. Cada numero siguiente de la secuencia se genera sumando el valor del step a un numero anterior. EL valor predeterminado es 1 su no se especifica. No es mas que una diferencia entre cada numero en el resultado. Por ejemplo, range(0,6,1). Aquí, step = 1

Para hacer una lista de números, podemos convertir los resultado de range() directamente en una lista utilizando la función list().

Veamos como podemos usar list() para convertir ese mismo conjunto de número en una lista:

números = list(range(1,6))

print(números)

Y este es el resultado: [1, 2, 3, 4, 5]

También podemos usar la función range() para decirle a Python que omita números en un rango determinado. Si pasa un tercer argumento a range(), Python utiliza ese valor como un tamaño de paso al generar números. Por ejemplo, a continuación se indica como enumera los numero pares entre 1 y 10:

even\_number = list(range(2,11,2))

print(even\_numbers)

En este ejemplo, la función range() comienza con el valor 2 y, a continuación, agrega 2 a ese valor. Suma 2 repetidamente hasta que alcanza o pasa el valor final, 11, y se produce este resultado:

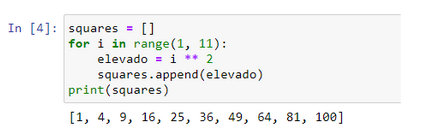
[2, 4, 6, 8, 10]

Notemos que podemos crear casi cualquier conjunto de número que deseemos utilizando la función range().

Por ejemplo. Observemos como podemos crear una lista de los primero s10 numero cuadrados (es decir, el cuadrado de cada entero del 10 al 10)

Recordemos que en Python, dos asteriscos (\*) representan exponentes.

Así es como podemos poner los primero 10 números cuadrados en una lista:



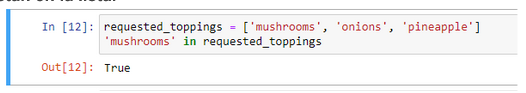
**COMPROBACION DE SI UN VALOR ESTA O NO ESTA EN UNA LISTA**

A veces, es importante comprobar si una lista contiene un determinado valor antes de realizar una acción. Por ejemplo:

Es posible que nos interese comprobar si ya existe un nuevo nombre de usuario en una lista de nombre de usuario actuales antes de completar el registro de alguien en nuestro sitio web.

*Para saber si un valor en particular ya esta en una lista, usamos la palabra clave* ***in.***

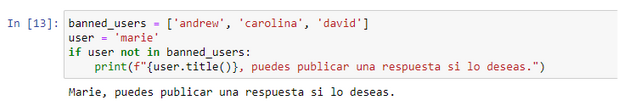
Supongamos una llista que contiene los ingredientes que un cliente solicito para una pizza. Luego, observamos como verificamos si ciertos ingredientes están en la lista-



**COMPROBACION SI UN VALOR NO ESTA EN UNA LISTA**

En otras ocasiones es importante saber si un valor no aparece en una lista. Para esto podemos utilizar la palabra clave **not in** para esta situación.

Por ejemplo, considere una lista de usuarios a los que se les prohíbe comentar en un foro. Puede verificar si un usuario tiene prohibido comentar antes de permitir que esa persona envíe un comentario.

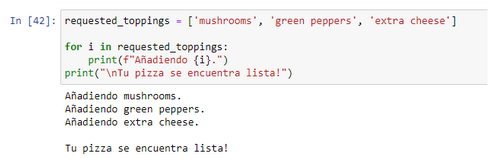


Si el valor de usuario no esta en la lista **banned\_users**, Python devuelve **True** y ejecuta la línea con sangría. El usuario ‘**marie’** no esta en la lista **banned\_users**, por lo que ve un mensaje que la invita a publicar una respuesta.

**AL USAR SENTENCIAS IF CON LISTAS PODEMOS**

* Observar valores especiales que deben tratarse de manera diferente a otros valores de la lista
* Administrar las condiciones cambiantes de manera eficiente, como la disponibilidad de ciertos artículos en un restaurante durante un turno
* También podemos comenzar a demostrar que nuestro código funciona como se espera en todas las situaciones posibles

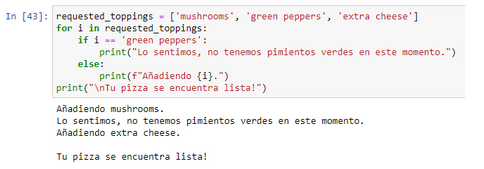
Continuemos con el ejemplo de la pizzería. La pizzería muestra un mensaje cada vez que se agrega un aderezo a su pizza, mientras se prepara. El código para esta acción se puede escribir haciendo una lista de ingredientes que el cliente ha solicitado y usando un bucle para anunciar cada ingrediente a medida que se agrega a la pizza.



**USAR SENTENCIAS IF CON LISTAS**

Supongamos que la pizzería se queda sin pimientos verdes.

Una declaración **if** dentro del **bucle** **for** puede manejar esta situación de manera apropiada:



* Esta vez verificamos cada articulo solicitando antes de agregarlo a la pizza.
* El código if i == ‘green peppers’ verifica si la persona solicito pimientos verdes. Si es así, mostramos un mensaje informándoles por que no pueden comer pimientos verdes
* El bloque else asegura que todos los demás ingredientes se agregan a la pizza
* El resultado muestra que cada solicitud se maneja de manera adecuada

**USAR VARIAS LISTAS**

La gente pedirá cualquier cosa, especialmente cuando se trata de toppings para pizza ¿Qué pasa si un cliente realmente quiere papas fritas en su pizza?

El siguiente ejemplo define 2 listas:

* La primera es una lista de ingredientes disponibles en la pizzería, y
* La segunda es la lista de ingredientes que el usuario ha solicitado.
* Esta vez, cada elemento de **topping solicitados** se compara con la lista de **toppings disponibles** antes de agregarlo a la pizza.

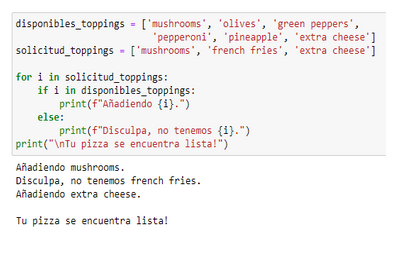
Ejemplo:

* Primero  **definimos una lista de ingredientes disponibles en esta pizzería**. Tenga en cuenta que esto podría ser una tupla si la pizzera tiene una selección estable de ingredientes.
* Luego, **hacemos una lista de ingredientes que un cliente ha solicitado**. Tengo en cuenta la solicitud inusual, ‘papas fritas’.
* Después recorremos la lista de ingredientes solicitados **(for i in solicitud\_toppings)**



Ejemplo:

* Dentro el ciclo, primero **verificamos i cada ingrediente solicitado esta realmente en la lista de ingredientes disponibles**. Si es así, agregamos esa cobertura a la pizza.
* Si la cobertura solicitada no esta en la lista de ingredientes disponibles, el **bloque else** ejecutara.
* El **bloque else** imprime un mensaje que le indica al usuario que ingredientes no está disponibles



**FUNCIONES**

Una función es un conjunto de instrucciones que podemos reutilizar. Es como si fuera una variables pero no lo es la principal diferencia se encuentra que con una variable buscamos almacenar un valor, mientras que con una función buscamos almacenar instrucciones. Es un nombre que reservamos para realizar acciones, o acceder al código que queremos reutilizar.

Imaginemos que cada aves que queremos imprimir algo en la consola, debemos especificar cientos de líneas de código, llamando a los diferentes elemento computacionales para lograr que nuestro computador imprima un mensaje. Esto era así en los primeros lenguajes de programación. Un día, los programadores pensaron: ‘¡Esto es muy tedioso! Mejor buscar una mejor forma de reutilizar este código para imprimir, de lo contrario tendré que repetir todas estas líneas de código para imprimir, de lo contrario tendré que repetir todas estas líneas de código cada vez que quiera imprimir un solo mensaje.’. Entonces decidieron llamar a este conjunto de instrucciones print().

**¿Por qué dar nombre a bloques de código?** Principalmente para no repetir la misma lógica innecesariamente. Además, nombrar bloques nos facilita poder modificar, corregir e incluso implementar código nuevo. Imaginémonos, en el ejemplo anterior , encontrar un error en las cientos de líneas que nos permitían imprimir un mensaje en la consola, para luego corregirlo en cada iteración donde queríamos imprimir.

Como su nombre lo indica, las funciones tienen una “**función”** determinada. **print()**, por ejemplo, imprime un mensaje en la consola, mientras que **len()** devuelve el tamaño de una cadena o lista. **Cada función tiene una funcionalidad especifica**.

**DEFINIENDO UNA FUNCION**

Para definir una función, primero debemos anteponer la palabra reservada **def**, luego el nombre que quedemos darle a nuestra función, luego agregar los argumentos entre paréntesis ( () ), y por último, ingresar al siguiente bloque de código con un símbolo de dos puntos :.

Así definimos una función:

**def** mi\_codigo **():**

**print(‘**Esta es mi primera función**’)**

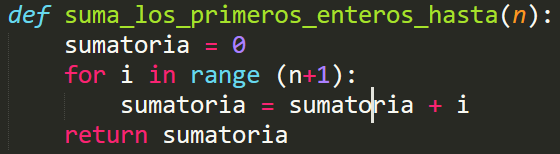
Las *best practices* para las funciones son las mismas que para las variables.

La p cuyo nombre es ‘mi\_codigo’.La palabra reservada **def** representa la intención de **definir** una ‘función’ cuyo nombre es ‘mi\_codigo’. Luego, continuamos con un par de paréntesis y dos puntos ,Dentro del bloque de código que le sigue, escribimos las instrucciones que queremos ejecutar y/o reutilizar. En el ejemplo anterior, estamos imprimiendo un mensaje.

Para usar la función, y acceder al código que queremos reutilizar, necesitamos “llamarla”, y esto se hace de la siguiente forma:

mi\_codigo()

Ejemplo:



¿Qué está pasando aquí?

En la primera línea notaremos que agregamos el parámetro **n**. Esto indica que aquello que el usuario ponga entre paréntesis va a interactuar con las instrucciones que vienen a continuación. Efectivamente, hicimos que el comportamiento de la función dependiera de la información que le pasamos a la función, que en este caso llamamos n y colocamos entre los paréntesis que siguen al nombre de la función.

Luego, definimos una variable local sumatoria, que también va a ser parte de nuestro bloque de instrucciones. Le asignamos como valor inicial **0**.

Luego, en la línea "**for i in range(n+1):**" hicimos varias cosas. La función **range()** nos brinda una secuencia de números desde **0**, incrementando de **1** en 1 y termina antes de llegar al número que especificamos dentro de la función. En nuestro caso, "**n+1**" indica que queremos que la función también contabilice a nuestro parámetro n. En otras palabras, range(**n+1**) va a devolver todos los números desde **0** hasta **n** (inclusive).

En esa misma línea, utilizamos un bucle for para iterar por esa secuencia de números. Y le asignamos la letra i a los elementos de esa secuencia, para poder referenciarlos en el siguiente bloque de código. Esta es una lógica similar a cuando definimos el parámetro como "**n**" para poder referenciarlo dentro de nuestro código más adelante.

Dentro de ese bucle for, en un nivel de indentación más adentro, pedimos que para cada número "**i**"", dentro de la secuencia de **0** a **n**, sumemos ese número i a la variable "**sumatoria**".

Hasta aquí la lógica de la función sería algo así:

**definimos la función** suma\_los\_primeros\_enteros\_hasta(**un parámetro que llamaremos "n"**):

sumatoria = 0

**para todos los números entre 0 y n que llamaremos "i":**

**sume el valor de ese número que llamamos "i" a la variable "sumatoria"**

**devuelva** sumatoria

RETURN:

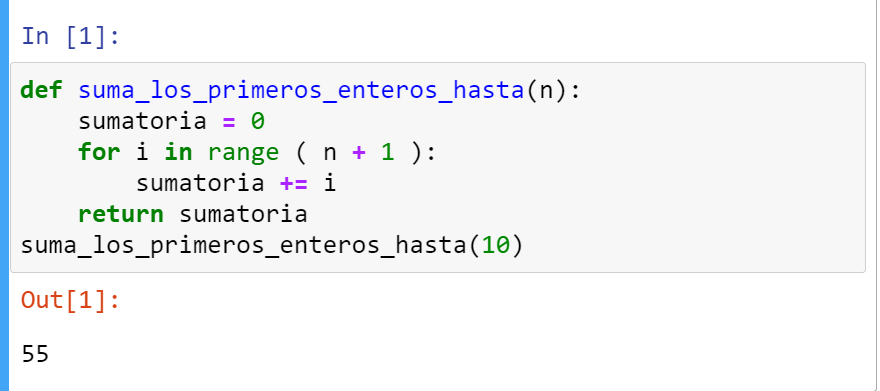
Notaremos que en el primer ejemplo de este bloque, cuando definimos mi\_codigo(), nos referimos a este como una función entre comillas. Esto es porque realmente esto era un **procedimiento**.

La diferencia entre un **procedimiento** y una **función** es el uso de la palabra reservada **return**.

Un **procedimiento** es una serie de instrucciones o líneas de código reutilizables, pero que no devuelven un resultado. Solamente ejecutan las instrucciones que tiene adentro. En nuestro caso, el procedimiento “mi\_codigo()” reutilizaba nuestra impresión de un mensaje.

Una **función**, por otro lado, **tiene el objetivo de devolver un resultado.** Esto es indicado con la palabra reservada **return** y puede ser todo tipo de objetos, en nuestro ejemplo, la función “suma\_los\_primero\_enteros\_hasta()” **devolvía** el resultado final de la sumatoria, después de haber sumado el valor de todos los números del 0 hasta n. En general, la palabra **return** se coloca al final de la función para devolver el resultado final de las instrucciones que realizamos.

**Practica:**



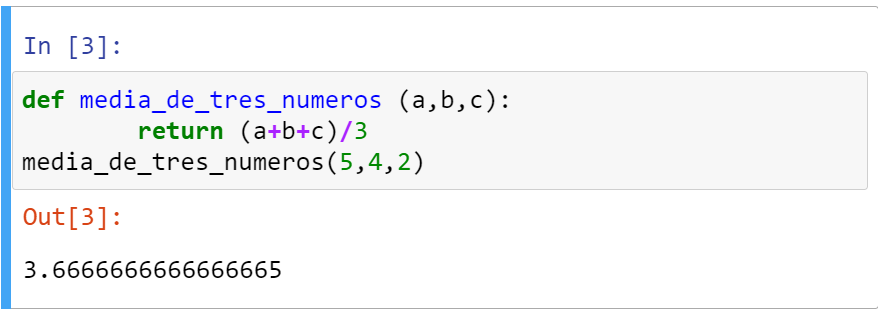
**NUMERO DE PARMETROS**

Como vimos en el ejemplo anterior un **parámetro** es **una variable que definimos dentro de los paréntesis en la creación de una función, para que interactúe con el código dentro de dicha función.**

Ahora, las funcione pueden no tener ningún parámetro, tener un parámetro, o tener mas de un parámetro. Este detalle solamente va a indicar cuantas variables queremos que interactúen con las instrucciones definidas dentro de la función.

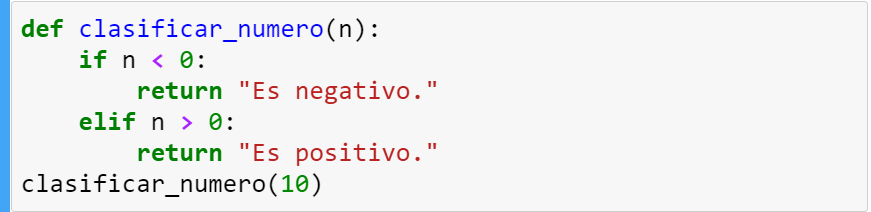
Para definir más que un parámetro solo basta con separar los parámetros con una como (,) dentro de los paréntesis en la definición de la función:

Un ejemplo de una función con múltiples parámetros:



**Ejercicios:**

**• Defina una función que reciba un número como parámetro e imprima un mensaje si ese número es positivo, y otro mensaje si ese número es negativo.**



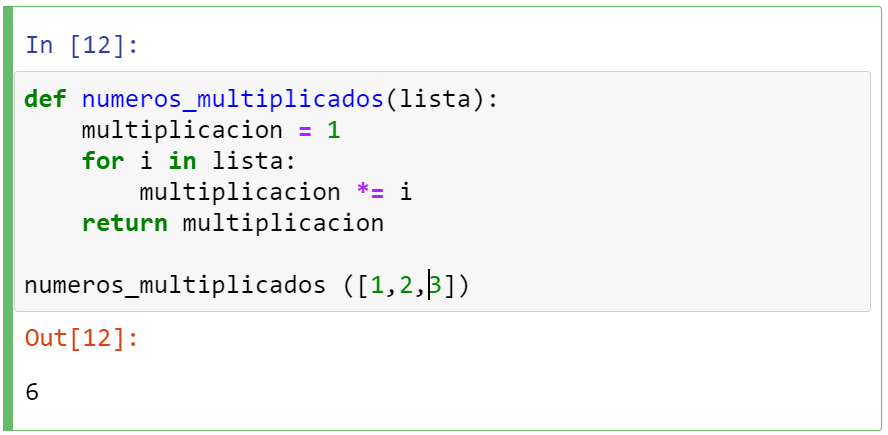
**• Defina una función que sea capaz de devolver el valor máximo entre 3 números.**



**• Defina una función sume todos los valores de una lista de números y devuelva el valor de la suma.**



**• Defina una función que multiplique todos los valores de una lista de números y devuelva el valor total.**

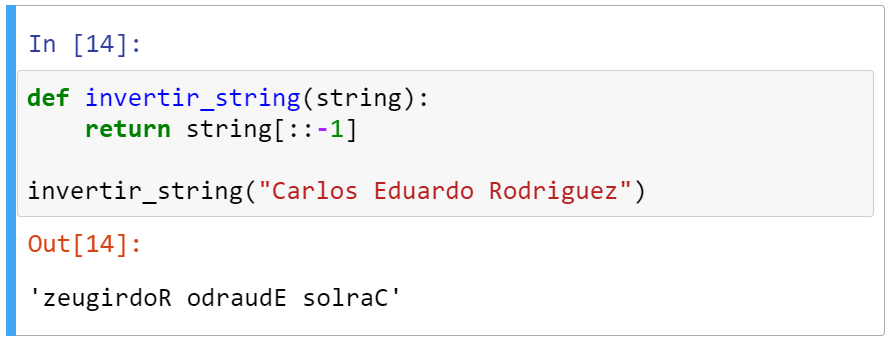


**• Defina una función "invertir()" que imprima un string que le pasemos como parámetro, de atrás hacia adelante.**

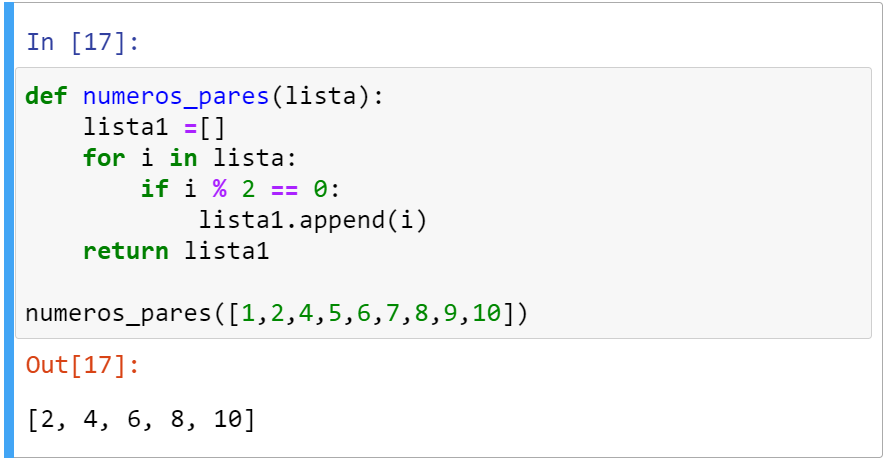
**Por ejemplo:**

**>>> invertir("Digital"):**

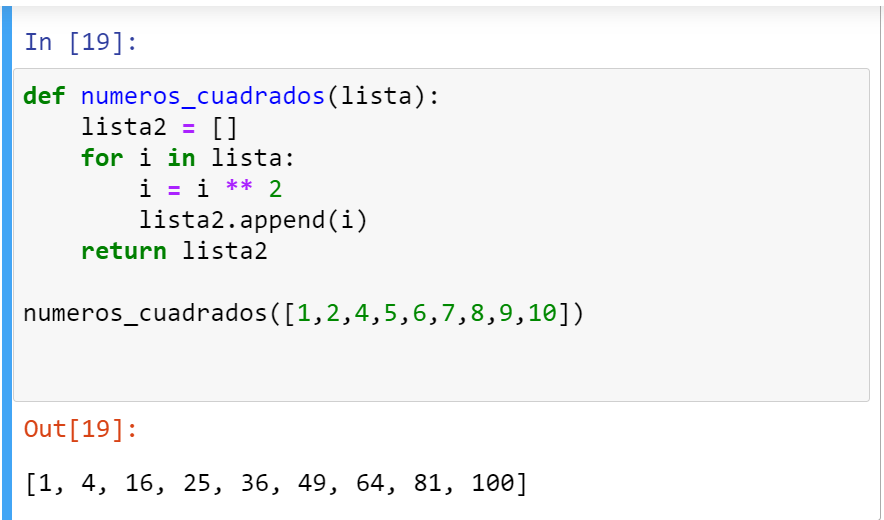
**latigiD**



**• Defina una función que imprima solamente los números pares de una lista de números pasada como parámetro.**



**• Defina una función que reciba una lista de números como parámetro y devuelva la misma lista pero con todos los valores elevados al cuadrado.**



**PARAMETROS Y ARGUMENTOS**

Vimos anteriormente que las funciones pueden cambiar su comportamiento en función de la información que les transmitimos. Esta información se llama “**Parámetros**” o ”b”. Por lo general, estas dos palabras se usan indistintamente, por a veces es importante distinguirlas.

El **parámetro** es el nombre de la variable que representa la información que interactúa con nuestra función, en el momento de definir la función

El **argumento** es el valor contenido en esa variable o, dicho de otra forma, el valor que le paso el usuario de la función.

Un ejemplo hará evidente la distinción:

>>> def imprimir\_mas\_bonito(dia, mes, año):

print(dia, mes, año, sep = '/')

>>>

>>> imprimir\_mas\_bonito(1, 4, 2021)

1/4/2021

En el ejemplo anterior, día, mes y año son **parámetros** de la función imprimir\_mas\_bonito(). Por otro lado, los valores 1,4 y2021 son **argumentos**.

**FIRMA DE FUNCION**

Otra terminología útil para conocer es la **firma** de una función. Esta se refiere al conjunto de parámetros (a veces con sus tipos identificados) utilizados por la función y sus documentaciones. Por ejemplo la firma de la función imprimir\_mas\_bonito() consta de tres parámetros numéricos

**Actividad**

Utilice la funcion **help()** para verificar la firma de distintas funciones built-in de Python (len(), range(),etc)

**ARGUMENTOS POSICIONALES Y NOMBRADOS**

Hay dos tipos de argumentos en Python: **posicionales** y **nombrados**. Consideremos este ejemplo:

>>> from math import pow

>>> pow(2, 3)

8

En otras palabras, pow(2, 3) calcula 2 elevado a la potencia de 3, esto da como resultado 8. ¿Cómo sabe Python que esta es la cuenta correcta, en lugar de 3 elevado a 2 (de resultado 9)?

Esto se debe por el orden de los argumentos, y es precisamente por eso que se dice que estos argumentos son **posicionales.**

Probablemente podemos suponer que la función pow definida de forma similar a esta:

def pow (n,m)

return n\*\*m

En este caso, en el de los argumentos posicionales, el primer argumento pasado por el usuario, será lo que la función tome como parámetro **n** y el segundo argumento, se convertirá en **m**.

Pero existe otra forma de pasar argumentos a las funciones. Vea el ejemplo a continuación:

>>> print (‘¡Hay novedades aquí’/end=’!’)

¡Hay algo nuevo aquí!>>>

En el ejemplo anterior, además del mensaje a imprimir, pasamos a la función print() un argumento llamado **end**, que estipula como termina la impresión en el intérprete. El valor predeterminado es con un carácter que representa el salto de línea (“/n”), por lo que, cada vez que usemos print(),, generalmente el cursor suele moverse a la siguiente línea. Pero en este caso lo reemplazamos con una exclamación (“!”) y es por eso que se vuelve un poco desordenado después del mensaje.

La ventaja de los argumentos nombrados es que no dependen de la posición como los argumentos posicionales. Esto los vuelve muy interesantes para utilizar en funciones con firmas complejas. La función **print()** todavía sirve para ejemplificar esto, ya que acepta otro argumento nombrado llamado “**sep**”, que establece como conecta una secuencia de mensajes a imprimir:

Ejemplos:

>>> print('Hay novedades aquí', 'las noticias no se detienen', sep = ' y ', end = '!\n')

Hay novedades aquí y las noticias no se detienen!

>>>

¡Experimente! Intente invertir el orden de "sep" y "end" dentro de la función:

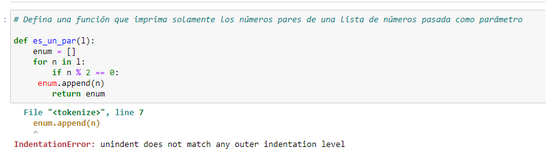
>>> print('Hay novedades aquí', 'las noticias no se detienen', end = '!\n', sep = ' y ')

Hay novedades aquí y las noticias no se detienen!

>>>

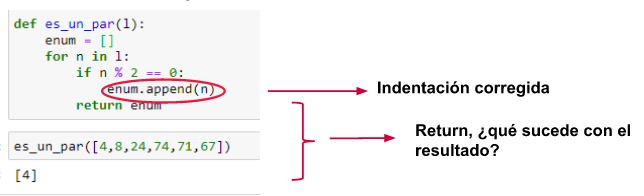
**INDENTATION ERROR**

Antes de que Python ejecute cualquier código en su programa, primero determinara el padre y los hijos correctos de cada línea. Python produce un *IndentationError* cada vez que se encuentra con una línea para la que no puede determinar el padre correcto para asignar. Por ejemplo:



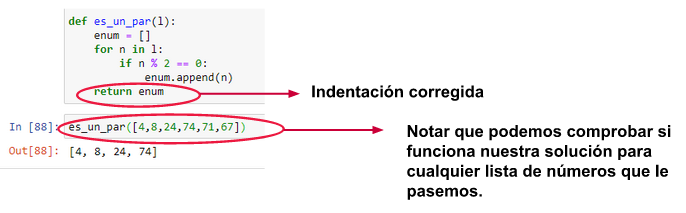
Python no pudo determinar si el padre de la línea enum.append(n) era la línea if.

Para solucionar este problema:



*Notar que la lista de números que pasamos no nos devuelve todos los números pares de la lista*

Para solucionar este problema:



*Ahora sí, la lista de números que pasamos nos devuelve todos los numero pares de la lista.*

**PROGRAMACION FUNCIONAL**

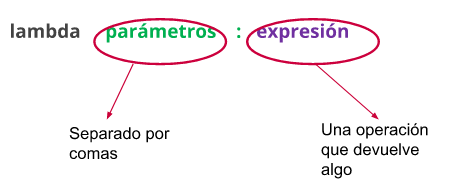
Hasta ahora, siempre hemos dados nombres a nuestras funciones usando la palabra clave **def**. Sin embargo, existe una clase especial de funciones para las que no es necesario especificar nombre de funciones.

Un **lambda** es una función anónima que devuelve algún tipo de datos.

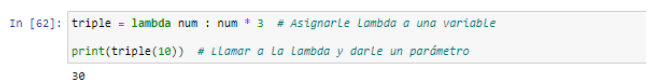
Las lambdas se definen mediante la palabra clave **lambda**. Dado que devuelven datos, resulta una buena practica asignarlos a una variable.

**Sintaxis #**

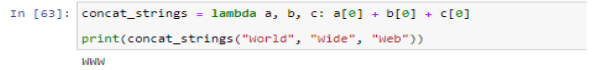
La siguiente sintaxis para crear lambdas:



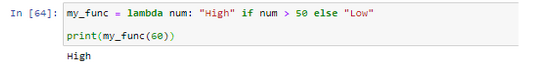
* Podemos encontrar una lambda que triplica el valor del parámetro y devuelve este nuevo valor:



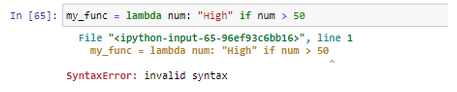
* Otro ejemplo de lambda que concatena los primeros caracteres de tres cadenas:



* Como podemos ver, las lambdas son mas simples y legibles que las funciones normales. Pero esta simplicidad viene con limitación.
* Una lambda no puede tener una expresión de varias líneas. Esto significa que nuestra expresión debe ser algo que se pueda escribir en una sola línea.
* Por lo tanto, las lambdas son perfectas para funciones cortas de una sola línea.
* También podemos usar declaraciones condicionales dentro de lambdas:



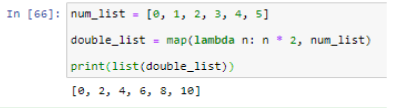
* Cuando se usan declaraciones condicionales en lambdas, el par **if-else** es necesario. Ambos casos deben cubrirse, de lo contrario la lambda arrojara un error:



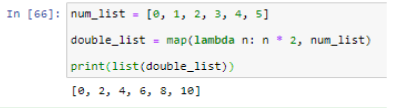
* La función **map()** incorporada crea un objeto de mapa usando una lista existente y una función como sus parámetros. Este objeto se puede convertir en una lista usando la función **list()**
* La sintaxis para **map()** es la siguiente:
  + **map** (**función**, **lista**)

*La función se aplicara a todos los elementos de la lista*

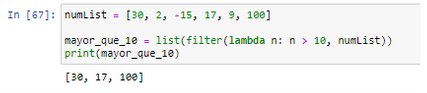
* A continuación, usaremos **map()** para duplicar los valores de una lista existente:



* Esto crea una nueva lista. La lista original permanece sin cambios.
* Podríamos haber creado una función que duplique un numero y usarla como argumento en **map()**, pero la lambda simplifico las cosas



* Otro ejemplo similar es la función **filter()**. Requiere una función y una lista.
* **filter()** filtra elementos de una lista si los elementos satisfacen la condición que se especifica en la función de argumento.
* Escribamos una función **filter()** que filtre todos los elementos que son mayores de 10:



* La función devuelve un objeto de filtro que se puede convertir en una lista usando **list()**
* Notemos que al igual que **map()**, **filter()** devuelve un objeto sin cambiar la lista original.