



# ESTRUCTURAS CONDICIONALES

Toma de decisiones

01 {

## [Continuación]

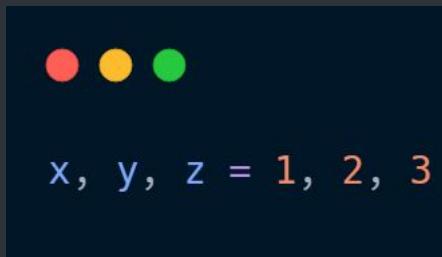
- Asignación múltiple de variables
- Función print().
- Operadores aritméticos. Sintaxis abreviada
- Comentarios
- Caracteres especiales

}

# Asignación múltiple de variables {

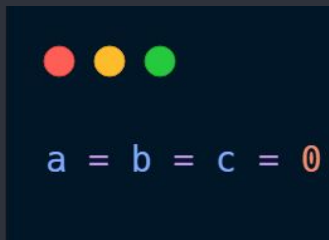
Se pueden **declarar varias variables en una sola línea** de código en Python. Esto se conoce como "asignación múltiple" o "asignación en paralelo".

Para hacerlo, se separan los nombres de las variables con comas y se les asigna un valor separado por comas en el mismo orden.



# Asignación múltiple de variables {

También es posible **asignar el mismo valor a varias variables** en una sola línea de código.



Es importante tener en cuenta que al utilizar la asignación múltiple, el **número de variables debe ser igual al número de valores que se están asignando**. Si el número de variables y valores no coincide, se producirá un **error**.

# Función print(). String multilínea {

En Python, se pueden **imprimir strings multilínea** utilizando la función print().

Para hacerlo, se utiliza una cadena de texto que contiene varias líneas, y se utiliza la sintaxis de **tres comillas simples (''' ) o dobles (""")** al principio y al final de la cadena para indicar que la cadena contiene varias líneas.

```
print('''Este es un ejemplo de
      un string multilínea
      en Python''')
```

```
print(f'''
Este es un ejemplo de
    un string multilínea
        en Python. Contiene {3} líneas
''')
```

# Función print(). Salto de línea {

`\n` es un carácter especial que representa un salto de línea. Se utiliza dentro de strings para indicar que se debe insertar un salto de línea en un punto específico del texto.

Al imprimir o mostrar en pantalla un string que contiene el carácter `\n`, Python interpreta este carácter especial y genera un salto de línea en el lugar donde se encuentra.



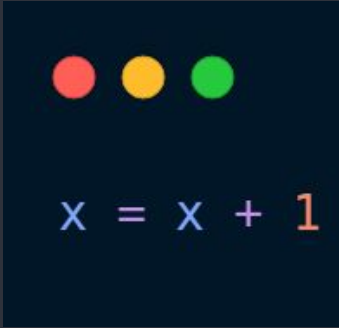
```
print("Resultado:\n32")
```

>>> Resultado:


32

# Operadores. Sintaxis abreviada {

En Python, se puede utilizar la **sintaxis abreviada** para realizar operaciones aritméticas y asignar el resultado a una variable. Por ejemplo:



```
x = x + 1
```



```
x += 1
```

# Comentarios {

En Python, los comentarios son **líneas de código que se utilizan para documentar y explicar el funcionamiento del código**. Los comentarios **se ignoran por el intérprete** de Python y no tienen ningún efecto en la ejecución del programa.

En Python, se pueden crear comentarios utilizando el símbolo **#**. Todo el texto que sigue al símbolo **#** en una línea de código se considera un comentario y se ignora por el intérprete.



```
# Este es un comentario  
print("Hola, mundo!") # Este también es un comentario
```



# Comentarios {

También se pueden utilizar comentarios para **desactivar temporalmente una línea de código**.

Por ejemplo, si se desea evitar que una línea de código se ejecute, se puede comentar la línea para que sea ignorada por el intérprete.



```
# x = 10  # Esta línea se comenta temporalmente  
y = 5  
y *= 3  # Esta línea se ejecuta normalmente
```

# Comentarios {

En Python, también es posible crear **comentarios de varias líneas** utilizando las comillas triples (''' o ''').

Este tipo de comentarios se conocen como comentarios multilínea o docstrings, y se utilizan para documentar funciones, módulos y clases.



```
'''
```

```
Este es un comentario de varias líneas.
```

```
Se utiliza para documentar funciones, módulos y clases.
```

```
Puede utilizar comillas simples o dobles
```

```
para crear un comentario multilínea.
```

```
'''
```



```
'''
```

```
x = 20
```

```
y = 1
```

```
x *= y
```

```
'''
```

# Caracteres especiales {

En Python, los caracteres especiales son **caracteres que tienen un significado especial** dentro de una cadena de texto.

Estos caracteres se utilizan para representar caracteres que no se pueden escribir directamente en una cadena, como **saltos de línea, tabulaciones, comillas y caracteres no imprimibles**.

}

# Caracteres especiales {

## Salto de línea

`\n`

```
cadena = 'Hola\nmundo'
print(cadena)
# Imprime:
# Hola
# mundo
```

## Tabulación

`\t`

```
cadena = 'Nombre:\tAntonio\nApellido:\tPerez'
print(cadena)
# Imprime:
# Nombre:    Juan
# Apellido:   Pérez
```

## Comilla simple

`\'`

```
cadena = 'Ella dijo: \'Hola, ¿cómo estás?\'
print(cadena)
# Imprime:
# Ella dijo: 'Hola, ¿cómo estás?'
```

## Comilla doble

`\"`

```
cadena = "Mi libro favorito es \"Cien años de soledad\""
print(cadena)
# Imprime:
# Mi libro favorito es "Cien años de soledad"
```

}

02 {

## [Condicionales]

- Condicionales
- Operadores de comparación
- Operadores lógicos
- Sentencia if
- Indentacion
- Valores falsos

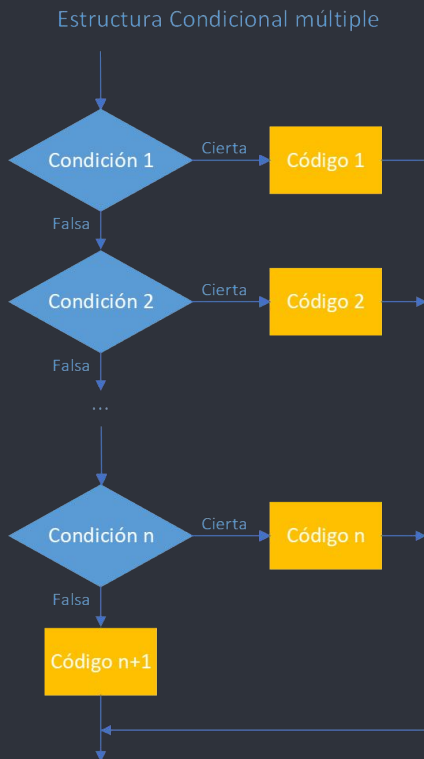
}

# Condicionales {

Una estructura condicional, también conocida como branching, es una **forma de dirigir el flujo del programa**.

En Python, los condicionales son estructuras de control que permiten ejecutar diferentes bloques de código según **se cumplan o no ciertas condiciones**.

}



# Operadores de comparación {

En Python, existen varios operadores de comparación, que se utilizan para **comparar dos valores y evaluar si una determinada relación es verdadera o falsa**.

Estos operadores devuelven un **valor booleano** (True o False) según se cumpla o no la relación de comparación.

}

# Operadores de comparación {

[ > ]

Mayor que

● ● ●

```
x = 5  
y = 10
```

```
print(y > x) # True
```

[ < ]

Menor que

● ● ●

```
x = 5  
y = 10
```

```
print(y < x) # False
```

[ >= ]

Mayor o igual que

● ● ●

```
x = 5  
y = 5
```

```
print(y >= x) # True
```

[ <= ]

Menor o igual que

● ● ●

```
x = 5  
y = 30
```

```
print(y <= x) # False
```

}



# Operadores de comparación {

[ == ]

Igual que

● ● ●

```
x = 5  
y = 5
```

```
print(y == x) # True
```

[ != ]

Distinto que

● ● ●

```
x = 5  
y = 7
```

```
print(y != x) # True
```

[ >= ]

Mayor o igual que

● ● ●

```
x = 5  
y = 5
```

```
print(y >= x) # True
```

[ <= ]

Menor o igual que

● ● ●

```
x = 5  
y = 30
```

```
print(y <= x) # False
```

}

# Operadores lógicos {

En Python, existen tres operadores lógicos: **and**, **or** y **not**. Los operadores lógicos se utilizan para **combinar expresiones booleanas y evaluar si una expresión es verdadera o falsa**.


Es importante tener en cuenta que los operadores lógicos se evalúan en un **orden específico: primero se evalúa not, luego and y finalmente or**. Si se desea cambiar el orden de evaluación, se pueden utilizar **paréntesis** para agrupar las expresiones booleanas de manera adecuada.

}

# Operadores lógicos {

## and

Devuelve **True** si ambas expresiones booleanas son verdaderas. Si alguna de las expresiones es falsa, el operador devuelve **False**.




```
x = 5
y = 7

print(x > 0 and y > 0) # True
```

## or

Devuelve **True** si al menos una de las expresiones booleanas es verdadera. Si ambas expresiones son falsas, el operador devuelve **False**.



```
x = 5
y = 7

print(x > 100 or y > x) # True
```

# Operadores lógicos {

## not

Devuelve el valor opuesto de la expresión booleana. Si la expresión es **True**, el operador devuelve **False**, y si la expresión es **False**, el operador devuelve **True**.

```
x = 5  
y = 7
```

```
print(not x > 10) # True
```

# Sentencia if {

En Python, **if** es una sentencia condicional que se utiliza para **ejecutar un bloque de código si se cumple una determinada condición**.

La sintaxis básica de la **sentencia if** en Python es la siguiente:




```
if condicion:
    # Código a ejecutar si la condición es verdadera
```

# Sentencia if {

**Si la condición es verdadera**, el bloque de código indentado después de la sentencia if se ejecuta.

**Si la condición es falsa**, el bloque de código se salta y la ejecución continúa con la siguiente sentencia después del bloque if.



```
x = 15
```

```
if x > 10:  
    print('x es mayor que 10')
```

## Ejemplo {

Debe desarrollar un programa que indique **si un número es par**. Se calcula el **residuo de la división entre el número y el 2**, si es igual a 0 entonces el número es par.



```
numero = int(input("Ingrese un numero: "))
if (numero % 2) == 0: # Si el resto obtenido es 0 entonces numero es par
    print(f"El numero {numero} es par.")

print("Fin del programa.")
```

}

# Indentación {

La indentación en Python es la forma en que se estructura el código para **indicar los bloques de código que están relacionados entre sí**.

En lugar de utilizar llaves o palabras claves para delimitar los bloques de código, como en otros lenguajes de programación, en Python se utiliza la **indentación**.

La indentación se realiza mediante la **inserción de espacios o tabulaciones al comienzo de las líneas** de código. Se recomienda utilizar **cuatro espacios para cada nivel** de indentación.

}



# Indentación {

La indentación es una forma de mejorar la legibilidad y la claridad del código, y es una parte fundamental de la sintaxis de Python.

Por ejemplo, en la siguiente estructura `if` en Python, se utiliza la indentación para indicar el bloque de código que se ejecuta si la condición es **verdadera**:



```
if x > 10:  
    print('x es mayor que 10')
```

# Valores falsos {

Se consideran **falsy values** (valores falsos) aquellos valores que se evalúan como **False** en un contexto booleano. Esto significa que, aunque estos valores no son iguales a **False**, se consideran equivalentes a **False** para propósitos booleanos.

}

# Valores falsos {

**False** El valor **booleano False**.

**None** El valor especial None, que representa **la ausencia de valor**.

**0** El valor **entero cero**.

**0.0** El valor **flotante cero**.

**''** La **cadena vacía** (una cadena sin caracteres).

}

# Valores falsos {

4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14

[ ]

La **lista vacía** (una lista sin elementos).

( )

La **tupla vacía** (una tupla sin elementos).

{ }

El **diccionario vacío** (un diccionario sin elementos)

set( )

El **conjunto vacío** (un conjunto sin elementos).

}

# Valores falsos {

Es importante tener en cuenta que, aunque estos valores se consideran falsy en un contexto booleano, **no son equivalentes a False en otros contextos.**

Por ejemplo, la cadena vacía (') no es igual a False en una comparación de igualdad (=).



```
if not '':  
    print('La cadena esta vacía')
```

03 {

## [Estructuras Condicionales]


- Sentencia else
- Sentencia elif
- Valores verdaderos

}

# Sentencia else {

En Python, la **sentencia else** se utiliza en combinación con la sentencia if para crear una estructura condicional más compleja. La sentencia else se ejecuta **si la condición del if es falsa**.

La sintaxis básica de la sentencia if-else en Python es la siguiente:



```
if condicion:
    # Código a ejecutar si la condición es verdadera
else:
    # Código a ejecutar si la condición es falsa
```

# Sentencia else {

**Si la condición es verdadera**, se ejecuta el bloque de código indentado después de la sentencia if.

**Si la condición es falsa**, se ejecuta el bloque de código indentado después de la sentencia else.



```
x, y = 5, 7
```

```
if x == y: # False
    print('x es igual a y')
```

```
else:
    print('x no es igual a y')
```



# Sentencia elif {

En Python, la **sentencia elif** (abreviatura de "else if") se utiliza en combinación con la sentencia if para **evaluar múltiples condiciones y crear una estructura condicional más compleja**.


La sintaxis básica de la sentencia if-elif-else en Python es la siguiente:

```
if condicion1:
    # Código a ejecutar si la condición1 es verdadera
elif condicion2:
    # Código a ejecutar si la condicion1 es falsa y la condicion2 es verdadera
elif condicion3:
    # Código a ejecutar si la condicion1 y la condicion2 son falsas y la condicion3 es verdadera
else:
    # Código a ejecutar si todas las condiciones anteriores son falsas
```

# Sentencia elif {

Es importante tener en cuenta que se pueden utilizar varias **sentencias elif** entre la sentencia if y la sentencia else para evaluar múltiples condiciones.

Sin embargo, la **sentencia else** siempre debe ir al final y solo puede haber una sentencia else en una estructura **if-elif-else**.



```
if x > 25:
    print('x es mayor que 25')
elif x < 25:
    print('x es menor que 25')
else:
    print('x es igual a 25')
```

## Ejercicio 5 {

Desarrolla en Python el *juego de **Piedra, Papel o Tijera***.  
Deberás pedir por teclado la opción del **jugador 1**, luego la  
opción del **jugador 2**, y posteriormente indicar quien es el  
ganador, o si fue un empate.

}

# Solución Ejercicio 5 {

```
print("Juego Papel, Piedra o Tijera")

print("Jugador 1 es su turno")
jugador_1 = int(input("Seleccione piedra(0), papel(1), tijera(2): "))

print("Jugador 2 es su turno")
jugador_2 = int(input("Seleccione piedra(0), papel(1), tijera(2): "))

print("Resultado")
if jugador_1 == jugador_2:
    print("Empate") # Si los numeros son iguales, entonces es empate
else:
    # Evaluamos todos los casos posibles en los que el jugador q gana
    # Piedra le gana a tijera
    caso_1 = (jugador_1 == 0 and jugador_2 == 2)
    # Papel le gana a piedra
    caso_2 = (jugador_1 == 1 and jugador_2 == 0)
    # Tijera le gana a papel
    caso_3 = (jugador_1 == 2 and jugador_2 == 1)

    if caso_1 or caso_2 or caso_3:
        print("Gano el jugador 1.")
    else:
        print("Gano el jugador 2.")
```

## Ejercicio 6 {

Realice un programa **que pida dos números al usuario** y que permita realizar las siguientes **operaciones matemáticas**:

- Suma.
- Resta.
- Multiplicación.
- División.
- Potencia.
- Modulo.
- Valor absoluto.
- Mayor que.
- Menor que.

}

## Ejercicio 7 {

Te contrataron para realizar un programa que calcule el **premio de un juego**, el puntaje es recibido por un `input()` del usuario. Debe mostrar el resultado por la consola.

Los premios están basados en puntos:

Puntos	Premio
1-50	No hay premio
51-150	Bronce
151-180	Plata
181-200	Oro

}

# Valores verdaderos {

En Python, un **valor "truthy"** es **cualquier valor que se evalúa como verdadero en un contexto booleano.**

Esto significa que incluso si el valor no es estrictamente **True** (el valor booleano verdadero), se considerará verdadero si se evalúa en una expresión booleana.

}

# Valores verdaderos {

En Python, un **valor "truthy"** es **cualquier valor que se evalúa como verdadero en un contexto booleano.**

Esto significa que incluso si el valor no es estrictamente **True** (el valor booleano verdadero), se considerará verdadero si se evalúa en una expresión booleana.

}



# Valores verdaderos {

01

Cualquier **número que no sea cero** (1, 2.5, -3)

02

Cualquier **cadena de caracteres no vacía** ("hello", " ")

03

La lista, tupla, diccionario y otros tipos de **contenedores que no están vacíos** (['a', 'b'], (1, 2), {'a': 1, 'b': 2})

04

El valor **booleano True**

}

## 04 {

## [Estructuras Repetitivas]

- Ciclos
- Ciclo while

}

# Ciclos {

En programación, los ciclos (también llamados bucles) son estructuras de control que **permiten ejecutar repetidamente un bloque de código mientras se cumple una condición determinada.**

En Python, existen dos tipos de ciclos: el **ciclo "for"** y el **ciclo "while"**.

El número de **iteraciones** de una estructura repetitiva puede ser definido de una de dos maneras:

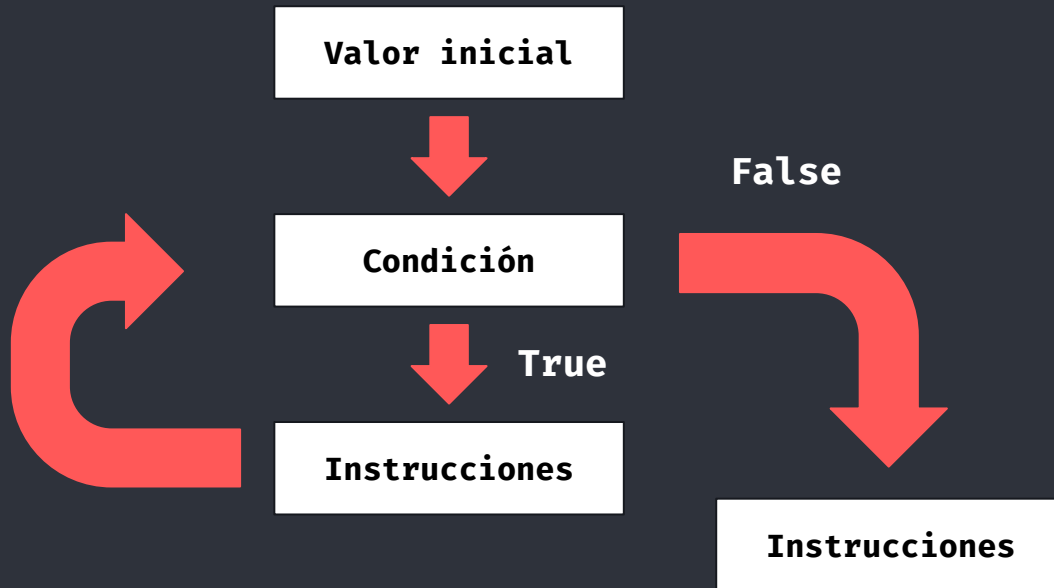
- Por condición
- Por iterador

}

# Ciclos {

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14

}



# Ciclo While {

En Python, **"while"** es una palabra clave que se utiliza para crear un ciclo o bucle que **se ejecuta mientras una determinada condición sea verdadera**.

La sintaxis básica del **ciclo "while"** en Python es la siguiente:



```
while condicion:  
    #Bloque de código
```

}

# Ciclo While {

El **ciclo "while"** comienza evaluando la condición **especificada**. Si la condición es verdadera, se ejecuta el bloque de código.

Después de que se ha ejecutado el bloque de código, **la condición se evalúa nuevamente**. Si la condición sigue siendo verdadera, se ejecuta el bloque de código nuevamente.

Este proceso se repite hasta que la condición se evalúa como **falsa**.

}

# Ciclo While {



```
iteracion = 0
```

```
while (iteracion < 10):
```

```
    x += 1
```

```
    print(f"Iteracion numero {iteracion}")
```

```
}
```

# Ciclo While {

Es perfectamente posible que un **ciclo while** nunca llegue a ejecutarse debido a que su condición no se cumpla en ningún momento.

Si no se tiene cuidado, **las instrucciones de un while pueden repetirse indefinidamente**. Es decir, es perfectamente posible que la condición asignada al bloque nunca deje de cumplirse.

}



## Ejercicio 8 {

Desarrolle un programa que dado un valor ingresado por el usuario, evalúe si es un **número entero**.

Se exige tanto el uso del método .isdigit(), como el uso del **ciclo while**.

}

## Ejercicio 8 {



```
numero = input("Introduce un número: ")

while not numero.isdigit():
    print("El valor introducido no es un número.")
    numero = input("Introduce un número: ")

print("El número introducido es:", numero)
```

}

## Ejercicio 9 {

1. Solicite al usuario su **nombre**, **cedula** y **edad**, y luego imprima por consola la frase:  
*"{nombre} es una persona de CI {cedula} y de edad {edad}"*.
2. Valide que el **nombre** sea una palabra que contenga **solo letras**. Debe utilizar **.isalpha()**.
3. Valide que la **cédula** sea una cadena que contenga **únicamente números**. Debe utilizar **.isnumeric()**.
4. Valide que la **edad** sea una **cadena numérica**, y que sea un **número entre 0 y 100**. Debe utilizar **.isnumeric()**.

# Ejercicio 9 {

```
cedula = ''
nombre = ''
edad = ''
nombre = input('Por favor, introduzca el nombre: ')
#validando input de nombre
while not nombre.isalpha() or len(nombre) == 0 : # Mientras nombre no sea una cadena alfabetica o este vacio (de longitud 0)
    print('Dato incorrecto.')
    nombre = input('Por favor, introduzca el nombre: ')

cedula = input('Por favor, introduzca la cedula: ')
#validando input de cedula
while not cedula.isnumeric() or len(cedula) == 0 : # Mientras cedula no sea una cadena de numeros o este vacio (de longitud 0)
    print('Dato incorrecto.')
    cedula = input('Por favor, introduzca la cedula: ')

edad = input('Por favor, introduzca la edad: ')
#validando input de edad
while not edad.isnumeric() or int(edad) < 0 or int(edad) > 100: # Mientras edad no sea una cadena numerica o no este dentro del rango
    print('Dato incorrecto.')
    edad = input('Por favor, Introduzca la edad: ')

print(f'{nombre} es una persona de ci {cedula} y de edad {edad}')
```

05 {

## [Funciones Built-in]

- Funciones built-in
- Métodos built-in

}

# Funciones built-in {

Las funciones son **bloques de código reutilizables que se ejecutan al ser llamadas**. Las funciones pueden retornar un valor como respuesta a esta llamada, o realizar alguna acción.

Para usar las funciones built-in de Python, debemos colocar el nombre de la misma y colocar paréntesis. Dentro del paréntesis se deben ingresar los argumentos separados por comas, en caso de ser necesarios.



nombre\_funcion()

# Funciones built-in {

Algunos ejemplos de funciones "built-in" en Python son:

- **print():** Imprime un mensaje en la consola.
- **len():** Devuelve la longitud de una secuencia (como una cadena, una lista o una tupla).
- **input():** Lee una entrada de datos introducida por el usuario.
- **str():** Convierte un objeto en una cadena de caracteres.
- **int():** Convierte una cadena de caracteres o un número en un entero.
- **float():** Convierte una cadena de caracteres o un número en un número de punto flotante.
- **type():** Devuelve el tipo de un objeto.
- **range():** Genera una secuencia de números enteros.

}

# Metodos built-in {

En Python, los **métodos "built-in"** (o integrados) son **métodos predefinidos que están disponibles en los objetos de ciertos tipos de datos** y que se pueden utilizar sin necesidad de importar ningún módulo adicional.

Estos métodos proporcionan una serie de funcionalidades comunes para trabajar con los objetos de datos y son específicos de cada tipo de objeto. Dentro del paréntesis se deben ingresar los argumentos separados por comas, en caso de ser necesarios.



```
variable.nombre_metodo()
```



# Metodos built-in de Strings {

Algunos de los **métodos "built-in"** para objetos de tipo cadena de caracteres (str) son:

- **.upper():** Devuelve una copia de la cadena con todos los caracteres en mayúsculas.
- **.lower():** Devuelve una copia de la cadena con todos los caracteres en minúsculas.
- **.strip():** Devuelve una copia de la cadena sin los espacios en blanco iniciales y finales.
- **.replace():** Devuelve una copia de la cadena con todas las apariciones de una subcadena dada reemplazadas por otra subcadena dada.
- **.find():** Devuelve la posición de la primera aparición de una subcadena dada en la cadena, o -1 si la subcadena no está presente.

}

# Metodos built-in de Strings {

- **.title():** Devuelve una versión del string con formato de título (La primera letra de cada palabra en mayúscula).
- **.isdigit():** Indica si un string contiene solo dígitos, y retorna un booleano con el resultado.
- **.isalpha():** Indica si el string contiene solo letras y retorna un booleano con el resultado.
- **.isalnum():** Indica si el string contiene letras y/o números y retorna un booleano con el resultado.
- **.replace():** Reemplaza un valor dentro del string, por un nuevo valor indicado.
- **.split(separador):** Separa el string de acuerdo a un valor indicado y retorna una lista con todos los substrings derivados.

}