

## ÍNDICE

1.	Introducción .....	2
1.1.	Objetivos .....	2
2.	Estructuras de datos.....	3
2.1.	Listas.....	3
A.	Estructura de Listas .....	3
B.	Funciones de Listas.....	4
2.2.	Tuplas .....	5
A.	Estructura de Tupla.....	5
B.	Funciones de Tupla .....	6
2.3.	Diccionarios .....	6
A.	Estructura de Diccionarios.....	7
B.	Funciones de los diccionarios:.....	8
2.4.	Conjuntos .....	8
A.	Estructura de Conjuntos.....	9
B.	Funciones de los conjuntos: .....	9
3.	Condicionales .....	11
3.1.	IF .....	11
3.2.	ELSE .....	11
3.3.	ELIF .....	12
4.	Ejecuciones Iterativas .....	14
4.1.	FOR .....	14
4.2.	WHILE .....	14
4.3.	Sentencias Adicionales .....	15
4.4.	Iteradores .....	18
ANEXO	.....	20

## **1. Introducción**

En esta etapa, nos centraremos en tres aspectos esenciales: las estructuras de datos, las ejecuciones condicionales y las ejecuciones iterativas. Estos conceptos son fundamentales para construir programas más complejos y eficientes, y te brindarán herramientas poderosas para resolver problemas de programación de manera efectiva.

Durante esta sesión, aprenderás a utilizar las diversas estructuras de datos que ofrece Python para organizar y almacenar información de manera eficiente. También exploraremos las sentencias condicionales, que te permitirán controlar el flujo de ejecución de tu programa en función de condiciones específicas. Además, te sumergirás en las ejecuciones iterativas, que te darán la capacidad de repetir bloques de código para realizar tareas repetitivas o procesar conjuntos de datos.

### **1.1. Objetivos**

- Comprender las diversas estructuras de datos disponibles en Python y saber cómo utilizarlas para almacenar y organizar objetos complejos.
- Familiarizarte con las funciones esenciales asociadas a cada estructura de datos, lo que te permitirá aprovechar al máximo su potencial y realizar operaciones eficientes.
- Dominar las sentencias condicionales para tomar decisiones en la ejecución de tus programas en función de diferentes situaciones y resultados.
- Aprender las diferentes técnicas de ejecuciones iterativas mediante bucles, permitiéndote repetir bloques de código para optimizar la eficiencia y abordar problemas complejos.
- Utilizar de forma efectiva los iteradores para recorrer y manipular secuencias de valores en Python.

## 2. Estructuras de datos

Una estructura de datos es una forma de organizar y almacenar datos de manera eficiente en la memoria de una computadora. En Python, contamos con diferentes tipos de estructuras de datos que nos permiten almacenar y manipular información de manera versátil.

### 2.1. Listas

Una lista es una colección ordenada y modificable de elementos. Puedes almacenar cualquier tipo de dato en una lista, como números, cadenas o incluso otras listas. Las listas se representan con corchetes ([]), y los elementos se separan por comas. Puedes acceder a los elementos de una lista utilizando índices y realizar operaciones como agregar elementos, eliminar elementos o modificar elementos existentes.

#### A. Estructura de Listas

- **Almacenar cualquier tipo de dato:** Puedes almacenar cualquier tipo de dato en una lista. Por ejemplo, puedes tener una lista de números, cadenas, booleanos u otros objetos.

```
# Ejemplo de una lista con diferentes tipos de datos
lista = [1, 'Hola', True, 3.14, ['otra', 'lista']]
```

- **Crear listas vacías:** Puedes crear una lista vacía e ir agregando elementos a medida que lo necesites.

```
# Ejemplo de lista vacía
lista_vacia = list()
# O también
lista_vacia = []
```

- **Acceder a posiciones:** Puedes acceder a los elementos de una lista utilizando índices. Los índices comienzan desde 0 para el primer elemento y pueden ser negativos para contar desde el final de la lista.

```
# Ejemplo de acceso a elementos de una lista
lista = ['manzana', 'banana', 'cereza']
```

```
print(lista[0])      # Imprime: manzana
print(lista[-1])     # Imprime: cereza
```

- **Rango de índices:** Puedes utilizar el operador de segmentación (:) para obtener un rango de elementos de una lista.

```
# Ejemplo de rango de índices en una lista
lista = ['a', 'b', 'c', 'd', 'e']
print(lista[1:3])    # Imprime: ['b', 'c']
print(lista[:2])     # Imprime: ['a', 'b']
print(lista[3:])     # Imprime: ['d', 'e']
```

## B. Funciones de Listas

- **len(lista):** Devuelve la longitud de la lista, es decir, el número de elementos en la lista.
- **lista.append(elemento):** Agrega un elemento al final de la lista.
- **lista.insert(posición, elemento):** Inserta un elemento en una posición específica de la lista.
- **lista.remove(elemento):** Elimina la primera aparición del elemento en la lista.
- **lista.pop([posición]):** Elimina y devuelve el elemento en la posición especificada, o el último elemento si no se proporciona ninguna posición.
- **lista.index(elemento):** Devuelve el índice de la primera aparición del elemento en la lista.
- **elemento in lista:** Verifica si un elemento está presente en la lista.
- **lista.sort():** Ordena los elementos de la lista de forma ascendente.
- **lista.reverse():** Invierte el orden de los elementos en la lista.

## Ejercicios

```
numeros = [5, 2, 8, 1, 10]

print(len(numeros))    # Imprime: 5
numeros.append(4)
print(numeros)         # Imprime: [5, 2, 8, 1, 10, 4]
numeros.insert(2, 7)
print(numeros)         # Imprime: [5, 2, 7, 8, 1, 10, 4]
```

```
numeros.remove(8)
print(numeros)          # Imprime: [5, 2, 7, 1, 10, 4]
elemento = numeros.pop(3)
print(elemento)         # Imprime: 1
print(numeros)          # Imprime: [5, 2, 7, 10, 4]
print(2 in numeros)     # Imprime: True
numeros.sort()
print(numeros)          # Imprime: [2, 4, 5, 7, 10]
numeros.reverse()
print(numeros)          # Imprime: [10, 7, 5, 4, 2]
```

## 2.2. Tuplas

Una tupla es similar a una lista, pero es inmutable, lo que significa que no se pueden modificar una vez creadas. Las tuplas se representan con paréntesis (), y los elementos se separan por comas. Aunque no se pueden modificar, las tuplas son útiles cuando necesitas almacenar datos que no deben cambiar, como coordenadas geográficas.

### A. Estructura de Tupla

- **Almacenar datos inmutables:** Las tuplas se utilizan para almacenar datos que no deben modificarse después de su creación, como coordenadas o datos constantes.

```
# Ejemplo de una tupla de coordenadas
coordenadas = (10, 20)
```

- **Crear Tupla vacía:** Para crear una tupla vacía, puedes utilizar la función tuple() sin pasar ningún argumento.

```
# Ejemplo de tupla vacía
tupla_vacia = tuple()

# O también
tupla_vacia = ()
```

- **Acceder a elementos:** Puedes acceder a los elementos de una tupla utilizando índices de la misma manera que en las listas.

```
# Ejemplo de acceso a elementos de una tupla
coordenadas = (10, 20)
print(coordenadas[0])    # Imprime: 10
print(coordenadas[1])    # Imprime: 20
```

- **Desempaquetado de tuplas:** Puedes asignar los elementos de una tupla a variables individuales en una sola operación.

```
# Ejemplo de desempaquetado de tuplas
punto = (3, 4)
x, y = punto
print(x)    # Imprime: 3
print(y)    # Imprime: 4
```

## B. Funciones de Tupla

- **len(tupla):** Devuelve la longitud de la tupla, es decir, el número de elementos en la tupla.
- **tupla.index(elemento):** Devuelve el índice de la primera aparición del elemento en la tupla.
- **elemento in tupla:** Verifica si un elemento está presente en la tupla.

## Ejercicios

```
punto = (3, 7, 2)

print(len(punto))          # Imprime: 3
print(punto.index(7))      # Imprime: 1
print(2 in punto)          # Imprime: True
```

## 2.3. Diccionarios

Un diccionario es una estructura de datos que almacena pares clave-valor. Cada elemento en un diccionario consiste en una clave única y su valor correspondiente. Los diccionarios se representan con llaves (`{}`), y los pares clave-valor se separan por comas. Puedes acceder a los valores de un diccionario utilizando sus claves, y también puedes agregar, modificar o eliminar pares clave-valor.

## A. Estructura de Diccionarios

- **Almacenar datos con claves únicas:** Cada elemento en un diccionario tiene una clave única asociada a su valor correspondiente.

```
# Ejemplo de un diccionario de frutas y sus cantidades
frutas = {'manzanas': 10, 'naranjas': 5, 'plátanos': 7}
```

- **Crear Diccionario vacío:** Para crear un diccionario vacío, puedes utilizar la función dict() o simplemente las llaves {} sin ningún par clave-valor dentro.

```
# Ejemplo de diccionario vacío
diccionario_vacio = dict()
# O también
diccionario_vacio = {}
```

- **Acceder a valores mediante claves:** Puedes acceder a los valores de un diccionario utilizando sus claves.

```
# Ejemplo de acceso a valores de un diccionario
frutas = {'manzanas': 10, 'naranjas': 5, 'plátanos': 7}
print(frutas['manzanas']) # Imprime: 10
```

- **Agregar y modificar elementos:** Puedes agregar nuevos pares clave-valor o modificar valores existentes en un diccionario.

```
# Ejemplo de agregar y modificar elementos en un diccionario
frutas = {'manzanas': 10, 'naranjas': 5}
frutas['plátanos'] = 7      # Agrega un nuevo par clave-valor
frutas['manzanas'] = 15     # Modifica el valor existente
print(frutas)              # Imprime: {'manzanas': 15, 'naranjas':
5, 'plátanos': 7}
```

## B. Funciones de los diccionarios:

- **len(diccionario):** Devuelve la cantidad de pares clave-valor en el diccionario.
- **diccionario[key]:** Accede al valor asociado con una clave específica.
- **diccionario[key] = valor:** Asigna un valor a una clave en el diccionario.
- **diccionario.keys():** Devuelve una lista con todas las claves del diccionario.
- **diccionario.values():** Devuelve una lista con todos los valores del diccionario.
- **diccionario.items():** Devuelve una lista de tuplas con los pares clave-valor del diccionario.

## Ejercicios

```
frutas = {'manzanas': 10, 'naranjas': 5, 'plátanos': 7}

print(len(frutas))          # Imprime: 3
print(frutas['manzanas'])    # Imprime: 10

frutas['peras'] = 3
print(frutas)               # Imprime: {'manzanas': 10, 'naranjas': 5,
                             'plátanos': 7, 'peras': 3}

print(frutas.keys())        # Imprime: dict_keys(['manzanas',
                             'naranjas', 'plátanos', 'peras'])
print(frutas.values())      # Imprime: dict_values([10, 5, 7, 3])
print(frutas.items())       # Imprime: dict_items([('manzanas', 10),
                             ('naranjas', 5), ('plátanos', 7), ('peras', 3)])
```

## 2.4. Conjuntos

Un conjunto es una colección desordenada de elementos únicos. Los conjuntos son útiles cuando necesitas almacenar elementos sin duplicados y no te importa el orden. Los conjuntos se representan con llaves {}, pero a diferencia de los diccionarios, no tienen pares clave-valor. Puedes realizar operaciones de conjunto, como unión, intersección o diferencia, y también agregar o eliminar elementos de un conjunto.



## A. Estructura de Conjuntos

- **Almacenar elementos únicos:** Los conjuntos no contienen elementos duplicados, por lo que son útiles para almacenar una colección de valores únicos.

```
# Ejemplo de un conjunto de colores
colores = {'rojo', 'verde', 'azul'}
```

- **Crear Conjunto vacío:** Para crear un conjunto vacío, puedes utilizar la función `set()` sin pasar ningún argumento.

```
# Ejemplo de conjunto vacío
conjunto_vacio = set()
```

- **Operaciones de conjunto:** Puedes realizar operaciones de conjunto, como unión, intersección y diferencia, utilizando los métodos y operadores proporcionados por los conjuntos.

```
# Ejemplo de operaciones de conjunto
conjunto1 = {1, 2, 3}
conjunto2 = {2, 3, 4}
union = conjunto1 | conjunto2           # Unión de conjuntos
interseccion = conjunto1 & conjunto2    # Intersección de
conjuntos
diferencia = conjunto1 - conjunto2      # Diferencia de conjuntos
print(union)                           # Imprime: {1, 2, 3, 4}
print(interseccion)                    # Imprime: {2, 3}
print(diferencia)                       # Imprime: {1}
```

## B. Funciones de los conjuntos:

- **`len(conjunto)`:** Devuelve la cantidad de elementos en el conjunto.
- **`conjunto.add(elemento)`:** Agrega un elemento al conjunto.
- **`conjunto.remove(elemento)`:** Elimina un elemento del conjunto. Genera un error si el elemento no está presente.

- **conjunto.discard(elemento):** Elimina un elemento del conjunto si está presente. No genera un error si el elemento no existe.
- **elemento in conjunto:** Verifica si un elemento está presente en el conjunto.
- **conjunto.union(otro\_conjunto):** Devuelve un nuevo conjunto que es la unión de dos conjuntos.
- **conjunto.intersection(otro\_conjunto):** Devuelve un nuevo conjunto que es la intersección de dos conjuntos.
- **conjunto.difference(otro\_conjunto):** Devuelve un nuevo conjunto que es la diferencia entre dos conjuntos.

## Ejercicios

```
vocales = {'a', 'e', 'i'}

print(len(vocales))           # Imprime: 3
vocales.add('o')
print(vocales)                # Imprime: {'a', 'i', 'o', 'e'}
vocales.remove('i')
print(vocales)                # Imprime: {'a', 'o', 'e'}
print('u' in vocales)         # Imprime: False

numeros1 = {1, 2, 3}
numeros2 = {3, 4, 5}
union = numeros1.union(numeros2)
print(union)                  # Imprime: {1, 2, 3, 4, 5}
interseccion = numeros1.intersection(numeros2)
print(interseccion)           # Imprime: {3}
diferencia = numeros1.difference(numeros2)
print(diferencia)             # Imprime: {1, 2}
```

### 3. Condicionales

Las ejecuciones condicionales nos permiten tomar decisiones en el flujo de un programa en función de ciertas condiciones. En Python, utilizamos las declaraciones "if", "elif" y "else" para implementar ejecuciones condicionales. Estas declaraciones evalúan una condición y ejecutan un bloque de código si se cumple esa condición.

#### 3.1. IF

La declaración if nos permite ejecutar un bloque de código si una condición especificada es verdadera. La sintaxis básica es la siguiente:

```
if condición:  
    # Bloque de código a ejecutar si la condición es verdadera
```

La condición puede ser cualquier expresión que se evalúe como verdadera o falsa. Si la condición es verdadera, el bloque de código indentado que sigue al if se ejecutará. Si la condición es falsa, el bloque de código se omitirá.

```
"""  
En este ejemplo, si la variable edad es mayor o igual a 18  
, se imprimirá "Eres mayor de edad".  
"""  
edad = 20  
  
if edad >= 18:  
    print("Eres mayor de edad")
```

#### 3.2. ELSE

La declaración else se utiliza como una cláusula final en una estructura condicional y se ejecuta si ninguna de las condiciones anteriores es verdadera. La sintaxis básica es la siguiente:

```
if condición1:  
    # Bloque de código a ejecutar si la condición1 es verdadera  
else:  
    # Bloque de código a ejecutar si la condición1 es falsa
```

El bloque de código dentro del `else` se ejecutará solo si todas las condiciones anteriores resultan falsas.

```
"""
En este ejemplo, si el valor de hora es menor que 12,
se imprimirá "Buenos días". De lo contrario, se imprimirá "Buenas tardes".
"""
hora = 15

if hora < 12:
    print("Buenos días")
else:
    print("Buenas tardes")
```

### 3.3. ELIF

La declaración `elif` nos permite evaluar múltiples condiciones alternativas después de un `if` inicial. La sintaxis básica es la siguiente:

```
if condición1:
    # Bloque de código a ejecutar si la condición1 es verdadera
elif condición2:
    # Bloque de código a ejecutar si la condición1 es falsa y la condición2
    es verdadera
```

Puedes tener tantas cláusulas `elif` como desees. Se evaluarán en orden secuencial hasta que se encuentre una condición verdadera. Si se encuentra una condición verdadera, el bloque de código correspondiente se ejecutará y el resto de las cláusulas `elif` y el `else` se omitirán.

```
"""
En este ejemplo, se evalúa la variable puntuación y
se imprime un mensaje dependiendo del rango de puntuación
en el que se encuentre.
"""
puntuación = 85

if puntuación >= 90:
    print("Excelente")
```

```
elif puntuación >= 80:  
    print("Buen trabajo")  
elif puntuación >= 70:  
    print("Aprobado")  
else:  
    print("Reprobado")
```

## 4. Ejecuciones Iterativas

Las ejecuciones iterativas nos permiten repetir bloques de código múltiples veces. En Python, tenemos dos tipos de bucles para implementar ejecuciones iterativas: el bucle "for" y el bucle "while".

### 4.1. FOR

El bucle "for" se utiliza cuando se conoce el número exacto de repeticiones o cuando deseas iterar sobre una secuencia de elementos, como una lista o una cadena de caracteres. La sintaxis básica es la siguiente:

```
for elemento in secuencia:  
    # Bloque de código a ejecutar para cada elemento de la secuencia
```

El bucle "for" itera sobre los elementos de la secuencia uno por uno, asignando cada elemento a la variable "elemento". Luego, se ejecuta el bloque de código indentado que sigue al bucle "for" una vez para cada elemento de la secuencia.

```
"""  
En este ejemplo, el bucle "for" itera sobre la lista de frutas.  
En cada iteración, la variable "fruta" toma el valor de cada elemento de la  
lista,  
y se ejecuta el bloque de código que imprime cada fruta en una línea  
separada.  
"""  
frutas = ["manzana", "plátano", "naranja"]  
  
for fruta in frutas:  
    print(fruta)
```

### 4.2. WHILE

El bucle "while" se utiliza cuando la repetición se basa en una condición que puede cambiar durante la ejecución. El bucle se ejecuta mientras la condición especificada sea verdadera. La sintaxis básica es la siguiente:

```
while condición:  
    # Bloque de código a ejecutar mientras la condición sea verdadera
```

Antes de cada iteración, se evalúa la condición. Si es verdadera, se ejecuta el bloque de código indentado que sigue al bucle "while". Después de cada iteración, la condición se vuelve a evaluar. Si la condición es falsa, el bucle se detiene y la ejecución continúa con la siguiente línea de código después del bucle.

```
"""
En este ejemplo, el bucle "while" se ejecutará mientras el valor de
"contador" sea menor que 5.
En cada iteración, se imprime el valor actual del contador y se incrementa
en 1.
El bucle se detiene cuando el contador alcanza el valor de 5.
"""
contador = 0

while contador < 5:
    print(contador)
    contador += 1
```

#### 4.3. Sentencias Adicionales

- **Break:** Es importante tener cuidado al utilizar bucles "while" para evitar bucles infinitos. Asegúrate de tener una condición de salida apropiada dentro del bucle para que eventualmente se vuelva falsa y el bucle se detenga.

```
# Manejo de break para evitar bucles infinitos
numero = 1

while numero <= 10:
    print(numero)
    numero += 1

    if numero > 15:
        break
```

En este ejemplo, el bucle "while" imprimirá los números del 1 al 10. Sin embargo, para evitar que se convierta en un bucle infinito, hemos incluido una condición de salida utilizando la instrucción "break". La condición `if numero > 15:` verifica si el valor de `numero` ha superado 15. Si eso ocurre, se ejecuta la instrucción "break" y el bucle se detiene,

permitiendo que la ejecución continúe con las líneas de código después del bucle.

La instrucción `break` se utiliza en un bucle `"for"` para detener la ejecución del bucle de forma prematura, sin recorrer todos los elementos de la secuencia.

```
frutas = ["manzana", "plátano", "naranja"]

for fruta in frutas:
    if fruta == "plátano":
        break
    print(fruta)
```

En este ejemplo, cuando el bucle `"for"` encuentra la fruta `"plátano"`, se ejecuta la instrucción `break`, lo que hace que el bucle se detenga inmediatamente y no se procesen los elementos restantes de la lista.

- **Continue:** se utiliza en Python para omitir el resto del bloque de código en una iteración específica de un bucle y pasar a la siguiente iteración. En otras palabras, cuando se encuentra la instrucción `continue` dentro de un bucle, el flujo del programa salta inmediatamente a la siguiente iteración sin ejecutar el resto del código dentro del bloque de esa iteración.

```
numero = 0

while numero < 10:
    numero += 1

    if numero % 2 == 0:
        continue

    print(numero)
```

En este ejemplo, el bucle `"while"` imprimirá los números impares del 1 al 10. Dentro del bucle, se evalúa si el número es par usando la condición `numero % 2 == 0`. Si el número es par, se encuentra la instrucción `continue`, lo que significa que se omitirá la impresión del número y se pasará directamente a la siguiente iteración del bucle.



```
frutas = ["manzana", "plátano", "naranja"]

for fruta in frutas:
    if fruta == "plátano":
        continue
    print(fruta)
```

En este ejemplo, cuando el bucle "for" encuentra la fruta "plátano", se ejecuta la instrucción continue, lo que salta a la siguiente iteración sin ejecutar el código de impresión para esa fruta en particular. El bucle continúa con las siguientes iteraciones.

- **Pass:** La instrucción pass no hace nada. Se utiliza como marcador de posición cuando no se requiere ninguna acción en ese punto del código.

Puedes usarlo dentro de un bucle "while" para evitar errores de sintaxis cuando aún no has implementado la lógica para esa parte del código.

```
while condición:
    # Código aún no implementado
    pass
```

La instrucción pass se utiliza en un bucle "for" cuando no se requiere ninguna acción en ese punto del código. Puedes utilizarla como marcador de posición para evitar errores de sintaxis.

```
frutas = ["manzana", "plátano", "naranja"]

for fruta in frutas:
    if fruta == "plátano":
        pass
    else:
        print(fruta)
```

En este ejemplo, cuando el bucle "for" encuentra la fruta "plátano", se utiliza la instrucción pass para no realizar ninguna acción adicional. El

bucle continúa con las siguientes iteraciones y, en el caso de las frutas diferentes a "plátano", se ejecuta el código de impresión.

- **Else:** Puedes usar la instrucción else en conjunción con el bucle "while". El bloque de código dentro del else se ejecutará cuando la condición del bucle se vuelva falsa (es decir, cuando el bucle termine normalmente sin un break).

```
contador = 0

while contador < 5:
    print(contador)
    contador += 1
else:
    print("El bucle ha finalizado")
```

El bloque else se puede utilizar con un bucle "for" de manera similar a un bucle "while". El código dentro del bloque else se ejecutará al finalizar el bucle, después de que se hayan procesado todos los elementos de la secuencia, siempre y cuando no se haya ejecutado la instrucción break para salir del bucle de forma prematura.

```
frutas = ["manzana", "plátano", "naranja"]

for fruta in frutas:
    print(fruta)
else:
    print("Se han procesado todas las frutas")
```

#### 4.4. Iteradores

Un iterador es un objeto que implementa los métodos `__iter__()` y `__next__()` en Python. El método `__iter__()` devuelve el propio objeto iterador, mientras que el método `__next__()` devuelve el siguiente elemento de la secuencia o lanza una excepción `StopIteration` cuando no hay más elementos para recorrer.

El protocolo del iterador es una forma de diseño en Python que define cómo se implementa y utiliza un iterador. El protocolo requiere que un objeto

iterador tenga los métodos `__iter__()` y `__next__()`, y permite a los bucles `for` y otras construcciones de control trabajar con iteradores de manera uniforme.

```
frutas = ["manzana", "plátano", "naranja"]

iterador_frutas = iter(frutas) # Crear un iterador a partir de la lista

for fruta in iterador_frutas:
    print(fruta)
```

En este ejemplo, utilizamos la función `iter()` para crear un iterador a partir de la lista `frutas`. Luego, el bucle `for` itera sobre el iterador y en cada iteración obtiene el siguiente elemento de la lista y lo asigna a la variable `fruta`. El resultado es la impresión de cada elemento de la lista.

Python también proporciona la función `iter()` que permite crear un iterador a partir de un objeto que es iterable. Además, es posible trabajar con iteradores que generan una secuencia infinita de elementos, lo que es útil en algunos escenarios específicos.

```
import itertools

iterador_infinito = itertools.count() # Iterador que genera una secuencia
infinita de números

for num in iterador_infinito:
    print(num)
    if num >= 10:
        break
```

En este ejemplo, utilizamos el módulo `itertools` y la función `count()` para crear un iterador infinito que genera una secuencia infinita de números. El bucle `for` itera sobre el iterador y muestra los números hasta que se alcanza un límite (en este caso, 10) utilizando la instrucción `break`.

## **ANEXO**

## random

En Python es una librería estándar que proporciona funciones para generar números aleatorios y realizar operaciones relacionadas con aleatoriedad. Es parte de la biblioteca estándar de Python, por lo que no es necesario instalarla por separado, ya que viene incluida con cualquier instalación de Python.

La librería "random" ofrece diversas funciones que permiten generar números pseudoaleatorios y trabajar con ellos. Algunas de las principales funciones que se pueden utilizar son:

- **random.random():** Genera un número decimal (flotante) pseudoaleatorio en el rango [0, 1). Es decir, puede generar cualquier valor entre 0 (incluido) y 1 (excluido).
- **random.randint(a, b):** Genera un número entero pseudoaleatorio en el rango [a, b], ambos extremos incluidos.
- **random.choice(lista):** Elige un elemento aleatorio de una lista.
- **random.shuffle(lista):** Mezcla los elementos de una lista de forma aleatoria, modificando la lista original.
- **random.sample(population, k):** Devuelve una muestra aleatoria de tamaño k de la población dada sin repetir elementos.
- **random.seed(x):** Establece la semilla del generador de números aleatorios para obtener resultados reproducibles. Si se utiliza la misma semilla, se generarán los mismos números aleatorios.

Es importante mencionar que los números generados con la librería "random" son pseudoaleatorios, lo que significa que aunque aparentan ser aleatorios, en realidad están determinados por una semilla inicial y un algoritmo interno. Si se necesita aleatoriedad más segura, por ejemplo, para fines

criptográficos, se debe utilizar la librería "secrets" o módulos especializados que proporcionen aleatoriedad criptográficamente segura.

## **math**

en Python es otra librería estándar que proporciona funciones matemáticas y constantes matemáticas. Al igual que la librería "random", la librería "math" también viene incluida en cualquier instalación de Python y no requiere instalación adicional.

La librería "math" ofrece una amplia variedad de funciones matemáticas que permiten realizar operaciones matemáticas más avanzadas y precisas. Algunas de las principales funciones que se pueden utilizar en la librería "math" son:

- **Funciones trigonométricas:** `math.sin()`, `math.cos()`, `math.tan()`, `math.asin()`, `math.acos()`, `math.atan()`, entre otras.
- **Funciones exponenciales y logarítmicas:** `math.exp()`, `math.log()`, `math.log10()`, `math.sqrt()`, etc.
- **Funciones de redondeo:** `math.ceil()`, `math.floor()`, `math.trunc()`, etc.
- **Funciones de potencia:** `math.pow()`, `math.sqrt()`, etc.
- **Constantes matemáticas:** `math.pi` ( $\pi$ ), `math.e` (euler), `math.tau` ( $\tau$ ), etc.

La librería "math" es útil cuando se requieren cálculos matemáticos más avanzados, como funciones trigonométricas, logaritmos, exponenciales y más. A diferencia de la librería "random", la librería "math" no se utiliza para generar números aleatorios, sino para realizar operaciones matemáticas específicas

### Base Types

integer, float, boolean, string

```
int 783 0 -192
float 9.23 0.0 -1.7e-6
bool True False
str "One\nTwo" 'I\'m'
```

↑  
immutable,  
ordered sequence of chars

new line  
multiline  
escaped  
tab char

### Container Types

- ordered sequence, fast index access, repeatable values
- no *a priori* order, unique key, fast key access ; keys = base types or tuples

```
list [1,5,9] ["x",11,8.9] ["word"] []
tuple (1,5,9) 11,"y",7.4 ("word",) ()
dict {"key":"value"} {}
set {"key1","key2"} {1,9,3,0} set()
```

↑  
immutable  
expression with just comas  
key/value associations

### Identifiers

for variables, functions, modules, classes... names

a..zA..Z followed by a..zA..Z\_0..9

- diacritics allowed but should be avoided
- language keywords forbidden
- lower/UPPER case discrimination

```
ⓐ a toto x7 y_max BigOne
ⓑ 8y and
```

### Variables assignment

```
x = 1.2+8+sin(0)
y,z,r = 9.2,-7.6,"bad"
```

↑  
value or computed expression  
variable name (identifier)

variables  
names

container with several values (here a tuple)

increment  
decrement → x-=2

x=None « undefined » constant value

### Conversions

type(expression)

```
int("15") can specify integer number base in 2nd parameter
int(15.56) truncate decimal part (round(15.56) for rounded integer)
float("-11.24e8")
str(78.3) and for litteral representation → repr("Text")
see other side for string formatting allowing finer control
```

bool → use comparators (with ==, !=, <, >, ...), logical boolean result

list("abc") → use each element from sequence → ['a','b','c']

dict([(3,"three"),(1,"one")]) → use each element from sequence → {1:'one',3:'three'}

set(["one","two"]) → use each element from sequence → {'one','two'}

":".join(["toto","12","pswd"]) → 'toto:12:pswd'  
joining string sequence of strings

"words with spaces".split() → ['words','with','spaces']

"1,4,8,2".split(",") → ['1','4','8','2']  
splitting string

### Sequences indexing

for lists, tuples, strings, ...

```
len(lst) → 6
```

individual access to items via [index]

```
lst[1] → 67
lst[0] → 11 first one
lst[-2] → 42
lst[-1] → 1968 last one
```

access to sub-sequences via [start slice : end slice : step]

```
lst[1:3] → [67,"abc"]
lst[-3:-1] → [3.14,42]
lst[:2] → [11,"abc",42]
lst[3:] → [11,67,"abc"]
lst[4:] → [42,1968]
```

Missing slice indication → from start / up to end.

On mutable sequences, usable to remove `del lst[3:5]` and to modify with assignment `lst[1:4]=['hop',9]`

### Boolean Logic

Comparators: < > <= >= == !=

a and b logical and

a or b logical or

not a logical not

True true constant value

False false constant value

### Statements Blocks

```
parent statement:
statements block 1...
:
parent statement:
statements block 2...
:
next statement after block 1
```

### Conditional Statement

statements block executed only if a condition is true

```
if logical expression:
statements block
```

can go with several elif, elif... and only one final else, example :

```
if x==42:
# block if logical expression x==42 is true
print("real truth")
elif x>0:
# else block if logical expression x>0 is true
print("be positive")
elif bFinished:
# else block if boolean variable bFinished is true
print("how, finished")
else:
# else block for other cases
print("when it's not")
```

### Maths

floating point numbers... approximated values! angles in radians

Operators: + - \* / // % \*\*

x ÷ integer ÷ remainder

```
(1+5.3)*2 → 12.6
abs(-3.2) → 3.2
round(3.57,1) → 3.6
```

```
from math import sin,pi...
sin(pi/4) → 0.707...
cos(2*pi/3) → -0.4999...
acos(0.5) → 1.0471...
sqrt(81) → 9.0
log(e**2) → 2.0 etc. (cf doc)
```



statements block executed as long as condition is true

### Conditional loop statement

**while** logical expression:  
→ statements block

**s = 0**  
**i = 1** } initializations **before** the loop

condition with at least one variable value (here **i**)

**while i <= 100:**

# statement executed as long as  $i \leq 100$

**s = s + i\*\*2**  
**i = i + 1** } make condition variable change

**print("sum:", s)** } computed result after the loop

be careful of infinite loops!

$$s = \sum_{i=1}^{i=100} i^2$$

statements block executed for each item of a container or iterator

### Iterative loop statement

**for** variable in sequence:  
→ statements block

Go over sequence's values

**s = "Some text"** } initializations **before** the loop

**cnt = 0** } loop variable, value managed by **for** statement

**for c in s:**

**if c == "e":**

**cnt = cnt + 1**

**print("found", cnt, "e")**

Count number of **e** in the string

loop on dict/set = loop on sequence of keys

use slices to go over a subset of the sequence

Go over sequence's index

□ modify item at index

□ access items around index (before/after)

**lst = [11, 18, 9, 12, 23, 4, 17]**

**lost = []**

**for idx in range(len(lst)):**

**val = lst[idx]**

**if val > 15:**

**lost.append(val)**

**lst[idx] = 15**

**print("modif:", lst, "-lost:", lost)**

Limit values greater than 15, memorization of lost values.

Go simultaneously over sequence's index and values:

**for idx, val in enumerate(lst):**

### Display / Input

**print("v=", 3, "cm :", x, " ", y+4)**

items to display: literal values, variables, expressions

print options:

□ **sep=" "** (items separator, default space)

□ **end="\n"** (end of print, default new line)

□ **file=f** (print to file, default standard output)

**s = input("Instructions: ")**

input always returns a string, convert it to required type (cf boxed Conversions on on ther side).

### Operations on containers

**len(c)** → items count  
**min(c)** **max(c)** **sum(c)**  
**sorted(c)** → sorted copy  
**val in c** → boolean, membership operator **in** (absence **not in**)  
**enumerate(c)** → iterator on (index, value)  
Special for **sequence containers** (lists, tuples, strings):  
**reversed(c)** → reverse iterator  
**c.index(val)** → position  
**c\*5** → duplicate  
**c+c2** → concatenate  
**c.count(val)** → events count

### Operations on lists

**lst.append(item)** add item at end  
**lst.extend(seq)** add sequence of items at end  
**lst.insert(idx, val)** insert item at index  
**lst.remove(val)** remove first item with value  
**lst.pop(idx)** remove item at index and return its value  
**lst.sort()** **lst.reverse()** sort / reverse list in place

### Operations on dictionaries

**d[key]=value**  
**d[key]→value**  
**d.clear()**  
**del d[clé]**  
**d.update(d2)** { update/add  
**d.keys()** { associations  
**d.values()** { views on keys, values  
**d.items()** { associations  
**d.pop(clé)**

### Operations on sets

Operators:  
**|** → union (vertical bar char)  
**&** → intersection  
**-** **^** → difference/symmetric diff  
**< <= > >=** → inclusion relations  
**s.update(s2)**  
**s.add(key)** **s.remove(key)**  
**s.discard(key)**

### Generator of int sequences

frequently used in **for** iterative loops  
**range([start, stop [, step]])**  
**range(5)** → 0 1 2 3 4  
**range(3, 8)** → 3 4 5 6 7  
**range(2, 12, 3)** → 2 5 8 11  
**range** returns a « generator », converts it to list to see the values, example:  
**print(list(range(4)))**

### Function definition

function name (identifier)  
named parameters  
**def fctname(p\_x, p\_y, p\_z):**  
"""documentation"""  
# statements block, res computation, etc.  
**return res** result value of the call.  
if no computed result to return: **return None**  
parameters and all of this bloc only exist in the block and during the function call ("black box")

### Function call

**r = fctname(3, i+2, 2\*i)**  
one argument per parameter  
retrieve returned result (if necessary)

### Files

storing data on disk, and reading it back  
**f = open("fil.txt", "w", encoding="utf8")**  
file variable for operations  
name of file on disk (+path...)  
opening mode  
□ 'r' read  
□ 'w' write  
□ 'a' append...  
encoding of chars for text files: utf8, ascii, latin1 ...  
writing  
**f.write("hello")**  
text file → read / write only strings, convert from/to required type.  
**f.close()** don't forget to close file after use  
Pythonic automatic close: **with open(...) as f:**  
very common: iterative loop reading lines of a text file  
**for line in f:**  
→ # line processing block

### Strings formatting

formatting directives  
values to format  
**"model {} {} {}".format(x, y, r) → str**  
**"{selection:formatting!conversion}"**  
□ Selection:  
**2**  
**x**  
**0.nom**  
**4[key]**  
**0[2]**  
Examples  
**"{:+2.3f}".format(45.7273)**  
→ **+'+45.727'**  
**"{1:>10s}".format(8, "toto")**  
→ **'toto'**  
**"{!r}".format("I'm")**  
→ **'"I\'m"'**  
□ Formatting:  
**fillchar alignment sign minwidth.precision-maxwidth type**  
**<>^+= -+space** 0 at start for filling with 0  
integer: **b** binary, **c** char, **d** decimal (default), **o** octal, **x** or **X** hexa...  
float: **e** or **E** exponential, **f** or **F** fixed point, **g** or **G** appropriate (default),  
% percent  
string: **s** ...  
□ Conversion: **s** (readable text) or **r** (literal representation)