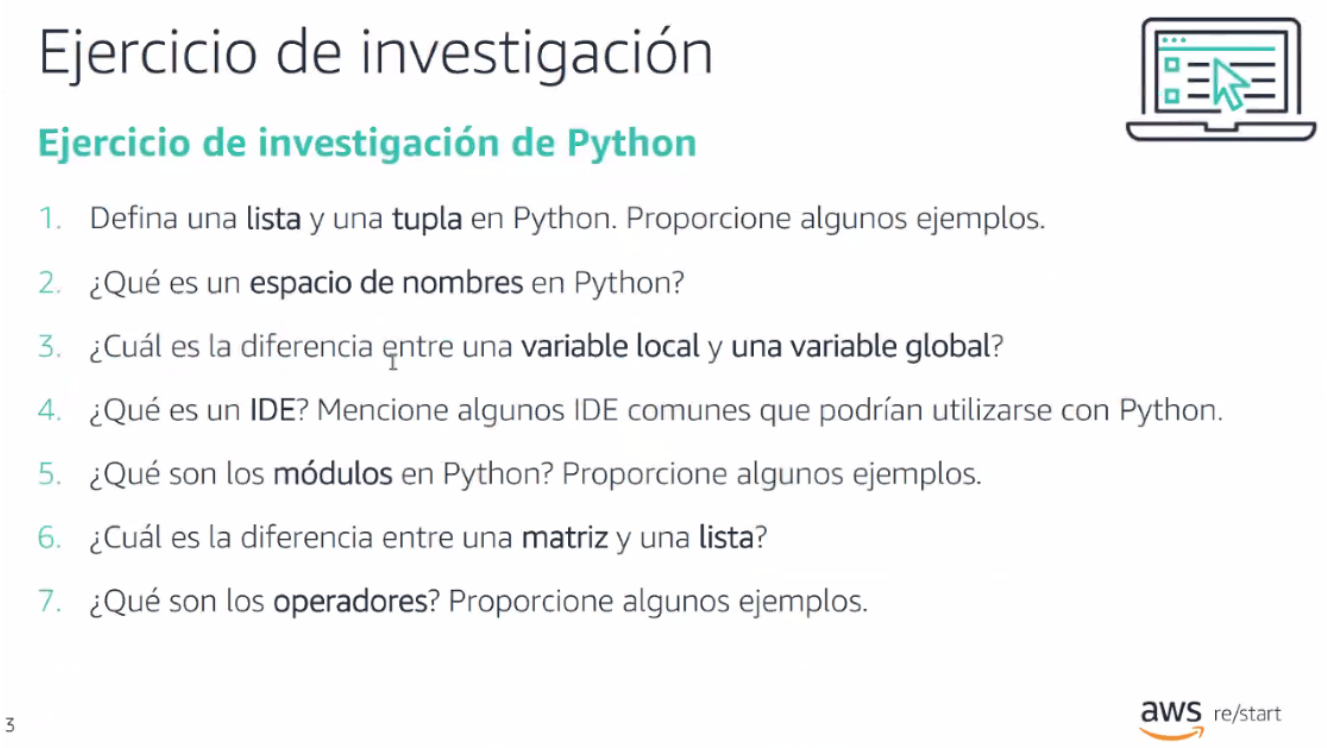
Ejercicio de investigación 19/5

Consigna



Ejercicio de investigación

Ejercicio de investigación de Python

1. Defina una **lista** y una **tupla** en Python. Proporcione algunos ejemplos.
2. ¿Qué es un **espacio de nombres** en Python?
3. ¿Cuál es la diferencia entre una **variable local** y **una variable global**?
4. ¿Qué es un **IDE**? Mencione algunos IDE comunes que podrían utilizarse con Python.
5. ¿Qué son los **módulos** en Python? Proporcione algunos ejemplos.
6. ¿Cuál es la diferencia entre una **matriz** y una **lista**?
7. ¿Qué son los **operadores**? Proporcione algunos ejemplos.

3

aws re/start

Resolución:

1) Listas y tuplas en Python

Listas

Una lista es una estructura de datos que se utiliza para almacenar una colección ordenada de elementos. Se puede pensar en una lista como una secuencia **mutable** de elementos, lo que significa que puedes agregar, eliminar y modificar elementos en la lista después de haberla creado. En una lista, los elementos pueden ser de diferentes tipos de datos, como enteros, cadenas, booleanos, objetos, etc. Además, los elementos de una lista están indexados, lo que permite acceder a ellos mediante un índice numérico.

```
#! Listas

bandas_rock = ["Led Zeppelin", "The Rolling Stones", "Queen", "AC/DC", "Pink Floyd"]
# Acceso a elementos de la lista
print(bandas_rock[0])  # Salida: Led Zeppelin
print(bandas_rock[2])  # Salida: Queen

# Modificación de elementos de la lista
bandas_rock[1] = "Guns N' Roses"
print(bandas_rock)  # Salida: ["Led Zeppelin", "Guns N' Roses", "Queen", "AC/DC", "Pink Floyd"]

# Agregar elementos a la lista
bandas_rock.append("The Beatles")
print(bandas_rock)  # Salida: ["Led Zeppelin", "Guns N' Roses", "Queen", "AC/DC", "Pink Floyd", "The Beatles"]

# Eliminar elementos de la lista
bandas_rock.remove("AC/DC")
print(bandas_rock)  # Salida: ["Led Zeppelin", "Guns N' Roses", "Queen", "Pink Floyd", "The Beatles"]
```

Tuplas

Una tupla también es una estructura de datos que se utiliza para almacenar una colección ordenada de elementos en Python. La principal diferencia entre una tupla y una lista es que una tupla es inmutable, lo que significa que una vez creada, **no se pueden modificar sus elementos**. A diferencia de las listas, las tuplas se definen utilizando paréntesis en lugar de corchetes. Al igual que las listas, los elementos de una tupla también pueden ser de diferentes tipos de datos y están indexados para acceder a ellos.

```
#! Tuplas

# Tupla de libros
libros = ("1984", "Matar a un ruiseñor", "Don Quijote de la Mancha", "sapiens")

# Acceso a elementos de la tupla
print(libros[0])  # Salida: 1984
print(libros[2])  # Salida: Don Quijote de la Mancha

# Intento de modificación de elementos (generará un error)
libros[1] = "Cien años de soledad"

# Concatenación de tuplas
libros_adicionales = ("Cien años de soledad", "Ulises")
todos_los_libros = libros + libros_adicionales
print(todos_los_libros)  # Salida: ("1984", "Matar a un ruiseñor", "Don Quijote de la Mancha", "sapiens", "Cien años de soleda
d", "Ulises")

# Desempaquetado de una tupla
libro1, libro2, libro3, libro4 = libros
print(libro1)  # Salida: 1984
print(libro2)  # Salida: Matar a un ruiseñor
```

Diferencias

Las diferencias que existen entre una lista y una tupla son:

1. **Mutabilidad:** Una lista es mutable, lo que significa que puedes modificar, agregar y eliminar elementos después de crearla. En cambio, una tupla es inmutable y no se pueden modificar sus elementos una vez que se ha creado.
2. **Sintaxis:** Para definir una lista, se utilizan corchetes `[ ]`, mientras que para definir una tupla se utilizan paréntesis `( )`. Por ejemplo:

   ```
   lista = [1, 2, 3]    # Lista
   tupla = (1, 2, 3)    # Tupla
   ```

3. **Uso de métodos:** Las listas ofrecen una amplia gama de métodos incorporados en Python, como `append()`, `remove()`, `sort()`, etc., que permiten modificar la lista. En cambio, las tuplas tienen menos métodos disponibles debido a su naturaleza inmutable.
4. **Rendimiento:** En general, las tuplas son más eficientes en términos de rendimiento que las listas. Como las tuplas son inmutables, ocupan menos espacio en la memoria y se pueden acceder más rápidamente. Por lo tanto, si tienes una colección de elementos que no necesitas modificar, las tuplas pueden ser más adecuadas.
5. **Uso y propósito:** Las listas se utilizan comúnmente cuando necesitas una estructura de datos que pueda modificarse durante la ejecución de tu programa. Por otro lado, las tuplas son útiles cuando deseas garantizar que los elementos no se modifiquen accidentalmente y se utilicen como valores que no cambiarán.

---

2) Espacio de Nombres en Python

Un espacio de nombres (namespace) es un sistema que se utiliza para mapear los nombres de las variables a los objetos correspondientes. Proporciona un contexto en el que los nombres únicos pueden asignarse a objetos para evitar conflictos y permitir la organización y estructuración del código.

Un espacio de nombres puede considerarse como un diccionario en el que los nombres son las claves y los objetos son los valores asociados. Cada espacio de nombres está asociado a un ámbito particular, que puede ser global (a nivel de módulo) o local (a nivel de función o clase). Cuando se crea una variable o se define una función en Python, se guarda en el espacio de nombres correspondiente.

Existen varios tipos de espacios de nombres en Python:

1. **Espacio de nombres global**: Es el espacio de nombres asociado al módulo en el que se define el código. Todas las variables y funciones definidas a nivel de módulo pertenecen a este espacio de nombres y son accesibles desde cualquier parte del módulo.
2. **Espacio de nombres local:** Es el espacio de nombres asociado a una función o método específico. Las variables y parámetros definidos dentro de la función se almacenan en este espacio de nombres y solo son accesibles dentro del ámbito de la función.
3. **Espacio de nombres incorporado:** Es el espacio de nombres que contiene todas las funciones y variables incorporadas en Python, como `print()`, `len()`, `range()`, entre otras. Estas funciones y variables están disponibles

automáticamente en todos los módulos sin necesidad de importar ningún módulo adicional.

```
# Espacio de nombres global
nombre = "Juan"

def saludar():
    # Espacio de nombres local
    nombre = "María"
    print("Hola,", nombre)

saludar()           # Salida: Hola, María
print("Hola,", nombre)   # Salida: Hola, Juan
```

3) Variables Locales y Globales

las variables locales y globales son dos tipos de variables que se utilizan para almacenar valores y datos dentro de un programa. La diferencia principal entre ellas radica en su alcance y accesibilidad dentro del programa.

Una variable local está restringida a un ámbito específico, como una función, y sólo es accesible dentro de ese ámbito; mientras que una variable global es accesible desde cualquier parte del programa.

```
def scope_test():
def do_local():
        spam = "local spam"

def do_nonlocal():
nonlocal spam
        spam = "nonlocal spam"

def do_global():
global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

💡 Diferencia con los espacios de nombre: los espacios de nombres son contextos que contienen variables y mapean nombres a objetos, mientras que las variables locales y globales son los nombres de las variables definidas dentro de un ámbito específico (local) o en todo el módulo (global). Los espacios de nombres permiten organizar y evitar conflictos entre los nombres de las variables, mientras que las variables locales y globales definen el alcance y la accesibilidad de las variables en un programa.

4) Entornos de desarrollo

Un IDE (Entorno de Desarrollo Integrado, por sus siglas en inglés) es una herramienta de software que proporciona un conjunto completo de características y funcionalidades para facilitar el desarrollo de software. Un IDE típicamente incluye un editor de código, un compilador o intérprete, herramientas de depuración, un explorador de archivos y otras características que ayudan a los desarrolladores a escribir, probar y depurar su código de manera más eficiente.

Para programar en Python podemos utilizar:

PyCharm

kdevelop

Jupyter Notebook(lab)

5) Módulos en Python

En Python, los módulos son archivos que contienen definiciones de variables, funciones y clases que se pueden utilizar en otros programas. Los módulos permiten organizar y reutilizar el código, ya que puedes dividir tu programa en varios archivos y luego importar los módulos necesarios en tu programa principal.

Si sales del intérprete de Python y vuelves a entrar, las definiciones que habías hecho (funciones y variables) se pierden. Por lo tanto, si quieres escribir un programa más o menos largo, es mejor que utilices un editor de texto para preparar la entrada para el intérprete y ejecutarlo con ese archivo como entrada. Esto se conoce como crear un script. A medida que tu programa crezca, quizás quieras separarlo en varios archivos para que el mantenimiento sea más sencillo. Quizás también quieras usar una función útil que has escrito en distintos programas sin copiar su definición en cada programa.

Para soportar esto, Python tiene una manera de poner definiciones en un archivo y usarlos en un script o en una instancia del intérprete. Este tipo de ficheros se llama módulo; las definiciones de un módulo pueden ser importadas a otros módulos o al módulo principal (la colección de variables a las que tienes acceso en un script ejecutado en el nivel superior y en el modo calculadora).

Ejemplo de Módulo

```
# Fibonacci numbers moduledef fib(n):    # write Fibonacci series up to na, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to nresult = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Ahora entra en el intérprete de Python e importa este modulo con el siguiente comando:

`>>>import fibo`

Esto no añade los nombres de las funciones definidas en fibo directamente al actual namespace (ver Ámbitos y espacios de nombres en Python para más detalles); sólo añade el nombre del módulo `fibo` allí. Usando el nombre del módulo puedes acceder a las funciones:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Si pretendes utilizar una función frecuentemente puedes asignarla a un nombre local:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6) Diferencia entre Matriz y Lista

En Python, la diferencia entre matrices y listas radica principalmente en su estructura y en cómo se accede a los elementos. Aquí se presentan las principales diferencias y cuándo conviene usar una u otra:

Estructura:

- Lista: Es una estructura de datos que puede contener elementos de diferentes tipos y crecer o reducir su tamaño dinámicamente. Se representa utilizando corchetes (`[]`).
- Matriz: No hay un tipo de datos de matriz incorporado en Python, pero se puede implementar utilizando listas anidadas. Cada lista interna representa una fila de la matriz, y las listas externas contienen las filas completas de la matriz.

Acceso a elementos:

- Lista: Los elementos de una lista se acceden mediante índices. Puedes acceder a un elemento específico utilizando `lista[indice]`.
- Matriz: Los elementos de una matriz se acceden utilizando índices dobles. Puedes acceder a un elemento específico utilizando `matriz[fila][columna]`.

Tipo de elementos:

- Lista: Puedes almacenar elementos de diferentes tipos en una lista, como números, cadenas, booleanos u otros objetos.
- Matriz: Por lo general, se utiliza para almacenar elementos homogéneos, como números, pero también puedes tener matrices de cadenas o cualquier otro tipo de objeto.

Tamaño:

- Lista: Las listas pueden crecer o reducir su tamaño dinámicamente. Puedes agregar elementos al final de la lista utilizando el método `append()` o eliminar elementos utilizando los métodos `pop()` o `remove()`.
- Matriz: Las matrices tienen una estructura fija. Una vez definidas, no pueden cambiar su tamaño. Si necesitas agregar o eliminar filas o columnas, debes crear una nueva matriz.

Cuándo usar cada una:

- Usa una lista cuando necesites una estructura de datos flexible que pueda contener elementos de diferentes tipos y pueda crecer o reducir su tamaño según sea necesario.
- Usa una matriz cuando necesites una estructura bidimensional y homogénea para representar una tabla, una matriz numérica o una cuadrícula. Las matrices son útiles cuando tienes datos organizados en filas y columnas y necesitas realizar operaciones matemáticas o acceder a elementos mediante índices dobles.

---

7) Operadores en Python

Los operadores en Python son **símbolos especiales que se utilizan para realizar operaciones en expresiones y valores**. Los operadores permiten realizar cálculos aritméticos, comparaciones, asignaciones, operaciones lógicas, entre otros. Estos operadores actúan sobre uno o más operandos y producen un resultado.

Python ofrece varios tipos de operadores que se clasifican en diferentes categorías:

Operadores aritméticos:

- `+`: Suma dos valores.
- `-`: Resta el segundo valor del primero.
- `*`: Multiplica dos valores.
- `/`: Divide el primer valor por el segundo (división real).
- `//`: Divide el primer valor por el segundo y redondea al entero más cercano (división entera).
- `%`: Devuelve el residuo de la división entre el primer valor y el segundo.
- `**`: Realiza la potenciación del primer valor elevado al segundo.

```
a = 10
b = 5

suma = a + b       # Suma
resta = a - b      # Resta
multiplicacion = a * b   # Multiplicación
division = a / b   # División (real)
division_entera = a // b   # División entera
residuo = a % b    # Residuo
potencia = a ** b  # Potenciación
```

Operadores de asignación:

- `=`: Asigna un valor a una variable.
- `+=`: Suma y asigna el resultado a una variable.
- `-=`: Resta y asigna el resultado a una variable.
- `*=`: Multiplica y asigna el resultado a una variable.
- `/=`: Divide y asigna el resultado a una variable.
- `//=`: Divide y redondea al entero más cercano, y asigna el resultado a una variable.
- `%=`: Calcula el residuo de la división y asigna el resultado a una variable.
- `**=`: Realiza la potenciación y asigna el resultado a una variable.

```
x = 10

x += 5   # Equivalente a: x = x + 5
x -= 3   # Equivalente a: x = x - 3
x *= 2   # Equivalente a: x = x * 2
x /= 4   # Equivalente a: x = x / 4
x %= 3   # Equivalente a: x = x % 3

print(x)   # Salida: 1
```

Operadores de comparación:

- `==`: Comprueba si dos valores son iguales.
- `!=`: Comprueba si dos valores son diferentes.
- `>`: Comprueba si el primer valor es mayor que el segundo.
- `<`: Comprueba si el primer valor es menor que el segundo.
- `>=`: Comprueba si el primer valor es mayor o igual que el segundo.
- `<=`: Comprueba si el primer valor es menor o igual que el segundo.

```
a = 5
b = 10

print(a == b)   # Salida: False
print(a != b)   # Salida: True
print(a > b)    # Salida: False
print(a < b)    # Salida: True
print(a >= b)   # Salida: False
print(a <= b)   # Salida: True
```

Operadores lógicos:

- `and`: Devuelve `True` si ambas condiciones son verdaderas.
- `or`: Devuelve `True` si al menos una de las condiciones es verdadera.
- `not`: Niega el valor de una condición, devuelve `True` si la condición es falsa.

```
x = 5
y = 10

print(x > 0 and y < 20)   # Salida: True
```

```
print(x > 0 or y < 0)     # Salida: True
print(not x > 0)         # Salida: False
```

Operadores de pertenencia:

- `in`: Devuelve `True` si un valor está presente en una secuencia.
- `not in`: Devuelve `True` si un valor no está presente en una secuencia.

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

print("manzana" in frutas)   # Salida: True
print("pera" in frutas)      # Salida: False
print("banana" not in frutas)   # Salida: False
```

Operadores de identidad:

- `is`: Devuelve `True` si dos variables apuntan al mismo objeto.
- `is not`: Devuelve `True` si dos variables no apuntan al mismo objeto.

```
a = 5
b = a
c = 10

print(a is b)   # Salida: True
print(a is c)   # Salida: False
print(a is not c)   # Salida: True
```