

✓ Introducción a Python

Autor: German Preciat; Universidad de Guadalajara

Carrera: Ing. Biomédica

Materia: Analisis de datos clínicos

Contacto: german.preciat@academicos.udg.mx

Python es un lenguaje de programación de alto nivel utilizado en una variedad de aplicaciones. Su nombre proviene de la comedia británica "Monty Python's Flying Circus". Python es conocido por su facilidad de uso y legibilidad de código, lo que lo convierte en una elección popular para programadores de todos los niveles de experiencia.

Python es versátil y puede utilizarse para realizar cálculos numéricos, manipular datos, crear aplicaciones web, desarrollar inteligencia artificial y mucho más. Es un lenguaje interpretado, lo que significa que puedes escribir y ejecutar código sin necesidad de compilarlo previamente.

Una de las ventajas de Python es su amplia comunidad de desarrolladores y la disponibilidad de numerosas bibliotecas y marcos de trabajo que facilitan el desarrollo de aplicaciones y la resolución de problemas en diversas áreas, como la ciencia de datos, la inteligencia artificial, la programación web y más.

El propósito de esta práctica es familiarizarse con algunas de las funciones básicas de Python y explorar su versatilidad en diversas aplicaciones.

✓ 1 Operaciones básicas y declaración de variables

En Python, puedes realizar operaciones tanto desde la consola interactiva como en un entorno de cuaderno (Notebook).

Los comentarios desempeñan un papel fundamental en la programación, ya que te ayudan a comprender y documentar tu código. Por lo tanto, es esencial incorporar comentarios en tu código para mejorar su legibilidad y comprensión.

En Python los comentarios se escriben con el caracter #:

```
# Esto es un comentario
```

```
2 + 45 # Suma
```

```
52 - 44 # Resta
```

```
4 * 12 # Multiplicación
```

```
225 / 15 # División
```

```
8 ** 2 # Potencia (8 elevado a la 2)
```

```
import math # Importa el módulo math para funciones matemáticas
math.sqrt(81) # Raíz cuadrada de 81
```

```
(12 + 2) * 4 # Paréntesis (12 + 2) multiplicado por 4
```

```
12 + (2 * 4) # 12 sumado a (2 multiplicado por 4)
```

```
↩ 20
```

```
1 > 20 # Desigualdad (resultado verdadero si 1 es mayor que 20, de lo contrario, falso)
```

```
↩ False
```

```
20 == 20 # Igualdad (verdadero si 20 es igual a 20, de lo contrario, falso)
```

```
↩ True
```


```
8 <= 20 # Desigualdad (verdadero si 8 es menor o igual a 20, de lo contrario, falso)
```

 True


```
list(range(1, 11)) # Serie con intervalo de una unidad (del 1 al 10)
```

 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```
list(range(1, 8, 2)) # Serie con intervalo de 2 unidades (del 1 al 7)
```

 [1, 3, 5, 7]

```
list(range(5, 0, -1)) # Serie con intervalo de -1 unidades (del 5 al 1)
```

 [5, 4, 3, 2, 1]

✓ 2 Librerías

En programación, una librería (o biblioteca) es un conjunto de funciones, clases y módulos predefinidos que pueden ser utilizados para realizar tareas específicas en un lenguaje de programación. Estas librerías contienen código previamente escrito y optimizado que te permite realizar ciertas operaciones sin necesidad de reinventar la rueda cada vez que las necesitas. En esencia, son como cajas de herramientas llenas de funciones útiles que puedes utilizar en tu código.

Nota sobre el uso de los puntos ('.'):

Los puntos ('.') se utilizan para acceder a funciones y atributos de una librería o un objeto en Python. En el caso de `numpy.size` y `numpy.shape`, el punto se utiliza para acceder a estas funciones en la librería **NumPy**.

A continuación, veremos ejemplos de dos librerías comunes en Python:

✓ Librería math

La librería `math` proporciona funciones matemáticas para realizar operaciones matemáticas avanzadas. Aquí tienes un ejemplo de cómo utilizarla para calcular la raíz cuadrada de un número:

```
# Esto es un comentario
```

```
import math
```

```
numero = 16  
raiz_cuadrada = math.sqrt(numero)  
print(raiz_cuadrada)
```

 4.0

En este ejemplo, importamos la librería `math`, luego utilizamos `math.sqrt()` para calcular la raíz cuadrada del número 16.

✓ Librería numpy

La librería `numpy` (numerical Python) es ampliamente utilizada en la computación científica y numérica. Proporciona un soporte eficiente para arrays multidimensionales y operaciones matemáticas en estos arrays. Aquí tienes un ejemplo de cómo utilizarla para realizar una suma de dos arrays:

```
import numpy as np
```

```
array1 = np.array([1, 2, 3])  
array2 = np.array([4, 5, 6])
```

```
resultado = array1 + array2  
print(resultado)
```

 [5 7 9]

En este caso, importamos la librería `numpy` como `np`, creamos dos arrays y luego utilizamos el operador `+` para realizar una suma de elementos en paralelo.

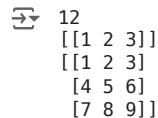
✓ Crear matrices y vectores

Un número, un vector, una matriz o el resultado de una operación se pueden almacenar en variables en Python, las cuales no pueden comenzar con números o símbolos. Para definir vectores y matrices, utilizamos corchetes `[]`. Cada nuevo elemento de un vector o fila de una matriz se separa por comas.

```
a = 12
b = a * 2
v1 = np.array([[1], [2]]) # Columna de 2 elementos
v2 = np.array([1, 2, 3]) # Vector de 1x3
M1 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]]) # Matriz de 2x4
M2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) # Matriz de 3x3
```

En Python, puedes utilizar la función `print()` para visualizar los resultados de variables, expresiones o cualquier otro contenido que desees mostrar en la consola. La función `print()` es una herramienta esencial para visualizar resultados y depurar tu código en Python. Puedes usarla para mostrar variables, resultados de cálculos, mensajes informativos y más.

```
print(a)
print(v2)
print(M2)
```



```
12
[1 2 3]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Para conocer las dimensiones de un vector o matriz en Python, usamos las siguientes funciones de la librería `numpy`

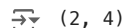
```
# Número de elementos en la matriz o vector
num_elements = np.size(M1)
print(num_elements)
```



```
8
```

`numpy.size` es una función de la librería **NumPy** que se utiliza para determinar el número total de elementos en un array. Básicamente, cuenta cuántos elementos hay en el array, independientemente de su forma (dimensiones).

```
# Dimensiones de la matriz o vector
dimensions = np.shape(M1)
print(dimensions)
```



```
(2, 4)
```

```
# Máxima dimensión de la matriz o vector
max_dimension = max(np.shape(M1))
print(max_dimension)
```



```
4
```

```
# Número de filas
num_rows = np.shape(M1)[0]
print(num_rows)
```



```
2
```

```
# Número de columnas
num_cols = np.shape(M1)[1]
print(num_cols)
```



```
4
```

`numpy.shape` es un atributo de los arrays de **NumPy** que te proporciona información sobre la forma o las dimensiones del array. Devuelve una tupla que representa la longitud de cada dimensión del array.

✓ Consultar la documentación

Para consultar la documentación de **NumPy** o cualquier otra librería en Python, puedes utilizar la función `help()`.

Por ejemplo:

```
import numpy as np
```

```
# Consultar la documentación de numpy.size
help(np.size)
```

🔗 Help on `_ArrayFunctionDispatcher` in module `numpy`:

```
size(a, axis=None)
    Return the number of elements along a given axis.

    Parameters
    -----
    a : array_like
        Input data.
    axis : int, optional
        Axis along which the elements are counted. By default, give
        the total number of elements.

    Returns
    -----
    element_count : int
        Number of elements along the specified axis.

    See Also
    -----
    shape : dimensions of array
    ndarray.shape : dimensions of array
    ndarray.size : number of elements in array

    Examples
    -----
    >>> a = np.array([[1,2,3],[4,5,6]])
    >>> np.size(a)
    6
    >>> np.size(a,1)
    3
    >>> np.size(a,0)
    2
```

✓ 3 Operación entre vectores y matrices

Para realizar operaciones entre matrices y vectores en Python, es importante tener en cuenta sus dimensiones.

Puedes utilizar la función `numpy.transpose()` o el operador `.T` para trasponer una matriz o un vector. La transposición es especialmente útil para asegurarte de que las dimensiones de las matrices sean compatibles para operaciones matriciales.

```
import numpy as np
```

```
# Crear una matriz
matriz = np.array([[1, 2, 3], [4, 5, 6]])
```

```
# Trasponer la matriz
matriz_transpuesta = np.transpose(matriz)
# O alternativamente:
matriz_transpuesta = matriz.T
```

```
print("Matriz original:")
print(matriz)
```

```
print("Matriz transpuesta:")
print(matriz_transpuesta)
```

```

↵ Matriz original:
  [[1 2 3]
   [4 5 6]]
Matriz transpuesta:
  [[1 4]
   [2 5]
   [3 6]]

```

El comportamiento de `.T` para realizar la transposición no se aplica automáticamente a todas las variables en Python. En realidad, es una característica proporcionada por la librería **NumPy** y se aplica a las matrices **NumPy** específicamente.

Cuando creas una matriz **NumPy**

```

import numpy as np

matriz = np.array([[1, 2, 3], [4, 5, 6]])

```

Estás creando una instancia de la clase `numpy.ndarray` de **NumPy**. Esta clase tiene propiedades y métodos específicos, y una de las propiedades es `.T`, que se usa para realizar la transposición de esa matriz específica.

Por lo tanto, si tienes otra variable no relacionada con **NumPy** y le intentas aplicar `.T`, es probable que obtengas un error porque esa propiedad no existe para variables comunes en Python. Solo las matrices NumPy tienen esta propiedad para realizar la transposición de manera eficaz.

Entonces, `.T` se aplica a matrices **NumPy** para hacer que las operaciones matriciales sean más convenientes, pero no funciona en todas las variables de Python en general.

```
print(v1) # Imprimir el vector v1
```

```
↵ [[1]
   [2]]
```

```
print(v2) # Imprimir el vector v2
```

```
↵ [[1 2 3]]
```

```
v3 = v2.T # Transponer v2
print(v3)
```

```
↵ [[1]
   [2]
   [3]]
```

```
v4 = v3 * a # Multiplicar v3 por la constante a
print(v4)
```

```
↵ [[12]
   [24]
   [36]]
```

```
M3 = M1.T # Transponer la matriz M1
print(M3)
```

```
↵ [[1 5]
   [2 6]
   [3 7]
   [4 8]]
```

```
M4 = np.outer(v1, v2) # Multiplicación de vectores v1 y v2
print(M4)
```

```
↵ [[1 2 3]
   [2 4 6]]
```

```
M5 = M4 * b # Multiplicación de la matriz M4 por la constante 'b'
print(M5)
```

```
↵ [[ 24  48  72]
   [ 48  96 144]]
```

```
M6 = np.hstack((M2, v3)) # Añadir una columna a la matriz M2
print(M6)
```

```
[[1 2 3 1]
 [4 5 6 2]
 [7 8 9 3]]
```

En Python, puedes utilizar la función `print()` para mostrar directamente el resultado de una expresión o acceso a un elemento. Por ejemplo, si deseas ver el primer elemento de un vector llamado `v1`, simplemente coloca `print(v1[0])`, y Python mostrará el valor del primer elemento en la consola.

En Python, para acceder a valores específicos dentro de un array (vector) o una matriz, utilizamos corchetes `[]`. Aquí hay un código de ejemplo que muestra cómo acceder a valores específicos y subconjuntos en Python utilizando NumPy:

```
print(v1[0]) # Acceder al primer elemento del vector v1
```

```
[1]
```

```
print(v1[-1]) # Acceder al último elemento del vector v1
```

```
[2]
```

```
print(M1[0, 0]) # Acceder al elemento en la fila 1, columna 1 de la matriz M1
```

```
1
```

```
print(M3[3, 1]) # Acceder al elemento en la fila 4, columna 2 de la matriz M1
```

```
8
```

```
print(M4[0, :]) # Todos los valores de la columna 1 en la matriz M4
```

```
[1 2 3]
```

```
print(M5[:, 2]) # Todos los valores de la fila 4 en M5
```

```
[ 72 144]
```

```
print(M6[1:, 1:3]) # Matriz de 2x2 dentro de M6
```

```
[[5 6]
 [8 9]]
```

✓ 4 Strings y arreglos

En Python, las listas pueden contener tanto números como texto, definidos como cadenas de caracteres (strings). Una cadena de caracteres debe estar definida entre comillas simples (`' '`) o dobles (`" "`) para ser reconocida como texto.

Para unir dos strings, puedes usar el operador de suma (`+`) o agregar las variables de texto directamente.

Hay diferentes tipos de estructuras de datos:

- Listas (equivalentes a arreglos en MATLAB)
- DataFrames (equivalentes a tablas en MATLAB)
- Diccionarios (equivalentes a arreglos de estructuras en MATLAB)

✓ Listas

Ejemplo de manejo de strings y listas en Python:

```
str1 = '¡Hola'
str2 = ' Mundo!'
concatenado = str1 + str2 # Concatenar dos strings usando el operador de suma
print(concatenado)
```

```
¡Hola Mundo!
```

```
espacio = ' '
str3 = concatenado + 'espacio' + '¿Cómo estás?'
print(str3) # Otra forma de concatenar
```

```
➦ iHola Mundo!espacio¿Cómo estás?
```

```
str4 = 'Este es el número 2'
a = 12
str5 = 'Y este es el número ' + str(a)
lista1 = [str4, str5]
print(lista1)
```

```
➦ ['Este es el número 2', 'Y este es el número 12']
```

```
lista2 = ['Dos', 2, '2', 'Doce', a, str(a)]
print(lista2)
```

```
➦ ['Dos', 2, '2', 'Doce', 12, '12']
```

▼ DataFrames (Tablas)

Para trabajar con tablas en Python, puedes utilizar la biblioteca Pandas. A continuación, te muestro cómo realizar tablas.

```
import pandas as pd
```

```
# Crear un DataFrame (equivalente a una tabla)
data = {'strLetra': ['A', 'B', 'C'],
        'numero': [1, 2, 3],
        'strNumero': ['One', 'Two', 'Three']}
t1 = pd.DataFrame(data)
print(t1)
```

```
➦
```

	strLetra	numero	strNumero
0	A	1	One
1	B	2	Two
2	C	3	Three

```
# Crear otro DataFrame
data2 = {'cellLines': ['Linea1', 'Linea2', 'Linea3'],
        'noOfCells': [2, 7, 12],
        'atpProduction': [4.0, 5.0, 6.0]}
t2 = pd.DataFrame(data2)
print(t2)
```

```
➦
```

	cellLines	noOfCells	atpProduction
0	Linea1	2	4.0
1	Linea2	7	5.0
2	Linea3	12	6.0

▼ Diccionarios

En Python, un diccionario es una estructura de datos que permite almacenar y organizar información de manera eficiente. Un diccionario es una colección de pares clave-valor, donde cada clave se mapea a un valor asociado.

```
d1 = {
    'vector': v1,
    'matrix': M1,
    'string': str1,
    'array': array1,
    'table': t2
}
print(d1)
```

```
➦ {'vector': array([[1,
                    [2]]], 'matrix': array([[1, 2, 3, 4],
                    [5, 6, 7, 8]]), 'string': 'iHola', 'array': array([1, 2, 3]), 'table':
0    Linea1      2      4.0
1    Linea2      7      5.0
2    Linea3     12      6.0}
```

```

d2 = {
    'vectors': {'v1': v1, 'v2': v1 + 1},
    'matrices': {'M1': M1, 'M2': M1 * 2}
}
print(d2['vectors'])

↵ {'v1': array([[1],
                [2]]), 'v2': array([[2],
                [3]])}

print(d2['matrices'])

↵ {'M1': array([[1, 2, 3, 4],
                [5, 6, 7, 8]]), 'M2': array([[ 2,  4,  6,  8],
                [10, 12, 14, 16]])}

print(d2['vectors']['v1'])

↵ [[1]
    [2]]

print(d2['vectors']['v1'][0])

↵ [1]

```

✓ 5 Figuras

En Python, contamos con una variedad de bibliotecas y funciones para visualizar datos. Algunos ejemplos de bibliotecas populares son:

- **Matplotlib**: Utilizado para crear gráficos de líneas, barras, dispersión y mapas de calor.
- **Seaborn**: Proporciona una interfaz de alto nivel para crear gráficos estadísticos informativos y atractivos.
- **Plotly**: Ofrece capacidades de visualización interactiva y en línea.

Aquí tienes un ejemplo de cómo crear una figura y personalizarla con **Matplotlib**:

```

import matplotlib.pyplot as plt

# Datos
x = [1, 2, 3]
y = [1, 2, 3]

# Graficar v2 vs v3
plt.plot(x, y, label='Grafica 1')

# Sobreponer otra gráfica en la misma figura
plt.plot(x, [2 * val for val in y], 'g--', linewidth=2, label='Grafica 2')

# Agregar un título
plt.title('Mi grafica')

# Etiquetas para los ejes
plt.xlabel('Eje X')
plt.ylabel('Eje Y')

# Agregar una leyenda
plt.legend()

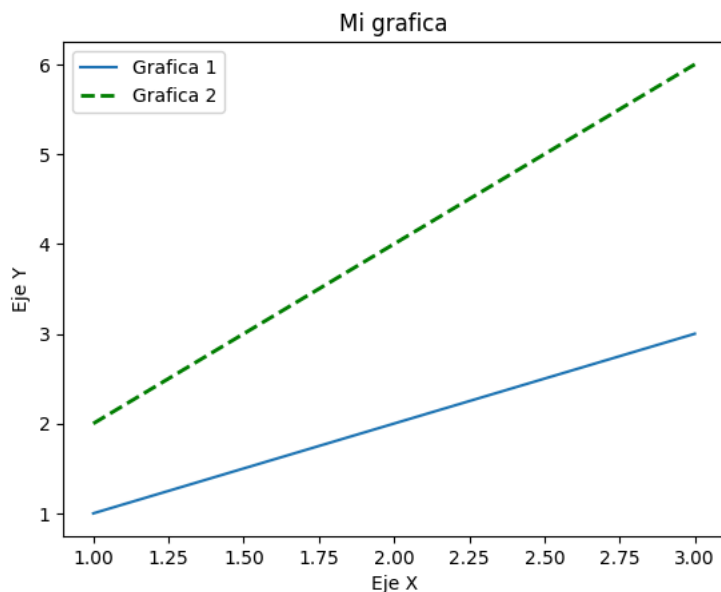
# Mostrar la gráfica
plt.show()

# Importar la biblioteca necesaria
from google.colab import files

# Guardar el gráfico en un archivo de imagen (por ejemplo, PNG)
plt.savefig('mi_grafico.png')

# Descargar el archivo guardado
files.download('mi_grafico.png')

```

<Figure size 640x480 with 0 Axes>

```
import matplotlib.pyplot as plt
```

```
# Datos
```

```
v2 = [1, 2, 3, 4, 5] # Reemplaza con tus datos
```

```
v3 = [3, 6, 2, 4, 8] # Reemplaza con tus datos
```

```
M5 = [[1, 2, 3], [4, 5, 6]] # Reemplaza con tus datos
```

```
M4 = [[7, 8, 9], [10, 11, 12]] # Reemplaza con tus datos
```

```
# Subplot 1: Gráfica de líneas
```

```
plt.subplot(2, 2, 1)
```

```
plt.plot(v2, v3)
```

```
plt.plot(v2, [2 * val for val in v3], 'g--', linewidth=2)
```

```
plt.title('Mi grafica')
```

```
plt.xlabel('Eje X')
```

```
plt.ylabel('Eje Y')
```

```
plt.legend(['Grafica 1', 'Grafica 2'])
```

```
# Subplot 2: Gráfica de barras
```

```
plt.subplot(2, 2, 2)
```

```
plt.bar(v2, v3)
```

```
plt.title('Gráfica de barras')
```

```
# Subplot 3: Gráfica de dispersión
```

```
plt.subplot(2, 2, 3)
```

```
plt.scatter(M5[0], M4[0], c='black', marker='o', label='Punto 1', alpha=1)
```

```
plt.scatter(M4[0], M5[0], c='blue', marker='o', label='Punto 2', alpha=1)
```

```
plt.scatter(M5[0], M5[0], c='red', marker='o', label='Punto 3', alpha=1)
```

```
plt.legend()
```

```
plt.title('Gráfica de dispersión')
```

```
# Subplot 4: Mapa de calor
```

```
plt.subplot(2, 2, 4)
```

```
plt.imshow(M5, cmap='YlOrRd', aspect='auto', interpolation='none')
```

```
plt.colorbar()
```

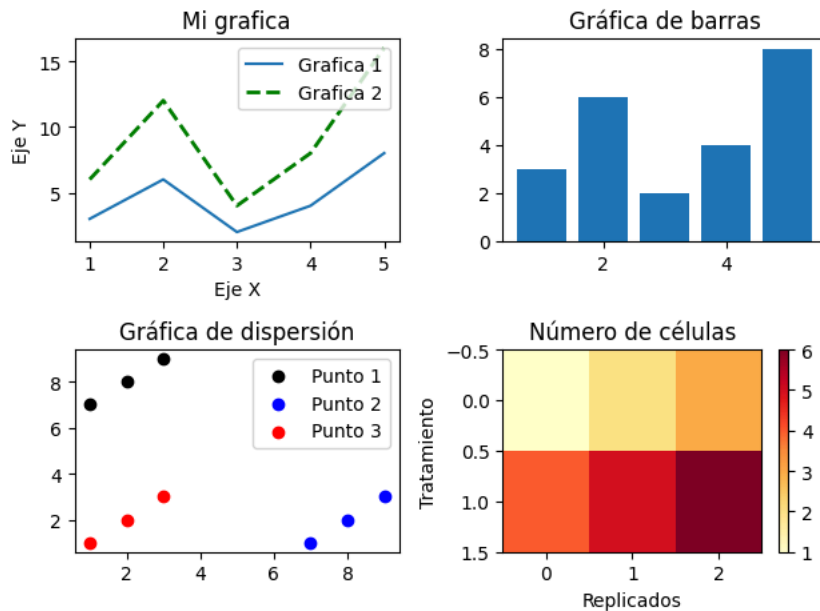
```
plt.title('Número de células')
```

```
plt.xlabel('Replicados')
```

```
plt.ylabel('Tratamiento')
```

```
plt.tight_layout() # Para evitar superposición de etiquetas
```

```
plt.show()
```



✓ 6 Estructuras de control

En lenguajes de programación, las estructuras de control permiten modificar el flujo de ejecución de las instrucciones de un programa. Con las estructuras de control se puede:

- De acuerdo con una condición, ejecutar un grupo u otro de sentencias (If)
- De acuerdo con el valor de una variable, ejecutar un grupo u otro de sentencias (if-elif-else)
- Ejecutar un grupo de sentencias un número determinado de veces (For)
- Ejecutar un grupo de sentencias solo cuando se cumpla una condición (While)

A continuación, algunos ejemplos de su uso.

✓ IF

Si `a` es mayor que 32 la variable `c` será igual a la variable `a`, sino será igual a 32.

```
if a > 32:
    c = a
else:
    c = 32
print(a)
```

12

✓ IF-ELIF-ELSE

Muestra texto diferente condicionalmente, dependiendo de un valor ingresado en el símbolo del sistema. La función `input` permite que el usuario ingrese un valor.

```
n = int(input('Ingresa un número:')) # Entrada de usuario (se convierte a entero)

if n == -1:
    print('menos uno')
elif n == 0:
    print('cero')
elif n == 1:
    print('uno')
else:
    print('otro valor')
```

```

-----
KeyboardInterrupt                                Traceback (most recent call last)
<ipython-input-57-1ee473a851a8> in <cell line: 1>()
----> 1 n = int(input('Ingresa un número:')) # Entrada de usuario (se convierte a entero)
      2
      3 if n == -1:
      4     print('menos uno')
      5 elif n == 0:

----- 1 frames -----
/usr/local/lib/python3.10/dist-packages/ipykernel/kernelbase.py in _input_request(self, prompt, ident, parent, password)
    893     except KeyboardInterrupt:
    894         # re-raise KeyboardInterrupt, to truncate traceback
--> 895         raise KeyboardInterrupt("Interrupted by user") from None
    896     except Exception as e:
    897         self.log.warning("Invalid Message:", exc_info=True)

KeyboardInterrupt: Interrupted by user

```

✓ FOR

Crear una matriz que al sumarla con M3 todos los valores sean igual a 12. Con `np.zeros` es posible generar un vector o matriz con las dimensiones definidas por el usuario. Otras funciones que se usan de forma similar son `np.ones` (crea una matriz o vector con las dimensiones indicadas).

```

# Conocer las dimensiones de M3
m, n = np.shape(M3)

# Crear una matriz M7 de ceros con las mismas dimensiones que M3
M7 = np.zeros((m, n))

# Recorrer cada fila de M3
for i in range(m):
    # Recorrer cada columna de M3
    for j in range(n):
        # Agregar un nuevo valor
        M7[i, j] = 12 - M3[i, j]

# Sumar M3 y M7
print(M3 + M7)
print(' ')
print(M7)

```

✓ WHILE

Calcular el factorial de 10 usando `while`.

```

n = 10
f = n

while n > 1:
    n = n - 1
    f = f * n

print(f'n! = {f}')

```

✓ 7 Scripts y funciones en Python

Las funciones y scripts son mecanismos que Python utiliza para simplificar la escritura de programas o la carga de datos iniciales. Pueden ser escritos directamente en la línea de comandos de Python o en archivos externos. En este caso, los scripts y las funciones se llaman de la misma manera que cualquier otra función o módulo en Python. La única condición es que estos archivos deben encontrarse en el directorio actual o en un directorio que esté incluido en la variable de entorno `PYTHONPATH`.

Para obtener más información sobre la configuración del `PYTHONPATH` y la ubicación de los archivos, consulta la documentación de Python.

```

def linear_equations_solver(matrices_and_vectors, print_level=0):
    """

```

Resuelve cada uno de los sistemas de ecuaciones lineales en matrices_and_vectors de la forma $A * x = b$

Args:

matrices_and_vectors (dict): Un diccionario con los siguientes campos:
 'matrices' (dict): Un diccionario de matrices.
 'vectors' (dict): Un diccionario de vectores.

print_level (int, optional): Nivel de información que se imprimirá (valor predeterminado: 0).

Returns:

dict: Un diccionario con los resultados de los sistemas resueltos.

Example:

```
result = linear_equations_solver(matrices_and_vectors, print_level)
"""
if print_level > 1:
    print("Comenzando...")
    print()

# Salvar el nombre de las variables en las matrices y vectores
name_of_variables_matrices = list(matrices_and_vectors['matrices'].keys())
name_of_variables_vectors = list(matrices_and_vectors['vectors'].keys())

if print_level > 1:
    print(f"Número de matrices: {len(name_of_variables_matrices)}")
    print(f"Número de vectores: {len(name_of_variables_vectors)}")
    print()

# Generar un diccionario vacío donde se guardarán los resultados
system_solution = {}

# Por cada una de las matrices haz la operación
for i in range(len(name_of_variables_matrices)):
    # Salvar la matriz en la variable "A" y en el vector la variable "b"
    A = matrices_and_vectors['matrices'][name_of_variables_matrices[i]]
    b = matrices_and_vectors['vectors'][name_of_variables_vectors[i]]

    # Resolver el sistema de ecuaciones lineales
    solution = np.linalg.solve(A, b)

    # Salvar los resultados de una matriz
    system_solution[f'Solution{i + 1}'] = solution

# Imprime mensajes al usuario
if print_level == 1:
    print(f"Sistemas resueltos: {len(name_of_variables_matrices)}")
elif print_level == 2:
    for i in range(len(name_of_variables_matrices)):
        print(f"Resultados del sistema {i + 1}:")
        print(system_solution[f'Solution{i + 1}'])
elif print_level > 2:
    print("Nivel de impresión no reconocido")

return system_solution
```

```
help(linear_equations_solver)
```

```
import numpy as np
```

```
matrices_and_vectors = {
    'matrices': {
        'A1': np.array([[2, 1], [1, 3]]),
        'A2': np.array([[3, 2], [1, 2]])
    },
    'vectors': {
        'b1': np.array([5, 4]),
        'b2': np.array([6, 3])
    }
}

result = linear_equations_solver(matrices_and_vectors, print_level=2)
print(result)
```

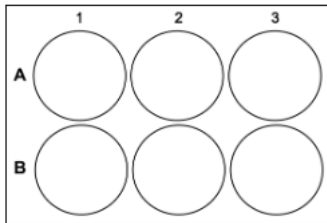
✓ 8 Ejercicio

Ejercicio Tienes diferentes muestras de microbiota humana donde se encuentran diferentes organismos, entre ellos bacterias, arqueas, hongos, protistas y virus. Estas muestras te seguirán llegando frecuentemente. Sin embargo, solo 1 muestra puede ser analizada a la vez en el espectrómetro de masas. El laboratorio se especializa en bacterias arqueas y protistas y tu obtienes cuantos pares de base se identificaron de cada organismo en cada pozo.

Necesitas desarrollar una función que identifique la muestra con mayor expresión de bacterias arqueas y protistas, además de generar una imagen para mostrarle a los jefes en una presentación.

✓ Paso 1 - Definir variables

Las muestras las tienen en una caja de cultivo de 6 pozos:



Crea un arreglo (e.g., pozos) con 6 strings que contenga la etiqueta de los pozos (a1, a2, a3, b1, b2 y b3).

Start coding or [generate](#) with AI.

Crea una matriz (A1) de 6x5 (muestras x organismos) con los pares de base identificados en cada muestra con los datos que están en la siguiente matriz en kilobase (kb):

$$A_1 = \begin{matrix} & \begin{matrix} Bacterias \\ Arqueas \\ Hongos \\ Protistas \\ Virus \end{matrix} & \\ \begin{matrix} a1 \\ a2 \\ a3 \\ b1 \\ b2 \\ b3 \end{matrix} & \begin{bmatrix} 4 & 4 & 2 & 5 & 1 \\ 1 & 2 & 1 & 1 & 3 \\ 5 & 4 & 1 & 3 & 3 \\ 5 & 1 & 5 & 2 & 4 \\ 4 & 4 & 4 & 4 & 4 \\ 4 & 1 & 2 & 4 & 4 \end{bmatrix} \end{matrix}$$

Start coding or [generate](#) with AI.

La siguiente muestra llega por replicado (M1 y M2). Necesitas definir M1 y M2 y sacar el promedio de la expresión entre ambas muestras (A2).

$$M_1 = \begin{bmatrix} 2 & 5 & 3 & 1 & 5 \\ 4 & 2 & 4 & 2 & 2 \\ 4 & 3 & 5 & 5 & 1 \\ 1 & 2 & 5 & 2 & 2 \\ 1 & 4 & 3 & 5 & 4 \\ 3 & 2 & 1 & 2 & 3 \end{bmatrix}, M_2 = \begin{bmatrix} 2 & 4 & 3 & 1 & 1 \\ 5 & 4 & 4 & 2 & 4 \\ 3 & 2 & 5 & 1 & 2 \\ 3 & 3 & 1 & 4 & 4 \\ 5 & 1 & 3 & 2 & 4 \\ 2 & 1 & 3 & 3 & 4 \end{bmatrix}$$

Start coding or [generate](#) with AI.

Ahora te llegaron 4 muestras más (A3, A4, A5, A6), utiliza el método "randint" para inventarte las kb encontradas en cada muestra y su rango sea de entre 1-5kb.

Start coding or [generate](#) with AI.

✓ Paso 2 - Hacer un script que resuelva la ecuación $Ax=b$

Ya que queremos la muestra más que tenga más pares de bases de organismos específicos necesitamos: Definir x como:

$$x = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{matrix} \text{Bacterias} \\ \text{Arqueas} \\ \text{Hongos} \\ \text{Protistas} \\ \text{Virus} \end{matrix}$$

(recuerda que no es lo mismo $m \times n$ que $n \times m$) Encontrar b usando $A1$ en $Ax=b$ ($Sv=0$) Encontrar la muestra más interesante para el laboratorio.

Start coding or [generate](#) with AI.

✓ Paso 3 - Hacer una funcion que imprima los resultados

- Guardar el arreglo wells en el diccionario experimentalData.
- Guardar las matrices A1, A2, A3, A4, A5 y A6 en el diccionario experimentalData.
- Guardar el vector x indicando nuestros organismos de interés en el diccionario experimentalData.
- Hacer una función de la forma `"wellSelectedPerSample = findTheBestSample(experimentalData, printLevel)";` Pista usar `open linearEquationsSolver` Incluir descripción de la función.
- `wellSelectedPerSample` señala el pozo que se debe de utilizar por muestra (pozo: *an* o *bn*; Paso 2 y lo ultimo de la Seccion 03. Operación entre vectores y matrices).
- `printLevel` igual 1 imprime una tabla que indique la expresión combinada de los organismosde interés por cada muestra en cada pozo y algunos mensajes del progreso de la función.
- `printLevel` igual 2 imprime una figura señalando la expresión combinada de los organismos de interés por muestra (en ; usar subplot y heatmap). Reorganizar de a para representar una caja de cultivo.