Introduccion a Numpy

April 5, 2024

1 Introducción a NumPy

Elaboración: Gabriel Armando Landín Alvarado

1.1 Contenido

- Definición de la libreria NumPy
- Crear objetos array
  - Array de una dimensión
  - Tipos de dato
  - Array de dos dimensiones
  - Atributos
    - Array de tres dimensiones
  - Crear array básico
  - Crear array a partir de una entrada
- Indexación y segmentación
- Manipular la forma
- Insertar, eliminar y ordenar elementos
- Unir y dividir
- Funciones estadísticas
  - Mínimos y máximos
  - Media y media ponderada
  - Mediana
  - Desviación estándar
  - Varianza
  - Percentiles
- Operaciones númericas básicas
- Escritura en archivos
- Lectura de archivos

1.2 Definición de la libreria NumPy

NumPy (Python-Númerico) es una librería de Python para análisis de datos númericos, es importante destacar que es mucho más eficiente en el uso de memoria que **Pandas**. **NumPy** da soporte en la creación de objetos de tipo vector(**unidimensional**) o matrices (**bidimensional**) comúnmente nombrados array(s), se puede crear arrays de tres o más dimensiones (**ndarray**), esto podría entenderse como una forma apilada de 'n' arrays o matrices. **NumPy** provee una variedad de funciones que operan de manera muy rápida, se incluyen por supuesto las de tipo matemático de alto nivel, pero también se tienen las de tipo lógico, de forma, ordenamiento y selección, así

como operaciones de computación científica, como son las transformadas de Fourier, álgebra lineal, operaciones de estadística, simulación aleatoria, entre otras.

1.3 Crear objetos array

Para craer un objeto de tipo array NumPy provee la función **array()** que recibe como parámetros básicos un objeto y el tipo de dato. En el siguiente ejemplo, lo primero será importar la librería NumPy y asignarle un alias, en este caso por convención será **'np'**, lo siguiente es crear un par de arrays a partir de una lista y tupla de python, ambos array son de una dimensión (1-D).

1.3.1 Array de una dimensión

[1]:
```
import numpy as np

array_1 = np.array([1, 2, 3, 4, 5])
array_2 = np.array((6, 7, 8, 9, 10))
print('Array 1 de 1-D:', array_1)
print('Array 2 de 1-D:', array_2)
```

```
Array 1 de 1-D: [1 2 3 4 5]
Array 2 de 1-D: [ 6  7  8  9 10]
```

1.3.2 Tipos de dato

Algunos de los tipos de datos más comunes que incluye Numpy son:

- **int32, int64:** enteros de 32 y 64 bits.
- **float32, float64:** números decimales o flotantes de 32 y 64 bits.
- **complex64, complex128:** números complejos de 64 y 128 bits.
- **bool:** Booleanos, representados por verdadera y falso (True y False).
- **object:** Tipo de datos genéricos que permite almacenar cualquier objeto de Python.
- **string_:** Tipo de datos para cadenas de longitud fija.
- **unicode_:** Tipo de datos para cadena Unicode de longitud fija.

A continuación, se muestra la creación de solo algunos arrays definiendo el tipo de dato.

[2]:
```
array_float32 = np.array([1, 2, 3, 4, 5], dtype='float32')
print('Array de 1-D tipo float:', array_float32,'\nTipo de dato:',␣
 ↪array_float32.dtype)
```

```
Array de 1-D tipo float: [1. 2. 3. 4. 5.]
Tipo de dato: float32
```

[3]:
```
array_complex = np.array([1, 2, 3, 4, 5], dtype='complex64')
print('Array de 1-D números complejos:', array_complex,'\nTipo de dato:',␣
 ↪array_complex.dtype)
```

```
Array de 1-D números complejos: [1.+0.j 2.+0.j 3.+0.j 4.+0.j 5.+0.j]
Tipo de dato: complex64
```

1.3.3 Array de dos dimensiones

Para crear un array de dos dimensiones se pasa como argumento a la función **array()** secuencias (listas o tuplas) de secuencias (listas o tuplas), y si se quiere un array de 3 dimensiones se pasan secuencias de secuencias, y así sucesivamente.

[131]:
```
array_2d_lista = np.array([[1, 2, 3], [4, 5, 6]])
print("Array de dos dimensiones:\n", array_2d_lista)
```

```
Array de dos dimensiones:
 [[1 2 3]
 [4 5 6]]
```

[132]:
```
array_2d_tupla = np.array(((7, 8, 9), (10, 11, 12)))
print("Array de dos dimensiones:\n", array_2d_tupla)
```

```
Array de dos dimensiones:
 [[ 7  8  9]
 [10 11 12]]
```

Una forma de entender un array o matriz de dos dimensiones es verlo como una tabla de dos ejes, es decir, **axis=0 para filas y axis=1 para columnas**, en los dos casos anteriores, se crearon dos arrays con 2 filas y 3 columnas. Resulta importante mencionar que se puede acceder a los elementos del array por medio de sus índices, asimismo, cabe recordar que los índices en Python se inician en cero, conviene tener esto en consideración para más adelante cuando se vea el tema de **slicing** o segmentación.

1.3.4 Atributos

En esta sección se cubren algunos de los atributos del array, uno que ya se mencionó fue **dtype**, este proporciona la información del tipo de dato. Otros importantes a tener en cuenta son:

- **ndim**: número de dimensiones de un array.
- **shape**: devuelve una tupla de enteros especificando el número de elementos a lo largo de cada dimensión del array.
- **size**: número fijo de elementos que están contenidos dentro del array.

[5]:
```
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype='int64')
print(a)
```

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

[6]:
```
a.dtype
```

[6]:
```
dtype('int64')
```

[7]:
```
a.ndim
```

[7]:
```
2
```

[8]:
```
a.shape
```

[8]:
```
(3, 3)
```

[9]:
```
a.size
```

[9]:
```
9
```

Array de tres dimensiones Después de conocer algunos atributos, es importante hacer un paréntesis para mostrar el ejemplo de un array de 3 dimensiones, asimismo, hacer uso de sus atributos.

[137]:
```
array_3d = np.array([ [[1, 2],[3, 4]], [[5, 6],[7, 8]] ])
print(array_3d)
```

```
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```

[138]:
```
array_3d.ndim
```

[138]:
```
3
```

[139]:
```
array_3d.shape
```

[139]:
```
(2, 2, 2)
```

[140]:
```
array_3d.size
```

[140]:
```
8
```

1.3.5 Crear array básico

NumPy ofrece algunos métodos o funciones para crear arrays básicos, como son: secuencias de ceros, de unos, valores espaciados en un intervalo específico, entre otros más, algunos ejemplos son los siguientes:

- **zeros()**: crea un array con una secuencia de ceros, se pasa como parámetro la forma o shape.
- **ones()**: crea un array con una secuencia de unos, se pasa como parámetro la forma o shape.
- **empty()**: crea un array cuyo contenido inicial es aleatorio y depende del estado de la memoria, puede que retorne ceros o no.
- **arange()**: crea un array con un rango de elementos, con un solo parámetro asigna el número de elementos, con dos parámetros se señalan el inicio y fin del rango, un tercer parámetro indica los saltos o pasos entre elementos.
- **linspace()**: crea un array con valores espaciados linealmente asignando un intervalo específico.

- **eye()**: crea un array cuadrado (mismas filas y columnas) donde los elementos de la diagonal principal son unos y el resto son ceros.

[14]:
```
array_ceros = np.zeros((3, 4))
print(array_ceros)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

[15]:
```
array_unos = np.ones((4, 3))
print(array_unos)
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

[16]:
```
array_empty = np.empty((3, 2))
print(array_empty)
```

```
[[ 4.94065646e-324 -1.92216604e-311]
 [ 2.12199579e-314  4.94065646e-324]
 [ 2.33419537e-312  0.00000000e+000]]
```

[17]:
```
array_arange1 = np.arange(5) # total de elementos
print(array_arange1)
```

```
[0 1 2 3 4]
```

[18]:
```
array_arange2 = np.arange(1, 10) # se excluye el último número
print(array_arange2)
```

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

[19]:
```
array_arange3 = np.arange(1, 20, 2) # paso de 2
print(array_arange3)
```

```
[ 1  3  5  7  9 11 13 15 17 19]
```

[20]:
```
array_linspace = np.linspace(0, 10, num=5) # cinco intervalos
print(array_linspace)
```

```
[ 0.   2.5  5.   7.5 10. ]
```

[21]:
```
array_eye = np.eye(5) # shape o cuerpo de (5, 5)
print(array_eye)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
```

```
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

1.3.6 Crear array a partir de una entrada

Otra forma de crear un array es hacer uso de la función **asarray()**, esta recibe un **input o entrada** como parámetro, lo más usual es que sean objetos de tipo lista o tupla, o bien, una lista de listas o tupla de tuplas, asimismo, se puede definir el tipo de dato con el parámetro **dtype=' '**.

[22]:
```
lista_1 = [1, 2, 3, 4, 5]
lista_1_array = np.asarray(lista_1, dtype='int16')
print('Array con entrada de una lista:\n', lista_1_array)
```

```
Array con entrada de una lista:
 [1 2 3 4 5]
```

[23]:
```
lista_2 = [[1, 2, 3], [4, 5, 6]]
lista_2_array = np.asarray(lista_2, dtype='int16')
print('Array con entrada de una lista de listas:\n', lista_2_array)
```

```
Array con entrada de una lista de listas:
 [[1 2 3]
 [4 5 6]]
```

[24]:
```
tupla_1 = (1, 2, 3, 4, 5)
tupla_1_array = np.asarray(tupla_1, dtype='int16')
print('Array con entrada de una tupla:\n', tupla_1_array)
```

```
Array con entrada de una tupla:
 [1 2 3 4 5]
```

[25]:
```
tupla_2 = ((1, 2, 3), (4, 5, 6))
tupla_2_array = np.asarray(tupla_2, dtype='int16')
print('Array con entrada de una tupla de tuplas:\n', tupla_2_array)
```

```
Array con entrada de una tupla de tuplas:
 [[1 2 3]
 [4 5 6]]
```

1.4 Indexación y segmentación

Para accesar al contenido de un objeto array de **'n' dimensiones (n-D)** se tiene el mecanismo de indexación y segmentación o corte (**slicing**). La segmentación permite extraer un subconjunto de elementos de un array de forma eficiente. Este método es similar al trabajo con listas o tuplas, esto quiere decir, que cada elemento posee un número de índice, es importante recordar que los índices inician en cero, asimismo, se puede pasar un valor negativo como -1, esto significa que estamos accediendo al último elemento del array.

El mecanismo consiste en colocar un par de corchetes después del objeto, lo siguiente es agregar o no un valor con el número inicial del índice que se quiere segmentar, enseguida colocar dos puntos para indicar el número final del índice, es importante señalar que este valor es excluyente, es decir, no

se obtiene el elemento que posee este número de índice. Los primeros ejemplos de este mecanismo de segmentación se harán sobre un array de una dimensión (**1-D**) para una mejor comprensión.

[26]:
```
array_1d_slicing = np.arange(1, 15, 2)
print(array_1d_slicing)
```

```
[ 1  3  5  7  9 11 13]
```

[143]:
```
print(array_1d_slicing)
print('Obtener el total de los elementos del array:\n', array_1d_slicing[:])#␣
 ↪el dejar en blanco significa que se desea cortar desde el primero hasta el␣
 ↪último elemento
```

```
[ 1  3  5  7  9 11 13]
Obtener el total de los elementos del array:
 [ 1  3  5  7  9 11 13]
```

[144]:
```
print(array_1d_slicing)
print('Retornar desde el elemnto con índice 0 hasta el elemento con índice 1:
 ↪\n', array_1d_slicing[0:2])
```

```
[ 1  3  5  7  9 11 13]
Retornar desde el elemnto con índice 0 hasta el elemento con índice 1:
 [1 3]
```

[145]:
```
print(array_1d_slicing)
print("Obtener los primeros tres elementos, iniciando con el elemento con␣
 ↪índice 0 hasta el elemento con índice 2:\n", array_1d_slicing[:3])
```

```
[ 1  3  5  7  9 11 13]
Obtener los primeros tres elementos, iniciando con el elemento con índice 0
hasta el elemento con índice 2:
 [1 3 5]
```

[146]:
```
print(array_1d_slicing)
print("Retornar los últimos elementos del array a partir del elemento con␣
 ↪índice 3 hasta el último:\n", array_1d_slicing[3:])
```

```
[ 1  3  5  7  9 11 13]
Retornar los últimos elementos del array a partir del elemento con índice 3
hasta el último:
 [ 7  9 11 13]
```

[147]:
```
print(array_1d_slicing)
print("Segmentar desde el tercer elemento final del array hasta el segundo␣
 ↪antes del final:\n", array_1d_slicing[-3:-1])
print('\nCabe recordar que el segundo parámetro es excluyente.')
```

```
[ 1  3  5  7  9 11 13]
Segmentar desde el tercer elemento final del array hasta el segundo antes del
final:
```

```
 [ 9 11]
```

Cabe recordar que el segundo parámetro es excluyente.

Para segmentar un array de dos dimensiones es muy similar a lo anterior, solo imagine que se tiene que señalar los índices para ambos ejes, es decir, para filas y columnas, esto se logra mediante una coma. Para una mejor comprensión se realizan los siguientes ejemplos a partir de un array de 15 elementos creado con la función **arange()**, posteriormente, con la función **reshape** se define la forma del array, en este caso, 3 filas y 5 columnas.

[148]:
```
array_2d_slicing = np.arange(15)
array_2d_slicing = array_2d_slicing.reshape(3, 5)
print(array_2d_slicing)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

[34]:
```
print("Obtener solo la primera fila, es decir, la que posee el índice 0:\n",␣
 ↪array_2d_slicing[0])
```

```
Obtener solo la primera fila, es decir, la que posee el índice 0:
 [0 1 2 3 4]
```

[35]:
```
print("Retornar las primeras 2 filas del array, aquellas con índice 0 y 1:\n",␣
 ↪array_2d_slicing[:2])
```

```
Retornar las primeras 2 filas del array, aquellas con índice 0 y 1:
 [[0 1 2 3 4]
 [5 6 7 8 9]]
```

[149]:
```
print("Segmentar un subconjunto de dos dimensiones, a partir de las dos␣
 ↪primeras filas y las dos primeras columnas:\n", array_2d_slicing[0:2, 0:2])
```

```
Segmentar un subconjunto de dos dimensiones, a partir de las dos primeras filas
y las dos primeras columnas:
 [[0 1]
 [5 6]]
```

[151]:
```
print('Segmentar un array a partir de la fila con índice 2 hasta la última,␣
 ↪para el segundo eje, iniciando con la columna con índice 3 hasta la última:
 ↪\n', array_2d_slicing[2: , 3:])
```

```
Segmentar un array a partir de la fila con índice 2 hasta la última, para el
segundo eje, iniciando con la columna con índice 3 hasta la última:
 [[13 14]]
```

1.5 Manipular la forma

En en análisis de datos es común tener que lidiar con el cambio de tamaño y forma de los arrays, para esta tarea NumPy posee diversas funciones, a continuación se describen algunas de las más

comunes:

- **reshape()**: retorna un nuevo array con la nueva de forma pasada como parámetro sin modificar los datos.
- **flat()**: "aplana" el array a una dimensión (1-D) y retorna el elemento con el valor del índice solicitado.
- **flatten()**: retorna una copia del array de entrada en forma "plana" (1-D).
- **ravel()**: retorna el array de entrada en forma "plana" (1-D).
- **transpose()**: transpone los ejes del array de 2 dimensiones (2-D).
- **resize()**: retorna un nuevo array con las especificaciones de forma pasadas como parámetro modificando los datos con base en el array de entrada.
- **flip()**: retorna el array con los elementos invertidos en su orden de acuerdo con un eje pasado como parámetro, la forma del array se conserva.

[152]:
```
arr = np.arange(1, 13)
# reshape
arr_2d_reshape = arr.reshape((3, 4))
print("Array con la nueva forma:\n", arr_2d_reshape)
```

```
Array con la nueva forma:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

[153]:
```
print(arr_2d_reshape)
# flat
arr_2d_flat = arr_2d_reshape.flat[9]
print("Elemento retornado por su índice después de redimensionar el array a 1-D:
 ↪\n", arr_2d_flat)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Elemento retornado por su índice después de redimensionar el array a 1-D:
 10
```

[154]:
```
print(arr_2d_reshape)
# flatten
arr_2d_flatten = arr_2d_reshape.flatten()
print("Array redimensionado a una dimensión:\n", arr_2d_flatten)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Array redimensionado a una dimensión:
 [ 1  2  3  4  5  6  7  8  9 10 11 12]
```

[155]:
```
print(arr_2d_reshape)
# ravel
arr_2d_ravel = arr_2d_reshape.ravel()
```

```
print("Array redimensionado a una dimensión:\n", arr_2d_ravel)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Array redimensionado a una dimensión:
 [ 1  2  3  4  5  6  7  8  9 10 11 12]
```

[157]:
```
# transpose
print("Array original:\n", arr_2d_reshape)
print("Transporner los ejes:\n", arr_2d_reshape.transpose())
```

```
Array original:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Transporner los ejes:
 [[ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]
 [ 4  8 12]]
```

[159]:
```
print(arr)
# resize
arr_resize = np.resize(arr, (4, 5))
print("Array redimensionado:\n", arr_resize)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12]
Array redimensionado:
 [[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12  1  2  3]
 [ 4  5  6  7  8]]
```

[160]:
```
print(arr_2d_reshape)
# resize
np.resize(arr_2d_reshape, (4, 5))
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

[160]:
```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12,  1,  2,  3],
       [ 4,  5,  6,  7,  8]])
```

[46]:
```
# flip
arr_filp = np.flip(arr)
```

```
print("Array 1-D en reversa:\n", arr_filp)
```

```
Array 1-D en reversa:
 [12 11 10  9  8  7  6  5  4  3  2  1]
```

[47]:
```
arr_2d_reshape_flip = np.flip(arr_2d_reshape)
print("Array 2-D en reversa:\n", arr_2d_reshape_flip)
```

```
Array 2-D en reversa:
 [[12 11 10  9]
 [ 8  7  6  5]
 [ 4  3  2  1]]
```

[48]:
```
arr_2d_reshape_flip_ax0 = np.flip(arr_2d_reshape, axis=0)
print("Array 2-D en reversa en filas:\n", arr_2d_reshape_flip_ax0)
```

```
Array 2-D en reversa en filas:
 [[ 9 10 11 12]
 [ 5  6  7  8]
 [ 1  2  3  4]]
```

[49]:
```
arr_2d_reshape_flip_ax1 = np.flip(arr_2d_reshape, axis=1)
print("Array 2-D en reversa en columnas:\n", arr_2d_reshape_flip_ax1)
```

```
Array 2-D en reversa en columnas:
 [[ 4  3  2  1]
 [ 8  7  6  5]
 [12 11 10  9]]
```

1.6 Insertar, eliminar y ordenar elementos

Durante el trabajo diario será usual que se necesite insertar, eliminar u ordenar los elementos en un array, para la primer tarea, NumPy ofrece las funciones de **append()**, **concatenate()** e **insert()** y para eliminar se tiene la función **delete()**, para la tercer tarea se tienen varias funciones con diferentes objetivos, pero en esta sección, solo se hará uso de la función **sort()**, si se desea conocer más sobre las otras funciones puede visitar la documentación oficial.

[50]:
```
arr_1d = np.arange(10)
print(arr_1d)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

La función **append** retorna un nuevo array con los elementos del array a insertar agregados al final. Es importante señalar que la dimensión del array que se desea insertar sea la misma que el primero, de lo contrario mostrará un error. Asimismo, se puede se puede realizar la inserción por eje, pasando como parámetro **axis** igual a 0 o 1.

[51]:
```
arr_ins = np.array([10, 11, 12])
arr_app_1d = np.append(arr_1d, arr_ins)
print(arr_app_1d)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12]
```

[52]:
```
# array de 2-D
arr_app_2d = arr_1d.reshape(2, 5)
print(arr_app_2d)
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

[53]:
```
# append en array de 2-D por filas
arr_app_2d_ax0 = np.append(arr_app_2d, [[10, 11, 12, 13, 14]], axis=0)
print("Array con inserción por filas:\n", arr_app_2d_ax0)
```

```
Array con inserción por filas:
 [[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

[54]:
```
# append en array de 2-D por columnas
arr_ins_ax1 = np.array([[98], [99]])
arr_app_2d_ax1 = np.append(arr_app_2d, arr_ins_ax1, axis=1)
print("Array con inserción por columnas:\n", arr_app_2d_ax1)
```

```
Array con inserción por columnas:
 [[ 0  1  2  3  4 98]
 [ 5  6  7  8  9 99]]
```

La función **concatenate** funciona similar a la función **append**, la diferencia es que los arrays a concatenar se pasan como parámetro en forma de tupla, asimismo, existe la posibilidad de señalar la inserción por fila o columna, de la misma manera hay que tener en cuenta que los objetos tengan la misma dimensión.

[55]:
```
a_concat = np.array([1, 2, 3, 4, 5])
b_concat = np.array([6, 7, 8, 9, 10])
print("Array de 1-D concatenado:\n", np.concatenate((a_concat, b_concat)))
```

```
Array de 1-D concatenado:
 [ 1  2  3  4  5  6  7  8  9 10]
```

[56]:
```
a_concat_ax0 = np.array([[0, 1], [2, 3], [4, 5]])
print("Array 2-D con cuerpo de 3x2:\n", a_concat_ax0)
b_concat_ax0 = np.array([[6, 7]])
print("Array 2-D concatenado por filas:\n", np.concatenate((a_concat_ax0,␣
 ↪b_concat_ax0), axis=0))
```

```
Array 2-D con cuerpo de 3x2:
 [[0 1]
 [2 3]
 [4 5]]
Array 2-D concatenado por filas:
 [[0 1]
```

```
 [2 3]
 [4 5]
 [6 7]]
```

[57]:
```
a_concat_ax1 = np.array([[0, 1], [2, 3], [4, 5]])
print("Array 2-D con cuerpo de 3x2:\n", a_concat_ax1)
b_concat_ax1 = np.array([[97], [98], [99]])
print("Array 2-D concatenado por columnas:\n", np.concatenate((a_concat_ax1,␣
 ↪b_concat_ax1), axis=1))
```

```
Array 2-D con cuerpo de 3x2:
 [[0 1]
 [2 3]
 [4 5]]
Array 2-D concatenado por columnas:
 [[ 0  1 97]
 [ 2  3 98]
 [ 4  5 99]]
```

La función **insert** al igual que las otras funciones, se emplea para la inserción de elementos dentro de un array existente, pero estos se inserten específicando un índice, asimismo, se puede pasar el eje como parámetro.

[58]:
```
arr_1d
print("Array 1-D original:\n", arr_1d)
arr_1d_insert = np.insert(arr_1d, 0, [1000])
print("Inserción de un elemento con índice 0 y valor de 1000 en el array␣
 ↪original:\n", arr_1d_insert)
```

```
Array 1-D original:
 [0 1 2 3 4 5 6 7 8 9]
Inserción de un elemento con índice 0 y valor de 1000 en el array original:
 [1000    0    1    2    3    4    5    6    7    8    9]
```

La operación anterior se puede aplicar a un array de dos dimensiones, NumPy "aplana" el array e inserta el elemento en el índice específicado.

[59]:
```
a_2d = np.array([[0, 0], [1, 1], [2, 2]])
print("Array original 2-D:\n", a_2d)
print("Array 'aplanado' con el nuevo elemento en el índice señalado:\n", np.
 ↪insert(a_2d, 6, 3)) # índice 6 elemento número 3
```

```
Array original 2-D:
 [[0 0]
 [1 1]
 [2 2]]
Array 'aplanado' con el nuevo elemento en el índice señalado:
 [0 0 1 1 2 2 3]
```

En el siguiente ejemplo se inserta el número 3 en la columna con índice 1, el valor se repilca en cada fila (**axis=0**). Se puede pasar un array con valores únicos para cada fila, es importante considerar

que exista una correspondencia entre longitudes.

[60]:
```
print("Array original 2-D:\n", a_2d)
print("Inserción en la columna con índice 1 del número 3 para cada fila:\n", np.
 ↪insert(a_2d, 1, 3, axis=1))
```

```
Array original 2-D:
 [[0 0]
 [1 1]
 [2 2]]
Inserción en la columna con índice 1 del número 3 para cada fila:
 [[0 3 0]
 [1 3 1]
 [2 3 2]]
```

[61]:
```
print("Array original 2-D:\n", a_2d)
print("Inserción en la columna con índice 1 de un array de 3 elementos␣
 ↪distintos:\n", np.insert(a_2d, 1, [10, 100, 1000], axis=1))
```

```
Array original 2-D:
 [[0 0]
 [1 1]
 [2 2]]
Inserción en la columna con índice 1 de un array de 3 elementos distintos:
 [[   0   10    0]
 [   1  100    1]
 [   2 1000    2]]
```

[62]:
```
print("Array original 2-D:\n", a_2d)
print("Inserción en la fila con índice 3 de número 300 para cada columna:\n",␣
 ↪np.insert(a_2d, 3, 300, axis=0))
```

```
Array original 2-D:
 [[0 0]
 [1 1]
 [2 2]]
Inserción en la fila con índice 3 de número 300 para cada columna:
 [[  0   0]
 [  1   1]
 [  2   2]
 [300 300]]
```

[63]:
```
print("Array original 2-D:\n", a_2d)
print("Inserción en la fila con índice 1 de un array de 3 elementos distintos:
 ↪\n", np.insert(a_2d, 1, [[100, 200]], axis=0))
```

```
Array original 2-D:
 [[0 0]
 [1 1]
 [2 2]]
```

```
Inserción en la fila con índice 1 de un array de 3 elementos distintos:
 [[  0   0]
 [100 200]
 [  1   1]
 [  2   2]]
```

La función **delete** permite eliminar elementos de forma individual o por fila y/o columna, para esto, se debe pasar por parámetro a la función el array, el índice y el eje. En el siguiente código se crea un array de 1-D con la función **arange** de 12 elementos, posteriormente se rehace la forma a 3 filas y 4 columnas.

En el primer ejemplo se pasa el array y una lista con los índices de los elementos a eliminar, observe que no se pasa el eje, esto significa que la función "aplana" el array o lo reconfigura a 1-D y elimina el primer y último elemento.

En los ejemplos 2 y 3, se elimnan filas y columnas respectivamente, en el primero se elimina la fila con índice 1 y en el segundo la columna con índice 2.

[64]:
```
arr_del = np.arange(12) + 1
arr_del_2d = arr_del.reshape((3, 4))
arr_del_2d
```

[64]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

[65]:
```
print("Se eliminan el primer y último elemento:\n", np.delete(arr_del_2d, [0,␣
↪11], None))
```

```
Se eliminan el primer y último elemento:
 [ 2  3  4  5  6  7  8  9 10 11]
```

[66]:
```
print("Se elimina la fila con índice 1:\n", np.delete(arr_del_2d, 1, axis=0))
```

```
Se elimina la fila con índice 1:
 [[ 1  2  3  4]
 [ 9 10 11 12]]
```

[67]:
```
print("Se elimina la columna con índice 2:\n", np.delete(arr_del_2d, 2, axis=1))
```

```
Se elimina la columna con índice 2:
 [[ 1  2  4]
 [ 5  6  8]
 [ 9 10 12]]
```

La función **sort** ordena de manera ascendente de acuerdo con valor de los elementos, por defecto, ordena por columnas, es decir, en **axis=1**, NumPy toma cada valor de la fila y los ordena de forma ascendente de acuerdo con el índice de cada columna (0, 1,…,n), caso contrario, con **axis=0**, ordena los elementos por su valor conforme al valor ascendente del índice de cada fila (0, 1,…,n), asimsimo, se puede no asignar un índice (**None**) lo que implica que NumPy "aplana" el array y ordena los elementos de manera ascendente.

[68]:
```
arr_sort = np.array([[2, 1, 3], [8, 9, 7], [6, 4, 5]])
arr_sort
```

[68]:
```
array([[2, 1, 3],
       [8, 9, 7],
       [6, 4, 5]])
```

[69]:
```
np.sort(arr_sort, axis=1)
```

[69]:
```
array([[1, 2, 3],
       [7, 8, 9],
       [4, 5, 6]])
```

[70]:
```
np.sort(arr_sort, axis=0)
```

[70]:
```
array([[2, 1, 3],
       [6, 4, 5],
       [8, 9, 7]])
```

[71]:
```
np.sort(arr_sort, None)
```

[71]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

1.7 Unir y dividir

Existe varias funciones en NumPy para unir arrays, una de ellas es **concatenate()** que ya se vió en la sección anterior, en este apartado se verán otras dos funciones, sin embargo existen más, pero solo nos concentraremos en **hstack()** y **vstack()** por ser de las más populares; en lo que respecta a la división, las funciones que se ejemplificaran son **split()**, **hsplit()** y **vsplit()**.

La función **hstack()** realiza la unión por el eje de las columnas (**axis=1**), es decir, concatena de forma horizontal.

[72]:
```
arr_hst_1 = np.array([[1, 2], [3, 4]])
arr_hst_2 = np.array([[5, 6], [7, 8]])
print(arr_hst_1,"\n")
print(arr_hst_2,"\n")
print("Array unido de manera horizontal:\n", np.hstack((arr_hst_1, arr_hst_2)))
```

```
[[1 2]
 [3 4]]

[[5 6]
 [7 8]]

Array unido de manera horizontal:
 [[1 2 5 6]
 [3 4 7 8]]
```

La función **vstack()** realiza la unión por el eje de las filas (**axis=0**), es decir, concatena de forma vertical.

[73]:
```
arr_vst_1 = np.array([[1, 2], [3, 4]])
arr_vst_2 = np.array([[5, 6], [7, 8]])
print(arr_vst_1,"\n")
print(arr_vst_2,"\n")
print("Array unido de manera vertical:\n", np.vstack((arr_vst_1, arr_vst_2)))
```

```
[[1 2]
 [3 4]]

[[5 6]
 [7 8]]

Array unido de manera vertical:
 [[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

La función **split()** divide un array en varios sub-arrays a lo largo de un eje, se puede pasar como parámetro un número entero que es el total de sub-arrays, se debe tener en cuenta que el entero debe ser un divisor del número total de elementos, de lo contario NumPy envía un error, asimismo, se puede pasar la división por medio de una lista de índices, el número de índice índica a partir de que elemento inicia la división.

[74]:
```
arr_split_1d = np.arange(9)
print(arr_split_1d)
print("\nDividir el array en tres sub-arrays:\n", np.split(arr_split_1d, 3))
```

```
[0 1 2 3 4 5 6 7 8]

Dividir el array en tres sub-arrays:
 [array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]
```

[75]:
```
a_split_2d_ax1 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(a_split_2d_ax1)
print("\nDivide el array 2-D en dos sub-arrays por el eje de las columnas:\n",␣
  ↪np.split(a_split_2d_ax1, 2, axis=1))
```

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

Divide el array 2-D en dos sub-arrays por el eje de las columnas:
 [array([[1, 2],
       [5, 6]]), array([[3, 4],
       [7, 8]])]
```

[76]:
```
print(a_split_2d_ax1)
print("\nDivide el array 2-D a partir de la columna con índice 2:\n", np.
 ↪split(a_split_2d_ax1, [2], axis=1))
```

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

Divide el array 2-D a partir de la columna con índice 2:
 [array([[1, 2],
        [5, 6]]), array([[3, 4],
        [7, 8]])]
```

[77]:
```
a_split_2d_ax0 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(a_split_2d_ax0)
print("\nDivide el array 2-D en dos sub-arrays por el eje de las filas:\n", np.
 ↪split(a_split_2d_ax0, 2, axis=0))
```

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

Divide el array 2-D en dos sub-arrays por el eje de las filas:
 [array([[1, 2, 3, 4]]), array([[5, 6, 7, 8]])]
```

[78]:
```
print(a_split_2d_ax0)
print("\nDivide el array 2-D a partir de la fila con índice 1:\n", np.
 ↪split(a_split_2d_ax0, [1], axis=0))
```

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

Divide el array 2-D a partir de la fila con índice 1:
 [array([[1, 2, 3, 4]]), array([[5, 6, 7, 8]])]
```

La función **hsplit()** es equivalente a la función **split()** con eje de columnas, es decir, con parámetro de **axis=1**.

[79]:
```
arr_hsplit_2d = np.array([(0, 1, 2, 3), (4, 5, 6, 7), (8, 9, 10, 11), (12, 13,␣
 ↪14, 15)])
```

[80]:
```
print(arr_hsplit_2d)
arr_hspl_1, arr_hspl_2 = np.hsplit(arr_hsplit_2d, 2)
print("\nDivide el array en dos sub-arrays por el eje de las columnas:\n",␣
 ↪arr_hspl_1, "\n\n", arr_hspl_2)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]

Divide el array en dos sub-arrays por el eje de las columnas:
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]

 [[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

La función **vsplit()** es equivalente a la función **split()** con eje de las filas, es decir, con parámetro de **axis=0**.

[81]:
```
arr_vsplit_2d = np.array([(0, 1, 2, 3), (4, 5, 6, 7), (8, 9, 10, 11), (12, 13,␣
 ↪14, 15)])
```

[82]:
```
print(arr_vsplit_2d)
arr_vspl_1, arr_vspl_2 = np.vsplit(arr_vsplit_2d, 2)
print("\nDivide el array en dos sub-arrays por el eje de las filas:\n",␣
 ↪arr_vspl_1, "\n\n", arr_vspl_2)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]

Divide el array en dos sub-arrays por el eje de las filas:
 [[0 1 2 3]
 [4 5 6 7]]

 [[ 8  9 10 11]
 [12 13 14 15]]
```

1.8 Funciones estadísticas

1.8.1 Mínimos y máximos

Para explorar y conocer la naturaleza de los datos lo más común es apoyarse de algunas medidas estadísticas, como son la media, la mediana, la desviación estándar, etcétera. Numpy ofrece funcionas para conocer estos estadísticos, a continuación, se ejemplifican algunos de los más usuales para un análisis exploratorio de los datos (EAD por sus siglas en inglés).

[83]:
```
arr_stats = np.array([[30, 45, 10], [15, 90, 40], [20, 65, 25]])
arr_stats
```

[83]:
```
array([[30, 45, 10],
       [15, 90, 40],
       [20, 65, 25]])
```

En primer lugar, se presentán las funciones de valores mínimos y máximos, para ello se puede hacer

uso de las funciones: **amin()** y **amax()**, o bien, las propias del objeto array creado: **min()** y **max()**, para las primeras es necesario pasar como argumento el array al cual se le quiere obtener la información.

[84]:
```
# uso de amin()
arr_st_amin = np.amin(arr_stats)
print("El valor mínimo de todo el array es:\n", arr_st_amin)
```

```
El valor mínimo de todo el array es:
 10
```

[85]:
```
# uso de min()
arr_st_min = arr_stats.min()
print("El valor mínimo de todo el array es:\n", arr_st_min)
```

```
El valor mínimo de todo el array es:
 10
```

[86]:
```
# uso de amax()
arr_st_amax = np.amax(arr_stats)
print("El valor máximo de todo el array es:\n", arr_st_amax)
```

```
El valor máximo de todo el array es:
 90
```

[87]:
```
# uso de max()
arr_st_max = arr_stats.max()
print("El valor máximo de todo el array es:\n", arr_st_max)
```

```
El valor máximo de todo el array es:
 90
```

Para obtener el valor mínimo y máximo para cada fila o columna se pasa como parámetro el eje o **axis** igual a 0 para filas y 1 para columnas como ya se ha visto.

[88]:
```
print(arr_stats)
arr_st_amin_ax0 = np.amin(arr_stats, axis=0)
print("Array con los elementos de valores mínimos para cada fila:\n",␣
 ↪arr_st_amin_ax0)
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
Array con los elementos de valores mínimos para cada fila:
 [15 45 10]
```

[89]:
```
print(arr_stats)
arr_st_min_ax1 = arr_stats.min(axis=1)
print("Array con los elementos de valores mínimos para cada columna:\n",␣
 ↪arr_st_min_ax1)
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
Array con los elementos de valores mínimos para cada columna:
 [10 15 20]
```

[90]:
```
print(arr_stats)
arr_st_amax_ax0 = np.amax(arr_stats, axis=0)
print("Array con los elementos de valores máximos para cada fila:\n",␣
 ↪arr_st_amax_ax0)
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
Array con los elementos de valores máximos para cada fila:
 [30 90 40]
```

[91]:
```
print(arr_stats)
arr_st_max_ax1 = arr_stats.max(axis=1)
print("Array con los elementos de valores máximos para cada columna:\n",␣
 ↪arr_st_max_ax1)
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
Array con los elementos de valores máximos para cada columna:
 [45 90 65]
```

1.8.2 Media y media ponderada

Al igual que con las funciones anteriores, se puede obtener el valor medio de todo el array, o bien, obtener un array con los valores medios ya sea para cada fila o columna esto con la función **mean()**, asimismo, se puede obtener una media ponderada al pasar como parámetro un array con pesos o la ponderación asociada a cada elemento.

[92]:
```
print(arr_stats)
print("El valor de medio de todo el array:\n", arr_stats.mean())
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
El valor de medio de todo el array:
 37.77777777777778
```

[93]:
```
print(arr_stats)
print("Los valores medios del array en cada fila son:\n", arr_stats.
 ↪mean(axis=0))
```

```
[[30 45 10]
 [15 90 40]
```

```
 [20 65 25]]
Los valores medios del array en cada fila son:
 [21.66666667 66.66666667 25.        ]
```

[94]:
```
print(arr_stats)
print("Los valores medios del array en cada columna son:\n", arr_stats.
  ↪mean(axis=1))
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
Los valores medios del array en cada columna son:
 [28.33333333 48.33333333 36.66666667]
```

Para calcular una media ponderada NumPy posee la función **average()**, la cual recibe como parámetro un array de pesos (**weights=**), este puede ser un array con pesos asociados a los valores del array, o bien, un array de 1-D con la misma longitud del eje dado, o bien, debe tener la misma forma del array. Si la función **average** no recibe el parámetro de pesos o ponderación retorna el mismo resultado que la función **mean()**.

[95]:
```
arr_avg_1d = np.arange(10, 15)
pesos = np.array([0.5, 0.75, 2.25, 1.5, 3])
print("Promedio sin ponderación:\n", np.average(arr_avg_1d))
print("Promedio ponderado:\n", np.average(arr_avg_1d, weights=pesos))
```

```
Promedio sin ponderación:
 12.0
Promedio ponderado:
 12.71875
```

[96]:
```
arr_avg_2d = np.arange(1, 13).reshape(3, 4)
print(arr_avg_2d)
w_ax0 = np.array([2, 0.5, 5])
print("Media ponderada para filas:\n", np.average(arr_avg_2d, weights=w_ax0,␣
  ↪axis=0))
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Media ponderada para filas:
 [6.6 7.6 8.6 9.6]
```

[164]:
```
print(arr_avg_2d)
w_ax1 = np.array([2, 0.5, 5, 0.25])
print("Media ponderada para columnas:\n", np.average(arr_avg_2d, weights=w_ax1,␣
  ↪axis=1))
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

```
Media ponderada para columnas:
 [ 2.4516129  6.4516129 10.4516129]
```

1.8.3 Mediana

NumPy tiene la función **median()** para devolver el valor de la mediana, asimismo, puede obtenerse para un eje en particular.

[98]:
```
print(arr_stats)
arr_median = np.median(arr_stats)
print("Valor de la mediana de todo el array:\n", arr_median)
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
Valor de la mediana de todo el array:
 30.0
```

[99]:
```
print(arr_stats)
print("Valores de la mediana por filas:\n", np.median(arr_stats, axis=0))
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
Valores de la mediana por filas:
 [20. 65. 25.]
```

[100]:
```
print(arr_stats)
print("Valores de la mediana por columnas:\n", np.median(arr_stats, axis=1))
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
Valores de la mediana por columnas:
 [30. 40. 25.]
```

1.8.4 Desviación estándar

NumPy tiene la función **std()** para devolver la desviación estándar, asimismo, puede obtenerse para un eje en particular.

[101]:
```
print(arr_stats)
arr_desv_estan = np.std(arr_stats)
print("Valor de la desviación estándar de todo el array:\n", arr_desv_estan)
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
Valor de la desviación estándar de todo el array:
 24.393887111222387
```

[102]:
```
print(arr_stats)
print("Valores de la desviación estándar por filas:\n", np.std(arr_stats,␣
 ↪axis=0))
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
Valores de la desviación estándar por filas:
 [ 6.23609564 18.40893503 12.24744871]
```

[103]:
```
print(arr_stats)
print("Valores de la desviación estándar por columnas:\n", np.std(arr_stats,␣
 ↪axis=1))
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
Valores de la desviación estándar por columnas:
 [14.33720878 31.18047822 20.13840996]
```

1.8.5 Varianza

NumPy tiene la función **var()** para devolver la varianza, asimismo, puede obtenerse para un eje en particular.

[104]:
```
print(arr_stats)
arr_var = np.var(arr_stats)
print("Valor de la varianza de todo el array:\n", arr_var)
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
Valor de la varianza de todo el array:
 595.0617283950618
```

[105]:
```
print(arr_stats)
print("Valores de la varianza por filas:\n", np.var(arr_stats, axis=0))
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
Valores de la varianza por filas:
 [ 38.88888889 338.88888889 150.        ]
```

[106]:
```
print(arr_stats)
print("Valores de la varianza por columnas:\n", np.var(arr_stats, axis=1))
```

```
[[30 45 10]
 [15 90 40]
 [20 65 25]]
```

```
Valores de la varianza por columnas:
 [205.55555556 972.22222222 405.55555556]
```

1.8.6 Percentiles

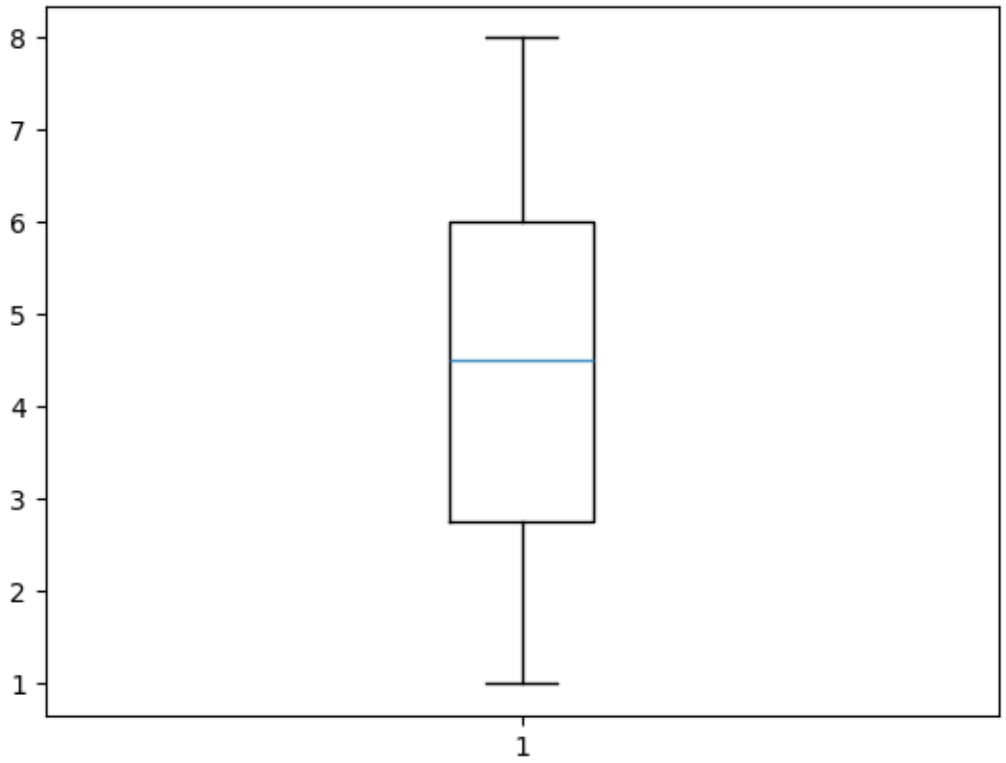
Para encontrar los valores percentil 'n' del array NumPy provee la función **percentile()**, como ya se ha visto, se puede obtener el valor o los valores para un eje específicado, en caso de no proporcionar el parámetro del eje, la función proporcionará un valor escalar. Los parámetros principales de la función son: el array, el valor del percentile 'n' y el eje, para un aprendizaje más profundo puede visitar la documentación oficial o hacer uso de la función **help()**.

[107]:
```
arr_perc = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [2, 3, 5, 6]])
print(arr_perc)
print("Percentil 25 del array sin asignar eje:\n", np.percentile(arr_perc, 25))
print("Percentil 50 del array sin asignar eje:\n", np.percentile(arr_perc, 50))
print("Percentil 75 del array sin asignar eje:\n", np.percentile(arr_perc, 75))
```

```
[[1 2 3 4]
 [5 6 7 8]
 [2 3 5 6]]
Percentil 25 del array sin asignar eje:
 2.75
Percentil 50 del array sin asignar eje:
 4.5
Percentil 75 del array sin asignar eje:
 6.0
```

A modo de visualización de los resultados obtenidos, se programará el siguiente boxplot del array con una transformación a 1-D. Para esto, de **matplotlib** se importa el módulo **pyplot** usada para gráficos en Python, cabe mencionar que el tema de los gráficos no es objetivo de este documento, solo es un apoyo. Asimismo, se obtendrán los mismos percentiles para los ejes de filas y columnas con los correspondientes gráficos de caja.

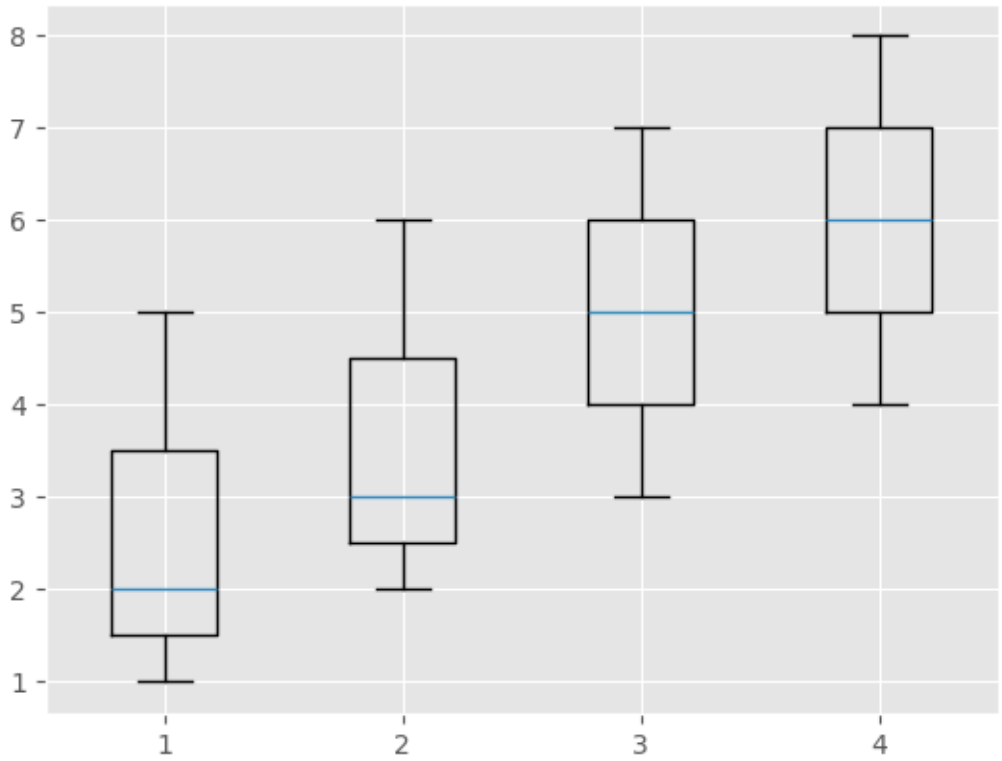
[165]:
```
from matplotlib import pyplot as plt
plt.boxplot(arr_perc.flatten())
plt.style.use('ggplot')
plt.show()
```



[109]:
```
print(arr_perc)
print("Percentil 25 del array para el eje de las filas:\n", np.
 ↪percentile(arr_perc, 25, axis=0))
print("Percentil 50 del array para el eje de las filas:\n", np.
 ↪percentile(arr_perc, 50, axis=0))
print("Percentil 75 del array para el eje de las filas:\n", np.
 ↪percentile(arr_perc, 75, axis=0))
```

```
[[1 2 3 4]
 [5 6 7 8]
 [2 3 5 6]]
Percentil 25 del array para el eje de las filas:
 [1.5 2.5 4.  5. ]
Percentil 50 del array para el eje de las filas:
 [2. 3. 5. 6.]
Percentil 75 del array para el eje de las filas:
 [3.5 4.5 6.  7. ]
```

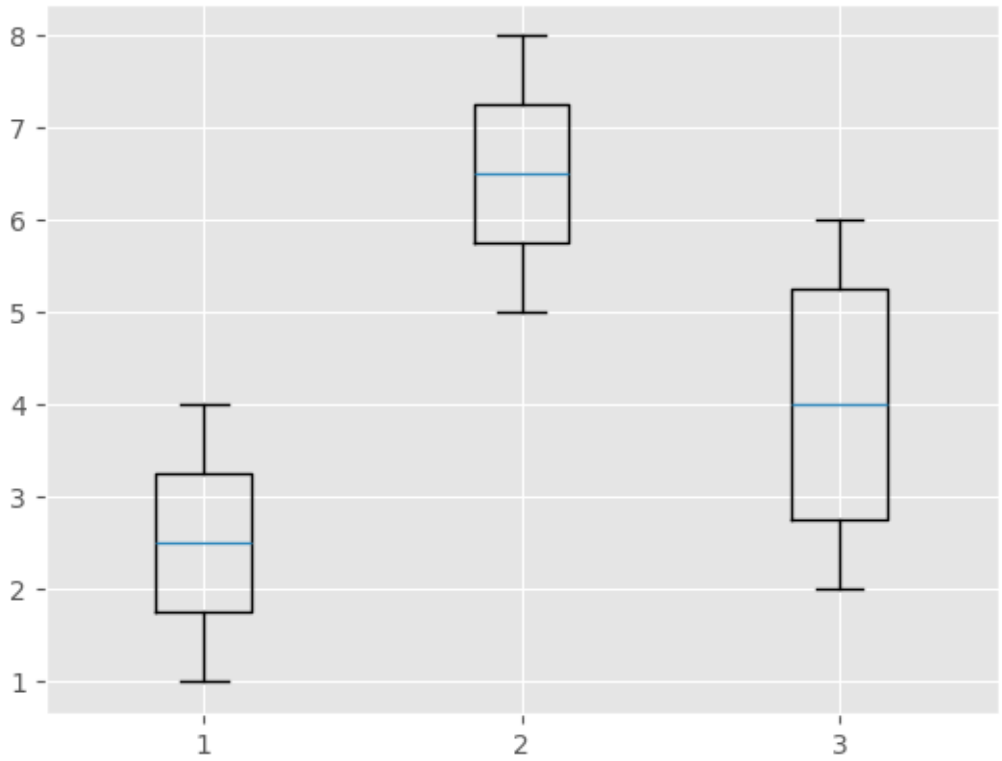
[166]:
```
plt.boxplot(arr_perc[:])
plt.style.use('ggplot')
plt.show()
```



[111]:
```
print(arr_perc)
print("Percentil 25 del array para el eje de las columnas:\n", np.
 ↪percentile(arr_perc, 25, axis=1))
print("Percentil 50 del array para el eje de las columnas:\n", np.
 ↪percentile(arr_perc, 50, axis=1))
print("Percentil 75 del array para el eje de las columnas:\n", np.
 ↪percentile(arr_perc, 75, axis=1))
```

```
[[1 2 3 4]
 [5 6 7 8]
 [2 3 5 6]]
Percentil 25 del array para el eje de las columnas:
 [1.75 5.75 2.75]
Percentil 50 del array para el eje de las columnas:
 [2.5 6.5 4. ]
Percentil 75 del array para el eje de las columnas:
 [3.25 7.25 5.25]
```

[167]:
```
plt.boxplot(arr_perc[:].T) # (T) transponer los ejes
plt.style.use('ggplot')
plt.show()
```



1.9 Operaciones númericas básicas

NumPy ofrece una gran cantidad de funciones matemáticas organizadas por categorías, algunas de estas son: funciones trigonométricas, hiperbólicas, redondeo, aritméticas, exponentes y logaritmos, entre muchas otras más, en este apartado solo se muestran algunas básicas de tipo aritmético y de exponente, si se desea conocer más se puede visitar la documentación oficial.

Las primeras funciones aritméticas que se ejemplifican son las más básicas, sumar, restar, multiplicar y dividir, las operaciones se realiza elemento a elemento; en caso de que no tengan la misma forma, sí es importante que coincidan en la forma del array de salida.

[113]:
```
arr_op_bas_1 = np.arange(1, 13).reshape(3, 4)
arr_op_bas_2 = np.arange(1, 5)
```

[114]:
```
# suma
print(arr_op_bas_1, "\n")
print(arr_op_bas_2, "\n")
print("Resultado de la suma:\n", np.add(arr_op_bas_1, arr_op_bas_2))
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

```
[1 2 3 4]

Resultado de la suma:
 [[ 2  4  6  8]
 [ 6  8 10 12]
 [10 12 14 16]]
```

[115]:
```
# resta
print(arr_op_bas_1, "\n")
print(arr_op_bas_2, "\n")
print("Resultado de la resta:\n", np.subtract(arr_op_bas_1, arr_op_bas_2))
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

[1 2 3 4]

Resultado de la resta:
 [[0 0 0 0]
 [4 4 4 4]
 [8 8 8 8]]
```

[116]:
```
# multiplicación
print(arr_op_bas_1, "\n")
print(arr_op_bas_2, "\n")
print("Resultado de la multiplicación:\n", np.multiply(arr_op_bas_1,␣
 ↪arr_op_bas_2))
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

[1 2 3 4]

Resultado de la multiplicación:
 [[ 1  4  9 16]
 [ 5 12 21 32]
 [ 9 20 33 48]]
```

[117]:
```
# división
print(arr_op_bas_1, "\n")
print(arr_op_bas_2, "\n")
print("Resultado de la división:\n", np.divide(arr_op_bas_1, arr_op_bas_2))
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

```
[1 2 3 4]

Resultado de la división:
 [[1.         1.         1.         1.        ]
 [5.         3.         2.33333333 2.        ]
 [9.         5.         3.66666667 3.        ]]
```

La función **power()** retorna el primer array elevado al segundo a un valor escalar.

[118]:
```
print(arr_op_bas_1)
print("Array elevado a una potencia de 2:\n", np.power(arr_op_bas_1, 2))
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Array elevado a una potencia de 2:
 [[  1   4   9  16]
 [ 25  36  49  64]
 [ 81 100 121 144]]
```

[119]:
```
print(arr_op_bas_2)
print("Array elevado a una potencia de 3:\n", np.power(arr_op_bas_2, 3))
```

```
[1 2 3 4]
Array elevado a una potencia de 3:
 [ 1  8 27 64]
```

[120]:
```
print("Array elevado a una potencia de otro array:\n", np.power(arr_op_bas_1,␣
 ↪arr_op_bas_2))
```

```
Array elevado a una potencia de otro array:
 [[    1     4    27   256]
 [    5    36   343  4096]
 [    9   100  1331 20736]]
```

La función **mod()** retorna el residuo o resto de una división entre arrays por elemento, esta función es equivalente a la función **remainder()**.

[121]:
```
arr_op_bas_3 = np.array([[13, 15, 16, 14], [20, 45, 58, 38], [8, 10, 19, 12]])
print(arr_op_bas_3, "\n")
arr_op_bas_4 = np.array([2, 5, 4, 3])
print(arr_op_bas_4)
```

```
[[13 15 16 14]
 [20 45 58 38]
 [ 8 10 19 12]]

[2 5 4 3]
```

[122]:
```
print("Array con los residuos:\n", np.mod(arr_op_bas_3, arr_op_bas_4))
```

```
Array con los residuos:
 [[1 0 0 2]
 [0 0 2 2]
 [0 0 3 0]]
```

[123]:
```
print("Array con los residuos:\n", np.remainder(arr_op_bas_3, arr_op_bas_4))
```

```
Array con los residuos:
 [[1 0 0 2]
 [0 0 2 2]
 [0 0 3 0]]
```

La función **sqrt()** devuelve la raíz cuadrada no negativa por elementos.

[124]:
```
print(arr_op_bas_3)
print("Raíz cuadrada no negativa de cada elemento:\n", np.sqrt(arr_op_bas_3))
```

```
[[13 15 16 14]
 [20 45 58 38]
 [ 8 10 19 12]]
Raíz cuadrada no negativa de cada elemento:
 [[3.60555128 3.87298335 4.         3.74165739]
 [4.47213595 6.70820393 7.61577311 6.164414  ]
 [2.82842712 3.16227766 4.35889894 3.46410162]]
```

1.10 Escritura en archivos

Para la escritura de datos en archivos NumPy proporciona diferentes formatos y funciones para esto, entre algunos formatos están los de tipo binario, texto, binarios sin formato, formato de cadena, de mapeo en memoria, entre otros, si desea más información puede dirigirse al siguiente enlace.

En este apartado sólo se ejemplificarán dos funciones: **save()** y **savetxt()**, de formatos binario y texto respectivamente.

La función **save()** como se mencionó guarda un array en un archivo binario con formato **.npy** que es una extensión de un formato interno de NumPy, los parámetros indispensables son la cadena de texto con la ubicación donde se guardará, así como nombre y extensión del archivo, el siguiente es el nombre del objeto o array que se va a escribir.

[125]:
```
arr_save = np.arange(24).reshape(6, 4)
arr_save
```

[125]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

[126]:
```
np.save("../data/arr_save.npy", arr_save)
```

La función **savetxt()** escribe el contenico de un array en un archivo de texto plano, entre los parámetros más importantes a pasar son la cadena con la ubicación dónde se va a guardar y el nombre y extensión del archivo, el array u objeto a escribir, el delimitador, existen otros parámetros que se pueden revisar en la siguiente liga.

[127]:
```
arr_savetxt = np.arange(0, 40, 2).reshape(4, 5)
arr_savetxt
```

[127]:
```
array([[ 0,  2,  4,  6,  8],
       [10, 12, 14, 16, 18],
       [20, 22, 24, 26, 28],
       [30, 32, 34, 36, 38]])
```

[128]:
```
np.savetxt('../data/arr_savetxt.csv', arr_savetxt, delimiter=',')
```

1.11 Lectura de archivos

De igual manera, para la lectura de archivos NumPy ofrece varias opciones, sin embargo, en este apartado nos concentraremos en las funciones para leer los archivos anteriormente guardados, para ello se hará uso de las funciones **load()** para archivos con extensión **.npy** y **.npz**, y la función **loadtxt()** para la lectura de archivos de texto plano, para esta última destaca el parámetro **delimiter** que recibe un string con el delimitador entre elementos o datos. Para este último ejemplo se hace uso de la función **astype()** para asignar el tipo de dato, ya que por defecto NumPy lo asigna en tipo decimal o float.

[129]:
```
arr_load = np.load("../data/arr_save.npy")
arr_load
```

[129]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

[130]:
```
arr_loadtxt = np.loadtxt("../data/arr_savetxt.csv", delimiter=',').astype(np.
↪int64)
arr_loadtxt
```

[130]:
```
array([[ 0,  2,  4,  6,  8],
       [10, 12, 14, 16, 18],
       [20, 22, 24, 26, 28],
       [30, 32, 34, 36, 38]], dtype=int64)
```

Para obtener el archivo puede visitar la siguiente liga:

https://github.com/LandinGabriel13/Introduccion_a_NumPy