14 - Numpy

December 1, 2022

1 Numpy

Hasta el momento, solo se ha trabajado con Python en temas genéricos, es decir, hemos utilizado módulos y funciones en programas que no pertenecen a un área en específico. Ahora, en este capítulo, se introducirá un módulo desarrollado, principalmente, para trabajar con matrices y vectores. Este módulo es llamado Numpy.

Numpy es un módulo de código abierto y gratuito que no forma parte de la biblioteca estándar de Python, por lo que requiere de una **instalación adicional**.

Debido a las características que ofrece, a menudo Numpy es utilizado en conjunto con otros módulos como lo son `mathplotlib` o `Pandas`. Lo anterior, para realizar operaciones que se trabajan en ambientes de programación científica como `Matlab` y `Octave`.

1.1 Característica Generales

Como se mencionó previamente, Numpy es un `módulo` que requiere instalación para que esté disponible para su uso en Python.

Una vez instalado, para poder utilizarlo, es necesario realizar la importación de funciones (como se vio en clases anteriores).

Dicha importación se realizará de la siguiente manera:

[ ]:
```
import numpy as np

# De esta manera el módulo numpy será llamado con las siglas np
# Cada vez que se requiera llamar una función de dicho módulo, se antepondrá las␣
 ↪letras np.<función> a utilizar
# Nótese que esa es la forma estándar de importar este módulo, aunque las otras␣
 ↪alternativas
# siguen siendo válidas
```

El uso de Numpy permite trabajar de manera directa con dos nuevos tipos de datos que representan a vectores (llamados arreglos) y a matrices. Dichos tipos de datos son `array` y `matrix` respectivamente.

1.2 Arreglos (`array`)

Los arreglos son representaciones de Python para definir vectores, los cuales pueden ser de unidimensionales o multidimensionales (arreglos de arreglos).

La ventaja de poder utilizar arreglos es que permite trabajar de manera vectorial, lo cual hace que no sea necesario el uso de ciclos para realizar operaciones matemáticas a cada elemento.

Para definir los arreglos, se utiliza la función `array`, la cual recibe como parámetro de entrada una lista.

[3]:
```
import numpy as np
# Ejemplo_1: Se define un vector (arreglo) y posterior a ello, realiza la␣
 ↪multiplicación de cada elemento por un número
arreglo_1 = np.array([2, 4, 6, 8, 10])
print(arreglo_1)

vector_1 = arreglo_1*3
print(vector_1)
```

```
[ 2  4  6  8 10]
[ 6 12 18 24 30]
```

Para definir un vector multidimensional, solo hace falta ingresar listas de listas como parámetro de entrada para la función `array`

[2]:
```
import numpy as np
# Ejemplo_2: Se define un vector (arreglo) multidimensional
arreglo_2 = np.array([[2, 4], [8, 10], [0, 6]])
print(arreglo_2)
```

```
[[ 2  4]
 [ 8 10]
 [ 0  6]]
```

Como ahora se puede trabajar con vectores, se pueden realizar operaciones matemáticas de manera directa. En el siguiente ejemplo se ve el caso de la suma de vectores.

[5]:
```
import numpy as np
# Ejemplo_3: Se definen dos vectores (arreglo) y posterior a ello, se realiza la␣
 ↪suma de ambos
arreglo_A_3 = np.array([2, 4, 6, 1, 3 , 7])
arreglo_B_3 = np.array([1, 5, 2, 8, 4, 1])

suma = arreglo_A_3 + arreglo_B_3
print(suma)
```

```
[3 9 8 9 7 8]
```

De la misma forma en que `Numpy` permite realizar la suma y resta, existe al posibilidad de poder realizar la operación de multiplicación

[6]:
```
import numpy as np
# Ejemplo_4: Se definen dos vectores (arreglo) y posterior a ello, se realiza la␣
 ↪multiplicación de ambos
arreglo_A_4 = np.array([2, 4, 6, 1, 3, 7])
arreglo_B_4 = np.array([1, 5, 2, 8, 4, 1])

multiplicacion = arreglo_A_4 * arreglo_B_4
print(multiplicacion)
```

```
[ 2 20 12  8 12  7]
```

Esto es válido para todas las operaciones aritméticas básicas: exponenciación (**), multiplicaciones y divisiones (*, /, // y %), sumas, identidad, resta y cambio de signo (+, -).

1.3 Matrices (`matrix`)

Numpy no solo ofrece la posibilidad de generar vectores sino que también permite el trabajo con matrices. Para ello, se utiliza el tipo de dato `matrix`, el cual recibe como parámetro de entrada una lista de listas.

[ ]:
```
import numpy as np
# Ejemplo_5: Se define una matriz a partir de una lista de listas
matriz_5 = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(matriz_5)
```

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

Al igual que en vectores (`array`), Numpy ofrece la posibilidad de realizar sumas en matrices (`matrix`)

[ ]:
```
import numpy as np
# Ejemplo_6: Se definen dos matrices y posterior a ello, se realiza la suma de␣
 ↪ambas
matriz_A_6 = np.matrix([[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]])
matriz_B_6 = np.matrix([[0, 1, 2, 3, 11], [2, 1, 4, 5, 6]])

print("\n")

suma = matriz_A_6 + matriz_B_6
print(suma)
```

```
[[ 2  5  8 11 21]
 [ 3  4  9 12 15]]
```

Además de la suma y resta, Numpy permite realizar multiplicación de matrices considerando sus

implicancias (las columnas de la primera matriz debe ser igual a las filas de la segunda matriz).

[9]:
```
import numpy as np
# Ejemplo_7: Se definen dos matrices y posterior a ello, se realiza la␣
 ↪multiplicación de ambas
matriz_A_7 = np.matrix([[2, 0, 1], [3, 0, 0], [5, 1, 1]])
matriz_B_7 = np.matrix([[1, 0, 1], [1, 2, 1], [1, 1, 0]])

multiplicacion = matriz_A_7 * matriz_B_7
print(multiplicacion)
```

```
[[3 1 2]
 [3 0 3]
 [7 3 6]]
```

1.4 Operaciones en Numpy

Al incorporar Numpy en Python, se abre un nuevo horizonte en donde se pueden realizar diversas operaciones tanto para vectores como matrices, representadas por el tipo de dato `array` y `matrix` respectivamente. Entre dichas operaciones se pueden encontrar:

1. Indexación
2. Asignación
3. Cortar
4. Iteración
5. Modificar dimensiones

1.4.1 1. Indexación

Tal como se vio en el caso de listas (`list`) y strings (`str`), es posible acceder a distintas posiciones tanto de un vector (`array`) como a una matriz (`matrix`). Esto se realiza de la siguiente manera:

1.a Arreglo (Vector)

[8]:
```
import numpy as np
# Ejemplo_8: Se define un vector (arreglo) y posterior a ello, se muestra el␣
 ↪mismo vector,
# pero cada elemento aparece cuadriplicado
arreglo = np.array([2, 4, 6, 8, 10])
print(arreglo)

arreglo_8 = arreglo*4
print(arreglo_8)
```

```
[ 2  4  6  8 10]
[ 8 16 24 32 40]
```

1.b Matriz

[ ]:
```
import numpy as np
# Ejemplo_9: Se define una matriz y posterior a ello, se solicita que muestre␣
 ↪por pantalla un elemento
# de una posición determinada
matriz = np.matrix([[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]])
print(matriz)

elemento_9 = matriz[1, 3]
print(elemento_9)
```

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

En el **Ejemplo_9**, se creó una matriz en Python y se pidió mostrar por pantalla el elemento que se encuentra en la **fila posición 1**, **columna posición 3**.

A simple vista, la indexación de las matrices en Python es muy similar a la de las listas de listas, no obstante, tiene una gran diferencia: para acceder a una posición determinada en una lista de lista, se usa doble corchete (uno a continuación del otro, p.e. `lista[1][3]`), en cambio, en las matrices solo se usa un corchete.

A continuación se muestra un ejemplo de lo anterior:

[ ]:
```
import numpy as np
# Ejemplo_10: Se define una matriz y una lista de listas posterior a ello, se␣
 ↪accede a la misma posición
matriz = np.matrix([[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]])
print(matriz)

elemento_10_matriz = matriz[1, 3]
print(elemento_10_matriz)

lista_de_listas = [[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]]
print(lista_de_listas)

elemento_10_lista = lista_de_listas[1][3]
print(elemento_10_lista)
```

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

Nota Hay que señalar una diferencia entre estos mecanismos para un `array` multimensional: - Para un `array` **multidimensional**, ambas notaciones producen el mismo resultado. - Para una `matrix`, un solo índice entrega una *submatriz*, que es, a su vez, una matriz, por lo que solo se puede acceder a los elementos entregando fila y columna como índices. - Para una lista, el índice solo puede ser un número entero, por lo que la notación de separar por coma produce un error de

sintaxis (SyntaxError).

Ejemplo:

[11]:
```
import numpy as np
lista = [[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]]
arreglo = np.array(lista)
matriz = np.matrix(lista)

print("Array:")
# Un "subarray" es otro array
print(arreglo[0])
# Se puede acceder a una posición en un array
print(arreglo[0][0])
# O se puede acceder a una coordenada en el array más grande
print(arreglo[0, 0])

print("Matrix:")
# Una submatriz es otra matriz
print(matriz[0])
# Por lo tanto, esto genera *otra matriz*
print(matriz[0][0])
# Esta es la forma de acceder a un elemento
print(matriz[0, 0])

print("List:")
# Un elemento de una lista de listas es una lista
print(lista[0])
# Por lo que podemos acceder a un elemento de esta
print(lista[0][0])
# Pero esto generará un error de sintaxis:
print(lista[0, 0])
```

```
Array:
[ 2  4  6  8 10]
2
2
Matrix:
[[ 2  4  6  8 10]]
[[ 2  4  6  8 10]]
2
List:
[2, 4, 6, 8, 10]
2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
c:\Users\Usuario\OneDrive - usach.cl\coordinaciones\FPI-FCyP\apuntes_colab\14 -␣
 ↪Numpy.ipynb Celda 46 in <cell line: 28>()
```

```
    <a href='vscode-notebook-cell:/c%3A/Users/Usuario/OneDrive%20-%20usach.cl/
↪coordinaciones/FPI-FCyP/apuntes_colab/14%20-%20Numpy.ipynb#Y204sZmlsZQ%3D%3D?
↪line=25'>26</a> print(lista[0][0])
    <a href='vscode-notebook-cell:/c%3A/Users/Usuario/OneDrive%20-%20usach.cl/
↪coordinaciones/FPI-FCyP/apuntes_colab/14%20-%20Numpy.ipynb#Y204sZmlsZQ%3D%3D?
↪line=26'>27</a> # Pero esto generará un error de sintaxis:
---> <a href='vscode-notebook-cell:/c%3A/Users/Usuario/OneDrive%20-%20usach.cl/
↪coordinaciones/FPI-FCyP/apuntes_colab/14%20-%20Numpy.ipynb#Y204sZmlsZQ%3D%3D?
↪line=27'>28</a> print(lista[0, 0])

TypeError: list indices must be integers or slices, not tuple
```

1.4.2 2. Asignación

De igual manera como sucede en las listas (`list`), Numpy permite crear arreglos y matrices en donde se puede asignar valores determinados a posiciones en particular.

2.a Arreglo (Vector)

[12]:
```
import numpy as np
# Ejemplo_11: Se define un vector (arreglo) y posterior a ello, se asignará un
# nuevo valor a una posición determinada
arreglo_11 = np.array([2, 4, 6, 7, 10])
print(arreglo_11)

arreglo_11[3] = 8
print(arreglo_11)
```

```
[ 2  4  6  7 10]
[ 2  4  6  8 10]
```

2.b Matriz

[ ]:
```
import numpy as np
# Ejemplo_12: Se define una matriz y posterior a ello, se asignará un nuevo valor
# a una posición determinada
matriz_12 = np.matrix([[2, 4, 6, 8, 10], [1, 4, 5, 7, 9]])
print(matriz_12)

print("\n")
matriz_12[1, 1] = 3
print(matriz_12)
```

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


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

1.4.3 3. Cortar

Al igual que en las listas, se pueden cortar los arreglos y matrices para extraer parte de su información generando un sub-arreglos o sub-matrices. Para realizar la acción anterior se deben indicar los rangos de datos a copiar.

La forma de cortar un arreglo y una matriz es la siguiente:

3.a Arreglo (Vector)

[ ]:
```
import numpy as np
#Ejemplo_13: Se define un vector (arreglo) y posterior a ello, se realizará un␣
 ↪corte para crear un sub-arreglo
arreglo_13 = np.array([2, 4, 6, 7, 10])
print(arreglo_13)

print("\n")

corte_13 = arreglo_13[2:4]
print(corte_13)
```

```
[ 2  4  6  7 10]


[6 7]
```

3.b Matriz

[ ]:
```
import numpy as np
#Ejemplo_14: Se define una matriz y posterior a ello, se realizará un corte para␣
 ↪crear un sub-arreglo
matriz_14 = np.matrix([[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]])
print(matriz_14)

print("\n")

corte_14 = matriz_14[0:2, 2:3]
print(corte_14)
```

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


[[6]
 [5]]
```

Como se observa en el **ejemplo_14**, se solicita que muestre desde la fila 0 hasta la fila 2-1 (la fila en posició 0 y la fila posición 1). En el caso de la columna, se indica que se extraiga solo la columna en posición 2 de cada fila.

En caso de que se quiera acceder solo a la primera fila y acceder a las primeras 3 columnas, el ejemplo anterior quedaría de la siguiente manera:

[15]:
```
import numpy as np
# Ejemplo_15: Se define una matriz y posterior a ello, se realizará un corte␣
 ↪para crear un sub-arreglo
matriz_15 = np.matrix([[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]])
print(matriz_15)

print("\n")

corte_15 = matriz_15[0, 0:3]
print(corte_15)
```

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


[[2 4 6]]
```

1.4.4 4. Iteración

Al igual que lo visto en listas (`list`) y en strings (`str`), tanto en arreglos y matrices generadas por Numpy es posible acceder a cada posición y realizar una determinada acción. En otras palabras, se puede realizar iteraciones (`for in` o `while`).

4.a Arreglo (Vector)

[16]:
```
import numpy as np
# Ejemplo_16: Se define un vector (arreglo) y posterior a ello, se duplicará␣
 ↪cada
# elemento del arreglo
arreglo_16 = np.array([2, 4, 6, 7, 10])
print(arreglo_16)

print("\n")

i = 0
while i < len(arreglo_16):
  arreglo_16[i] = arreglo_16[i]*2
  i = i + 1
print(arreglo_16)
```

```
[ 2  4  6  7 10]


[ 4  8 12 14 20]
```

4.b Matriz

[17]:
```
import numpy as np
# Ejemplo_17: Se define una matriz y posterior a ello, se imprimirá cada fila
matriz_17 = np.matrix([[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]])
```

```
print(matriz_17)

print("\n")

for fila in matriz_17:
  print(fila)
```

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


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

4.c Aplanar arreglos Tal como se mostró en el **ejemplo_17** al realizar una iteración en una matriz, se accede a las filas que la conforman. Para poder acceder a los elementos de cada fila (sea un arreglo multidimensional o una matriz), el módulo `Numpy` ofrece la propiedad llamada `.flat`, la cual permite, mediante una sola iteración, acceder a las columnas que conforman la fila de la matriz, "aplanando" el objeto.

[18]:
```
import numpy as np
# Ejemplo_18: Se define una matriz y posterior a ello, se imprimirá cada fila
matriz_18 = np.matrix([[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]])
print(matriz_18)

print("\n")

for numero in matriz_18.flat:
  print(numero)
```

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


2
4
6
8
10
1
3
5
7
9
```

1.4.5 5. Modificar Dimensiones

Una de las características que ofrece Numpy relacionado con arreglos y matrices es la posibilidad de poder cambiar las dimensiones, pero manteniendo la cantidad de elementos que conforman el tipo de dato definido.

5.a Arreglo (Vector)

[19]:
```
import numpy as np
# Ejemplo_19: Se define un vector (arreglo) multidimensional y posterior a ello,␣
 ↪se cambiarán las dimensiones
arreglo_19 = np.array([[2, 4, 6], [1, 3, 7]])
print(arreglo_19)

print("\n")

nuevo_arreglo = arreglo_19.flatten()
print(nuevo_arreglo)
```

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


[2 4 6 1 3 7]
```

El **ejemplo_19** muestra como el arreglo multidimensional cambió a un nuevo arreglo de una sola dimension, conservando todos los elementos originales

Además, es posible utilizar nuevas dimensiones arbitrarias, siempre y cuando el total de elementos se mantenga:

[23]:
```
import numpy as np

arreglo = np.array([[2, 4, 6, 8], [1, 3, 7, 9]])
print("Original:", arreglo, sep="\n", end="\n\n")

# Le damos una lista con la cantidad de elementos en cada nueva dimensión
arreglo = arreglo.reshape([4, 2])
print("Transpuesto:", arreglo, sep="\n", end="\n\n")

# Pueden ser más de dos dimensiones
arreglo = arreglo.reshape([2, 2, 2])
print("Dos matrices de 2x2:", arreglo, sep="\n", end="\n\n")
```

```
Original:
[[2 4 6 8]
 [1 3 7 9]]

Transpuesto:
[[2 4]
 [6 8]
```

```
 [1 3]
 [7 9]]

Dos matrices de 2x2:
[[[2 4]
  [6 8]]

 [[1 3]
  [7 9]]]
```

5.b Matriz

```
[ ]: import numpy as np
     #Ejemplo_20: Se define una matriz y posterior a ello, se cambiarán las␣
      ↪dimensiones
     matriz_20 = np.matrix([[2, 4, 6, 8, 10], [1, 3, 5, 7, 9]])
     print(matriz_20)

     print("\n")

     nueva_matriz = matriz_20.flatten()
     print(nueva_matriz)
```

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


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

El **ejemplo_20** muestra como la matriz cambió sus dimensiones y se convirtió en una nueva matriz donde tiene solo 1 fila y tantas columnas como elementos tenía la matriz original.

1.5 Funciones Generadoras

Dentro de las distintas herramientas que ofrece Numpy, se destacan algunas funciones que permiten facilitar el trabajo que se lleva a cabo con el uso de vectores y matrices, vale decir, Numpy ofrece funciones que generan determinadas matrices/vectores para algunos casos determinados. Entre dichas funciones se pueden encontrar:

1.5.1 1. Función arange()

La función `arange` genera un vector con la cantidad de datos que se desea. Para ello, se pueden ingresar 3 parámetros, `arange(x, y, z)`:

- **x**: número inicial
- **y**: número final (al cual no se accede)
- **z**: distancia entre los valores del vector

[24]:
```
import numpy as np
# Ejemplo 21: se genera un vector mediante el uso de la función arange
vector = np.arange(2, 20, 3)
print(vector)
```

```
[ 2  5  8 11 14 17]
```

La función arange puede usarse solo con 2 parámetros, `arange(x, y)`, omitiéndose el parámetro **z**, el cual por defecto toma el valor 1 (distancia entre los puntos)

[25]:
```
import numpy as np
# Ejemplo 22: se genera un vector mediante el uso de la función arange
vector = np.arange(2, 20)
print(vector)
```

```
[ 2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

Por último, la función `arange` puede invocarse solo con un parámetro `arange(y)`, omitiéndose los parámetros **x** e **z**. Dichos valores tomaran valores por defecto **0** y **1** respectivamente

[ ]:
```
import numpy as np
# Ejemplo 23: se genera un vector mediante el uso de la función arange
vector = np.arange(20)
print(vector)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

Si bien su uso se basa en la función nativa `range`, tiene dos diferencias importantes con esta: genera un `array` como resultado y puede usar números flotantes como parámetros:

[26]:
```
import numpy as np
# Ejemplo 21: se genera un vector mediante el uso de la función arange
vector = np.arange(-1.5, 2.0, 0.5)
print(vector)
```

```
[-1.5 -1.  -0.5  0.   0.5  1.   1.5]
```

1.5.2 2. Función `linspace()`

Una función similar a la vista anteriormente es la función linspace(), la cual genera un vector mediante el ingreso de 3 parámetros `linspace(x, y, z)`, donde: * **x**: Número inicial * **y**: Número final (que sí se incluye en el vector final) * **z**: cantidad de puntos equidistantes

[28]:
```
import numpy as np
#Ejemplo 24: se genera un vector mediante el uso de la función linspace
vector = np.linspace(2, 6, 8)
print(vector)
```

```
[2.         2.57142857 3.14285714 3.71428571 4.28571429 4.85714286
 5.42857143 6.        ]
```

La función `linspace` puede usarse solo con 2 parámetros `linspace(x, y)`, omitiéndose el parámetro **z**, el cual por defecto toma el valor 50 (cantidad de puntos equidistantes)

[27]:
```
import numpy as np
# Ejemplo 25: se genera un vector mediante el uso de la función linspace
vector = np.linspace(2, 100)
print(vector)
```

```
[  2.   4.   6.   8.  10.  12.  14.  16.  18.  20.  22.  24.  26.  28.
  30.  32.  34.  36.  38.  40.  42.  44.  46.  48.  50.  52.  54.  56.
  58.  60.  62.  64.  66.  68.  70.  72.  74.  76.  78.  80.  82.  84.
  86.  88.  90.  92.  94.  96.  98. 100.]
```

1.5.3 3. Función `diag()`

Numpy ofrece funciones para trabajar directamente con matrices, y una de las funciones que se puede destacar es `diag()`. Esta función recibe como entrada una lista o vector (arreglo) y entrega como salida un arreglo de dos dimensiones, donde el vector/lista ingresada corresponde a la **diagonal** y el resto de los números tienen valor 0.

[29]:
```
import numpy as np
# Ejemplo 26: se genera un vector de dos dimensiones cuya diagonal corresponde a␣
 ↪la lista ingresada
vector = np.diag([1, 2, 3, 4, 5])
print(vector)
```

```
[[1 0 0 0 0]
 [0 2 0 0 0]
 [0 0 3 0 0]
 [0 0 0 4 0]
 [0 0 0 0 5]]
```

1.5.4 4. Función `zeros()`

La función `zeros()` es una función generadora de vectores. Para utilizarla se requiere como entrada la cantidad de filas y columnas deseadas (**en formato de lista**), generando un vector con las dimensiones mencionadas con todos sus elementos de valor 0. Si se da solo un número como parámetro, se genera un vector plano con esa cantidad de elementos.

[30]:
```
import numpy as np
# Ejemplo 27: se genera un vector multidimensional con las dimensiones indicadas␣
 ↪en formato lista
vector = np.zeros([5, 3])
print(vector)
```

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

1.5.5 5. Función `ones()`

Al igual que la función `zeros()`, la función `ones()` es una generadora de vectores. Para utilizarla se requiere como entrada la cantidad de filas y columnas deseadas (**en formato de lista**), generando un vector con las dimensiones mencionadas con todos sus elementos de valor 1. Si se da solo un número como parámetro, se genera un vector plano con esa cantidad de elementos.

[31]:
```
import numpy as np
# Ejemplo 28: se genera un vector multidimensional con las dimensiones indicadas␣
 ↪en formato lista
vector = np.ones([4, 6])
print(vector)
```

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

1.5.6 6. Función `identity()`

La función `identity()` genera un vector multidimensional en donde la diagonal de dicho vector está constituido por valores 1. En otras palabras, esta función genera la matriz identidad de las dimensiones señaladas. Para poder utilizarla, se requiere que, como parámetro de entrada, se ingrese la cantidad de filas/columnas que se requiere (solo será un parámetro, puesto que genera una matriz de dimensiones $n \times n$).

[32]:
```
import numpy as np
# Ejemplo 29: se genera un vector multidimensional correspondiente a la matriz␣
 ↪identidad
vector_identidad = np.identity(4)
print(vector_identidad)
```

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

2 Bibliografía

3 Numpy

GeeksforGeeks. (2018, 15 octubre). Python Numpy. Recuperado 9 de agosto de 2022, de https://www.geeksforgeeks.org/python-numpy/

Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2. (Publisher link).

NumPy. (2022, 22 junio). Learn. Recuperado 9 de agosto de 2022, de https://numpy.org/learn/

Sundnes, J. (2020). Arrays and Plotting. En *Introduction to Scientific Programming with Python* (1.a ed., p. 81). Springer. https://doi.org/10.1007/978-3-030-50356-7