Operando con funciones universales

Hay muchas operaciones que procesan un ndarray, o varios, elemento a elemento aplicando la misma operación de forma reiterada. Este tipo de operación se denomina "función universal" (*ufunc*).

Numpy implementa todas estas funciones a bajo nivel en lenguaje "C" para conseguir una eficiencia similar a la obtenida si se ejecutaran directamente desde un programa compilado con dicho lenguaje. Este cuaderno se basa en la documentación oficial que puedes consultar aquí.

Configuración del entorno.

Como siempre, comenzamos cargando el módulo y visualizando su versión ya que este dato es importante para consultar la documentación.

En el momento de escribir este tutorial la versión de Numpy con la que se trabaja es 1.22.4.

Observa que además activamos de forma global una opción para que al imprimir números con una precisión de 3 cifras.

In [1]:
```
import numpy as np
print('Numpy version: {}'.format(np.__version__))
np.set_printoptions(precision=3)
```

```
Numpy version: 1.22.4
```

Funciones universales y vectorización de código.

Una función universal (ufunc) es una función que se aplica a cada elemento de un ndarray de forma independiente ("*element-wise*").

Supongamos que necesitamos calcular como salida un ndarray donde cada elemento sea la raíz cuadrada del correspondiente elemento de la entrada. Una posible implementación podría ser la función:

In [2]:
```
import math as m
def sqrt_python(input):
    """Calcular la raíz cuadrada de cada elemento del array de entrada.

    Versión python iterando sobre una vista plana del array aplicando
    la función sqrt del módulo python math."""
    output = np.empty_like(input)
    for i in range(input.size):
        output.flat[i] = m.sqrt(input.flat[i])
    return output
```

In [3]:
```
def sqrt_numpy(input):
    """Calcular la raíz cuadrada de cada elemento del array de entrada.
```

```
    Versión numpy utilizando la ufunc numpy.sqrt()"""
    return np.sqrt(input)
```

Observa ya en primer lugar cómo la versión "numpy" es muy simple.

Ahora vamos a compara los tiempos.

In [4]:
```
a = np.linspace(0, 1, 10000).reshape(-1, 100)
print('Calcular el array cuyos elementos son la raíz cuadrada,')
print(f'de un array con forma: {a.shape}:')
print('')

print('Tiempo utilizado por la versión "python":')
b = sqrt_python(a)
b_time = %timeit -o sqrt_python(a)
print('')

print('Tiempo utilizado por la versión "numpy":')
c = sqrt_numpy(a)
c_time = %timeit -o sqrt_numpy(a)

print('')
print(f'Speedup (numpy vs python)  : {int(b_time.average/c_time.average)}')
print(f'Diferencia entre resultados: {np.sum(np.abs(b-c))}')
```

```
Calcular el array cuyos elementos son la raíz cuadrada,
de un array con forma: (100, 100):

Tiempo utilizado por la versión "python":
3.1 ms ± 87.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

Tiempo utilizado por la versión "numpy":
7.04 µs ± 47.5 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

Speedup (numpy vs python)  : 440
Diferencia entre resultados: 0.0
```

Con el hardware usado para realizar este cuaderno la versión "numpy" es más de 400 veces más rápida que la versión "python" y el resultado es lógicamente el mismo.

Como vemos al utilizar la función universal `np.sqrt()` hemos simplifiado nuestro código eliminando bucles python (que son muy lentos como se puede apreciar). De esta forma se dice que hemos **vectorizado** nuestro código Python.

Funciones universales para aritmética.

Numpy implementa las principales funciones aritméticas como funciones universales. Se recomienda leer esta entrada antes de realizar los ejercicios siguientes

Ejercicio 01: Se desea obtener la suma de dos vectores con el mismo tamaño N.

La salida debería ser algo parecido a lo siguiente:

```
c:  [2. 2. 2. 2. 2.]
```

In [5]:
```
a = np.ones(5)
b = np.ones(5)
c = np.zeros(5)
#pon tu código aquí
```

```
c = np.add(a, b)
#
print('c: ', c)
```

```
c:  [2. 2. 2. 2. 2.]
```

Ejercicio 02: Se desea obtener la diferencia de dos vectores con el mismo tamaño N.

La salida debería ser algo parecido a lo siguiente:

```
c:  [1. 1. 1. 1. 1.]
```

In [8]:
```
a = np.full(5, 2.0)
b = np.ones(5)
c = np.zeros(5)
#pon tu código aquí

#
print('c: ', c)
```

```
c:  [0. 0. 0. 0. 0.]
```

Ejercicio 03: Se desea obtener el producto por elemento de dos vectores con el mismo tamaño N.

La salida debería ser algo parecido a lo siguiente:

```
c:  [4. 4. 4. 4. 4.]
```

In [9]:
```
a = np.full(5, 2.0)
b = np.full(5, 2.0)
c = np.zeros(5)
#pon tu código aquí

#
print('c: ', c)
```

```
c:  [0. 0. 0. 0. 0.]
```

Ejercicio 04: Se desea obtener la división con máxima precisión (*true division*) por elemento de dos vectores con el mismo tamaño N. Nota: hay dos funciones que hacen esto.

La salida debería ser algo parecido a lo siguiente:

```
c:  [1.5 1.5 inf nan 1.5]
```

In [10]:
```
a = np.full(5, 3.0)
b = np.full(5, 2.0)
b[2] = 0.0
a[3] = 0.0
b[3] = 0.0
c = None
#pon tu código aquí

#
print('c: ', c)
```

```
c:  None
```

Ejercicio 05: Se desea obtener la división con truncamiento "*floor_division*" por elemento de dos vectores

con el mismo tamaño N.

La salida debería ser algo parecido a lo siguiente:

```
c:  [ 1.  1. inf nan  1.]
```

In [11]:
```
a = np.full(5, 3.0)
b = np.full(5, 2.0)
b[2] = 0.0
a[3] = 0.0
b[3] = 0.0
#pon tu código aquí

#
print('c: ', c)
```

```
c:  None
```

Observa cómo, en ambos tipos de división, al dividir entre cero, Numpy nos dá un aviso y el resultado de la operación es `np.inf`. Por otro lado si se divide 0/0 da como resultando `np.nan` (Not A Number).

Usando escalares y operadores.

En las anteriores operaciones aritméticas podremos usar valor escalar para operar con cada uno de los valores. Además también podemos utilizar los operadores python `'+', '-', '\*', '/' y '//'`.

Ejercicio 06: Dado un vector de tamaño N, se desea dividir entre dos con truncamiento ("*floor division*") los elementos del array.

La salida debería ser algo parecido a lo siguiente:

```
c:  [1. 1. 1. 1. 1.]
```

In [12]:
```
a = np.full(5, 3.0)
c = None
#pon tu código aquí
#Sugerencia: prueba el operador '//'.

#
print('c: ', c)
```

```
c:  None
```

Aumentando la eficiencia.

Las funciones universales tienen un argumento opcional `out`. Este argumento se utiliza para proporcionar un lugar donde almacenar el resultado de la operación. En caso de que no se proporcione (la opción por defecto), será la propia función la que reserve ese espacio y lo devuelva como resultado.

En expresiones complejas, podremos aumentar la eficiencia reutilizando espacios de almacenamiento para cálculos intermedios aprovechando este argumento opcional.

Con el mismo objetivo, si utilizamos operadores python, podemos utilizar las operaciones "*in-place*" `'+='`, `'-='`, `'*='`, `'/='`. Por ejemplo la operación `a = a + b` se puede escribir como `a += b`.

Ejercicio 07: Sean A, B y C tres arrays. Se desea implementar la operación $D = A * B + C$ reutilizando el almacenamiento temporal de la operación A*B para sumar C y que sea finalmente el resultado.

La salida debería ser algo parecido a lo siguiente:

```
R=
[[ 2.     4.667]
 [ 7.333 10.   ]]
```

In [13]:
```
A = np.linspace(1., 1., 4).reshape(2, 2)
B = np.linspace(1., 2., 4).reshape(2, 2)
C = np.linspace(1., 8., 4).reshape(2, 2)
R = None

#Pon tu código aquí

#

print('R=')
print(R)
```

```
R=
None
```

Funciones universales para trigonometría.

Numpy implementa las principales funciones universales relativas a la trigonometría. Se recomienda leer esta entrada antes de realizar los siguientes ejercicios.

Ejercicio 08: Se desea dividir el espacio lineal $[-\pi/2, \pi/2]$ en 5 muestras y calcular la función 'seno'.

La salida debería ser algo parecido a lo siguiente:

```
X     :  [-1.571 -0.785  0.     0.785  1.571]
SEN(X):  [-1.    -0.707  0.     0.707  1.   ]
```

In [14]:
```
X=np.zeros(5)
Y=np.zeros(5)
#Pon tu código aquí.
#Sugerencia, utiliza la constante np.pi que proporciona numpy.

#
print('X     : ', X)
print('SEN(X): ', Y)
```

```
X     :  [0. 0. 0. 0. 0.]
SEN(X):  [0. 0. 0. 0. 0.]
```

Ejercicio 09: Se desea dividir el espacio lineal $[-\pi/2, \pi/2]$ en 5 muestras y calcular la función 'coseno'.

La salida debería ser algo parecido a lo siguiente:

```
X     :  [-1.571 -0.785  0.     0.785  1.571]
COS(X):  [6.123e-17 7.071e-01 1.000e+00 7.071e-01 6.123e-17]
```

In [15]:
```
X=np.zeros(5)
```

```
Y=np.zeros(5)
#Pon tu código aquí.
#Sugerencia, utiliza la constante np.pi que proporciona numpy.

#
print('X     : ', X)
print('COS(X): ', Y)
```

```
X     :  [0. 0. 0. 0. 0.]
COS(X):  [0. 0. 0. 0. 0.]
```

Ejercicio 10: Sen X,Y dos vectores con la función coseno/seno calculada a partir del espacio lineal $[0, 2\pi]$ en 9 muestras. Se quiere recuperar el ángulo medido grados sexagesimales usando la función arco tangente.

La salida debería ser algo parecido a lo siguiente:

```
COS(A):  [ 1.000e+00  7.071e-01  6.123e-17 -7.071e-01 -1.000e+00
-7.071e-01
 -1.837e-16  7.071e-01  1.000e+00]
SEN(A):  [ 0.000e+00  7.071e-01  1.000e+00  7.071e-01  1.225e-16
-7.071e-01
 -1.000e+00 -7.071e-01 -2.449e-16]
A     :  [ 0.000e+00  4.500e+01  9.000e+01  1.350e+02  1.800e+02
-1.350e+02
 -9.000e+01 -4.500e+01 -1.403e-14]
```

In [16]:
```
X=np.zeros(5)
Y=np.zeros(5)
A=np.zeros(5)
#Pon tu código aquí.
#Sugerencia, utiliza la constante np.pi que proporciona numpy.

#
print('COS(A): ', X)
print('SEN(A): ', Y)
print('A     : ', A)
```

```
COS(A):  [0. 0. 0. 0. 0.]
SEN(A):  [0. 0. 0. 0. 0.]
A     :  [0. 0. 0. 0. 0.]
```

Funciones universales para comparar y realizar operaciones lógicas.

Numpy implementa operaciones de comparación que dan como resultado un ndarray con valores lógicos `true` o `false`. Se recomienda leer la referencia antes de realizar los siguientes ejercicios.

Observar que estas funciones también pueden usarse a través de los operadores python `'<', '<=', '==', '>', '>='`.

Ejercicio 11: Se desea comparar obtener un array indicando que elementos del array A son menores que los del array B.

La salida debería ser algo parecido a lo siguiente:

```
A  :  [77 53 57 14 48]
```

```
B  :  [86 83 15 38 42]
A<B:  [ True  True False  True False]
```

In [17]:
```
A = np.array([77, 53, 57, 14, 48])
B = np.array([86, 83, 15, 38, 42])
C = np.zeros(5)
#Sugerencia: prueba a implementar usando una ufunc y también con el
# operador python correspondiente.

#
print('A  : ', A)
print('B  : ', B)
print('A<B: ', C)
```

```
A  :  [77 53 57 14 48]
B  :  [86 83 15 38 42]
A<B:  [0. 0. 0. 0. 0.]
```

Ejercicio 12: Se desea comparar obtener un array indicando que elementos del array A son menores o iguales que los del array B.

La salida debería ser algo parecido a lo siguiente:

```
A   :  [77 53 57 14 48]
B   :  [86 83 15 38 42]
A>=B:  [False False  True False  True]
```

In [18]:
```
A = np.array([77, 53, 57, 14, 48])
B = np.array([86, 83, 15, 38, 42])
C = np.zeros(5)
#Pon tu código aquí.
#Sugerencia: prueba a implementar usando una ufunc y también con el
# operador python correspondiente.

#
print('A   : ', A)
print('B   : ', B)
print('A>=B: ', C)
```

```
A   :  [77 53 57 14 48]
B   :  [86 83 15 38 42]
A>=B:  [0. 0. 0. 0. 0.]
```

Ejercicio 13: Dada un array A se desea indicar que elementos de A están en el intervalo $[14, 57)$.

La salida debería ser algo parecido a lo siguiente:

```
A         :  [77 53 57 14 48]
IN [10,57):  [False  True False  True  True]
```

In [ ]:
```

```

In [19]:
```
A = np.array([77, 53, 57, 14, 48])
R = np.zeros(5)
#Pon tu código aquí.
#Sugerencia: prueba a implementar usando ufuncs y también con el
# operador python correspondiente.

#
```

```
print('A         : ', A)
print('IN [10,57): ', R)
```

```
A         :  [77 53 57 14 48]
IN [10,57):  [0. 0. 0. 0. 0.]
```

Ejercicio 14: Dada un array A con valores de años, se desea indicar que elementos de A son años bisiestos. Recuerda que un año es bisiesto si es un mútiplo de 4, salvo que los que sean un múltipo de 100 y no de 400.

La salida debería ser algo parecido a lo siguiente:

```
Año    1: M4? 0 M100? 0 M400? 0 Bisiesto? 0
Año    4: M4? 1 M100? 0 M400? 0 Bisiesto? 1
Año  100: M4? 1 M100? 1 M400? 0 Bisiesto? 0
Año  200: M4? 1 M100? 1 M400? 0 Bisiesto? 0
Año  400: M4? 1 M100? 1 M400? 1 Bisiesto? 1
Año 1700: M4? 1 M100? 1 M400? 0 Bisiesto? 0
Año 1000: M4? 1 M100? 1 M400? 0 Bisiesto? 0
Año 2020: M4? 1 M100? 0 M400? 0 Bisiesto? 1
```

In [20]:
```
A = np.array([1, 4, 100, 200, 400, 1700, 1000, 2020])
M4 = np.zeros_like(A) # Es múltiplo de 4?
M10 = np.zeros_like(A) # Es múltiplo de 10?
M100 = np.zeros_like(A) # Es múltiplo de 100?
M400 = np.zeros_like(A) # Es múltiplo de 400?
R = np.zeros_like(A) #Es bisiesto?
#Pon tu código aquí.
#Sugerencia: para saber si un número N es múltiplo de 4, el resto
#de la división entera de N entre 4 debe ser 0.

#
for a in range(A.size):
    print(f'Año {A[a]:4}: M4? {M4[a]:1} M100? {M100[a]:1} M400? {M400[a]:1} Bisiesto? {R
```

```
Año    1: M4? 0 M100? 0 M400? 0 Bisiesto? 0
Año    4: M4? 0 M100? 0 M400? 0 Bisiesto? 0
Año  100: M4? 0 M100? 0 M400? 0 Bisiesto? 0
Año  200: M4? 0 M100? 0 M400? 0 Bisiesto? 0
Año  400: M4? 0 M100? 0 M400? 0 Bisiesto? 0
Año 1700: M4? 0 M100? 0 M400? 0 Bisiesto? 0
Año 1000: M4? 0 M100? 0 M400? 0 Bisiesto? 0
Año 2020: M4? 0 M100? 0 M400? 0 Bisiesto? 0
```

Funciones universales para manipular número flotantes.

Hay un conjunto de operaciones universales para manipular números flotantes. Se recomienda leer la referencia antes de realizar los ejercicios siguientes.

Ejercicio 15: Tras realizar una operación con dos arrays queremos indicar los valores que han resultado con valor infinito (tienen valor `np.inf`).

La salida debería ser algo parecido a lo siguiente:

```
A/B         :  [ 2.   1.5 -inf  1.5  inf]
Es infinito?:  [False False  True False  True]
```

```
A=np.array([2, 3, -1 , 3 , 5], 'f')
```

In [21]:
```
B=np.array([1, 2, 0 , 2 , 0], 'f')
R1=np.zeros(5)
R2=np.zeros(5)
#Pon tu código aquí.

#
print('A/B         : ',R1)
print('Es infinito?: ',R2)
```

```
A/B         :  [0. 0. 0. 0. 0.]
Es infinito?:  [0. 0. 0. 0. 0.]
```

Ejercicio 16: Tras realizar una operación con dos arrays queremos indicar los valores que no han podido calcularse (tienen valor np.nan).

La salida debería ser algo parecido a lo siguiente:

```
A//B    :  [ 2.  1. nan  1. inf]
Es nan? :  [False False  True False False]
```

In [22]:
```
A=np.array([2, 3, 0 , 3 , 1], 'f')
B=np.array([1, 2, 0 , 2 , 0], 'f')
R1=np.zeros(5)
R2=np.zeros(5)
#Pon tu código aquí.

#
print('A//B    : ',R1)
print('Es nan? : ',R2)
```

```
A//B    :  [0. 0. 0. 0. 0.]
Es nan? :  [0. 0. 0. 0. 0.]
```

Broadcasting.

Las funciones universales obtienen nuevos ndarrays a partir de otros ndarrays como entrada aplicando una función a cada uno de los elementos de la entrada, los cuales son en general una escalar.

Hay funciones universales que utilizan como entrada más de un ndarray, por ejemplo la suma. ¿Qué ocurre si las formas de las entrada no coinciden? ¿se podrá aplicar la función universal aún?. La respuesta a estas preguntas son denominadas como "broadcasting rules", que podríamos traducir como, reglas de propagación de la función universal sobre las entradas. Estas reglas se pueden resumir de la siguiente forma:

Supongamos una función universal con entradas `input_1`, `input_2`, ... `input_n`.

- Todas las entradas `input_x` con `input_x.ndim < max_dim = max{input_1.ndim, input_2.ndim, ..., input_n.ndim}` pre-añaden dimensiones con tamaño 1 hasta igualar `max_dim`.
- La salida tendrá `max_dim` dimensiones cuyos tamaños serán el tamaño máximo de la correspondiente dimensión en las entradas.
- Una entrada es válida si su tamaño para cada dimensión es 1 o igual al tamaño de la salida para esa dimensión.
- Si una entrada tiene tamaño 1 para una dimensión, el valor de ese elemento es utilizado para todas las operaciones en esa dimensión.

Por ejemplo, supongamos que vamos a ejecutar una función universal con cuatro argumentos: a, b, c y d. Si `a.shape` es (5,1), `b.shape` es (1,6), `c.shape` es (6,) y `d.shape` es () (es decir es una escalar), entonces antes de realizar la función universal, a, b, c, y d son propagadas ("*broadcasted*") a la forma (5, 6) aplicando las reglas:

- `a` se propaga como un ndarray (5, 6) donde `a[:,0]` se copia al resto de columnas.
- `b` se propaga como un ndarray (5, 6) donde `b[0,:]` se copia al resto de filas.
- `c` pre añade una dimesión con tamaño 1, actuando como un ndarray (1, 6) y después se propaga como un ndarray (5, 6) copiando `c[:]` en cada fila.
- `d` pre añade dos dimensiones con tamaño 1, actuando como un ndarray (1, 1) y por lo tanto se propaga como un ndarray (5, 6) copiando la escalar d en cada una de los 5x6 valores.

Así por ejemplo:

In [23]:
```
a = np.arange(10).reshape(5, 2)
b = np.array([1, 2])
print('a=\n', a)
print('a.shape = ', a.shape)
print('b=\n', b)
print('b.shape = ', b.shape)
c = a + b
print('c = a + b = \n', c)
print('b.shape fue pre añadida con 1 para obtener la forma (1, 2)\n',
      '  y propagada a la forma (5, 2) copiando la fila b[0, :] al resto de filas.')
```

```
a=
 [[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
a.shape =  (5, 2)
b=
 [1 2]
b.shape =  (2,)
c = a + b =
 [[ 1  3]
 [ 3  5]
 [ 5  7]
 [ 7  9]
 [ 9 11]]
b.shape fue pre añadida con 1 para obtener la forma (1, 2)
   y propagada a la forma (5, 2) copiando la fila b[0, :] al resto de filas.
```

Veamos ahora un ejemplo de una entrada no válida:

In [24]:
```
a = np.arange(10).reshape((2, 5))
b = np.array([1, 2])
print('a=\n', a)
print('a.shape = ', a.shape)
print('b=\n', b)
print('b.shape = ', b.shape)

try:
    c = a + b
except ValueError as e:
    print('¿Por qué no se puede realizar la operación c = a+b?\n', e)
    print('b.shape fue preañadida con 1 para obtener la forma (1, 2)\n',
      '  pero no se puede propagar a la forma (2, 5) ya que b.shape[1]==2 < 5.')
```

```
a=
```

```
 [[0 1 2 3 4]
 [5 6 7 8 9]]
a.shape =  (2, 5)
b=
 [1 2]
b.shape =  (2,)
¿Por qué no se puede realizar la operación c = a+b?
 operands could not be broadcast together with shapes (2,5) (2,)
b.shape fue preañadida con 1 para obtener la forma (1, 2)
   pero no se puede propagar a la forma (2, 5) ya que b.shape[1]==2 < 5.
```

Promoción del tipo (*type casting*).

Al igual que ocurre con las formas de las entradas, al combinar varias entradas para realizar una función universal puede ocurrir que los tipos de los elementos de las entradas no coincidan. Si esto ocurre, en general, antes de aplicar la función universal, se aplicará un proceso de promoción de tipos (type casting) que consiste en obtener una versión de las entradas donde los tipos con menor poder de representación numérica se han promocionado al tipo de mayor representación numérica usado en alguna de las entradas (y siempre que la promoción se pueda realizar de forma segura).

Por ejemplo si `a` es un array de enteros y `b` es un array de flotantes, ¿`c = a+b` qué tipo tendrá?:

In [25]:
```
a = np.arange(5, dtype='i')
print('a: ', a)
print('a.dtype =', a.dtype)
print('b: ', b)
b = np.arange(5, dtype='d')
print('b.dtype =', b.dtype)
print('a.dtype puede promocionarse a b.dtype de forma segura? :',
      'Sí.' if np.can_cast(a.dtype, b.dtype) else 'No.')
print('b.dtype puede promocionarse a a.dtype de forma segura? :',
      'Sí.' if np.can_cast(b.dtype, a.dtype) else 'No.')
c = a + b
print('c = a+b\n', c)
print('c.dtype =', c.dtype)
```

```
a:  [0 1 2 3 4]
a.dtype = int32
b:  [1 2]
b.dtype = float64
a.dtype puede promocionarse a b.dtype de forma segura? : Sí.
b.dtype puede promocionarse a a.dtype de forma segura? : No.
c = a+b
 [0. 2. 4. 6. 8.]
c.dtype = float64
```

Métodos `reduce` y `accumulate`.

Las funciones universales se representan como clases herederas de una superclase "ufunc" y proporcionan varios métodos útiles de los cuales vamos destacar los métodos `reduce` y `accumulate`.

Método `ufunc.reduce`.

El método reduce permite reducir una dimensión de un ndarray aplicando una ufunc a todos los elementos de esa dimensión. Por defecto se aplica sobre el primer eje (axis=0) pero podemos indicar sobre qué eje queremos aplicar la reducción.

Ejercicio 17: Se desea implementar el producto escalar de dos vectores que están dados como dos array con forma (N,). Recuerda que el producto escalar de dos vectores sería equivalente a sumar los productos parciales de cada componente, es decir, $pe = a[0] * b[0] + a[1] * b[1] + \cdots + a[N-1] * b[N-1]$

La salida debería ser algo parecido a lo siguiente:

```
Producto escalar : -1.5
```

In [26]:
```
A = np.array([-1.0, 0.0, 1.0])
B = np.array([2.0, 1.5, 0.5])
pe = 0.0
#Pon tu código aquí.
#Sugerencia: para reducir una dimensión sumando usaremos
#el método np.add.reduce

#
print("Producto escalar :", pe)
```

```
Producto escalar : 0.0
```

Por defecto la reducción se aplica al eje 0, pero podemos aplicar la reducción a otro eje.

Ejercicio 18: Supongamos que la matriz `a` define una tabla de frecuencias de dos variables conjuntas 'v1' que son las filas y 'v2' que son las columnas. Se desea obtener la distribución marginal de 'v1' y de 'v2', es decir, dos vectores donde cada elemento 'i' es la suma de todos los elementos de la fila 'i' (para v1) o de la columna i (para v2) de la matriz.

La salida debería ser algo parecido a lo siguiente:

```
A :
 [[ 8.  1.  9.  4. 10.]
 [ 9.  5.  9.  6.  5.]
 [ 4.  9.  6.  5.  2.]
 [ 2.  6.  7.  1.  3.]]
V1: [32. 34. 26. 19.]
V2: [23. 21. 31. 16. 20.]
```

In [27]:
```
A = np.array([[ 8.,  1.,  9.,  4., 10.],
 [ 9.,  5.,  9.,  6.,  5.],
 [ 4.,  9.,  6.,  5.,  2.],
 [ 2.,  6.,  7., 1.,  3.]])
V1 = 0
V2 = 0
#Pon tu código aquí.
#Sugerencia: para reducir una dimensión sumando usaremos
#el método np.add.reduce

#
print("A :\n", A)
print("V1:", V1)
print("V2:", V2)
```

```
A :
 [[ 8.  1.  9.  4. 10.]
 [ 9.  5.  9.  6.  5.]
 [ 4.  9.  6.  5.  2.]
 [ 2.  6.  7.  1.  3.]]
V1: 0
V2: 0
```

Ejercicio 19: Dado el array `a` queremos reducirlo como la escalar resultante de multiplicar todos sus valores.

La salida debería ser algo parecido a lo siguiente:

```
a:  [1 2 3 4]
reducción:  24
```

In [28]:
```
a = np.arange(1, 5)
v = 0
#Pon tu código aquí.
#Sugerencia: para reducir una dimensión multiplicando usaremos
#el método np.multiply.reduce

#
print('a: ', a)
print('reducción: ', v)
```

```
a:  [1 2 3 4]
reducción:  0
```

Método `ufunc.accumulate`.

Este método acumula el resultado de aplicar la función universal a lo largo de una dimensión dejando el total acumulado en el último elemento de dicha dimensión. Por defecto se aplica sobre el primer eje (axis=0) pero podemos indicar sobre qué eje queremos aplicar la acumulación.

Ejercicio 20: A partir de un array `a` queremos obtener una array `b` donde cada elemento i se igual a la suma de todos los elementos de `a` en el rango $[0:i]$.

La salida debería ser algo parecido a lo siguiente:

```
a:  [1 2 3 4]
reducción:  24
```

In [29]:
```
a=np.arange(5)
b=np.zeros(5)
#Pon tu código aquí.
#Sugerencia: para acumular en una dimensión usaremos
#el método np.add.acumulate

#
print('a: ', a)
print('b: ', b)
```

```
a:  [0 1 2 3 4]
b:  [0. 0. 0. 0. 0.]
```

Poniendo todo junto.

Veamos algunos ejemplos donde tendrás que combinar todo lo visto en los apartados anteriores.

Ejercicio 21: Definir una función para calcular la distancia euclídea de dos vectores. Recuerda que la distancia euclídea entre dos vectores se define como:

$d(a,b) = \sqrt{\sum_i (a_i - b_i)^2}$

La salida debería ser algo parecido a lo siguiente:

```
A:  [0 1 2 3 4]
B:  [1 2 3 4 5]
Distancia A,B: 2.236
```

In [30]:
```
def distancia_euclida(A, B):
    '''
    Distancia euclídea
    Params:
       A, B son vectores con el mismo tamaño.
    Retorna:
       La distancia euclídea.
    '''
    dist=0.0
    #Pon tu código aquí

    #
    return dist

A=np.arange(0, 5)
B=np.arange(1, 6)
print('A: ', A)
print('B: ', B)
print('Distancia A,B: {0:.3f}'.format(distancia_euclida(A, B)))
```

```
A:  [0 1 2 3 4]
B:  [1 2 3 4 5]
Distancia A,B: 0.000
```

Ejercicio 22: Definir una función para calcular la distancia de dos vectores definida como la suma de diferencias en valor absoluto:

$d(a,b) = \sum_i |a_i - b_i|$

La salida debería ser algo parecido a lo siguiente:

```
A:  [0 1 2 3 4]
B:  [1 2 3 4 5]
Distancia A,B: 5.000
```

In [31]:
```
def distancia_L1(A, B):
    '''
    Distancia usando la norma L1.
    Params:
       A, B son vectores con el mismo tamaño.
    Retorna:
       La norma L1 del vector diferencia.
    '''
    dist=0.0
    #Pon tu código aquí

    #
    return dist

A=np.arange(0, 5)
B=np.arange(1, 6)
print('A: ', A)
```

```
print('B: ', B)
print('Distancia A,B: {0:.3f}'.format(distancia_L1(A, B)))
```

```
A:  [0 1 2 3 4]
B:  [1 2 3 4 5]
Distancia A,B: 0.000
```

Ejercicio 23: Definir una función que evalúe la función gaussiana 1D en un intervalo dado.

$$f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\left(\frac{(x-\mu)^2}{2\sigma^2}\right)}$$

La salida debería ser algo parecido a lo siguiente:

```
Tabla función normal (x,y):
 [[-2.     0.054]
 [-1.6    0.111]
 [-1.2    0.194]
 [-0.8    0.29 ]
 [-0.4    0.368]
 [ 0.     0.399]
 [ 0.4    0.368]
 [ 0.8    0.29 ]
 [ 1.2    0.194]
 [ 1.6    0.111]
 [ 2.     0.054]]
```

In [32]:
```
def f_normal(x, mu=0, sigma=1.0):
    '''
    Función normal 1D.
    Params:
       x, array con valores de x donde evaluar la función.
       mu, la media de la distribución.
       sigma, la desviación típica de la distribución.
    Retorna:
       array con valores de la función para cada valor de x.
    '''
    y=np.zeros_like(x) #crea un array con la misma forma y tipo que x.
    #Pon tu código aquí
    #Sugerencia: usa el argumento 'out' en las ufuncs para mejorar la eficiencia.
    #No uses bucles, solo ufuncs.

    #
    return y

X=np.linspace(-2, 2, 11)
Y= f_normal(X, 0, 1)
print('Tabla función normal (x,y):\n', np.stack((X,Y), axis=1))
```

```
Tabla función normal (x,y):
 [[-2.   0. ]
 [-1.6  0. ]
 [-1.2  0. ]
 [-0.8  0. ]
 [-0.4  0. ]
 [ 0.   0. ]
 [ 0.4  0. ]
 [ 0.8  0. ]
 [ 1.2  0. ]
 [ 1.6  0. ]
 [ 2.   0. ]]
```

Ejercicio 24: Definir una función que evalúe la función gaussiana 2D en un intervalo dado.

$$f(x,y) = \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2(1-\rho^2)}\left(\frac{x^2}{\sigma_x^2} + \frac{y^2}{\sigma_y^2} - \frac{2\rho x y}{(\sigma_x\sigma_y)}\right)\right)$$

La salida debería ser algo parecido a lo siguiente:

```
Tabla función normal 2D:
 [[0.003 0.006 0.01  0.016 0.02  0.022 0.02  0.016 0.01  0.006 0.003]
 [0.006 0.012 0.022 0.032 0.041 0.044 0.041 0.032 0.022 0.012 0.006]
 [0.01  0.022 0.038 0.056 0.072 0.077 0.072 0.056 0.038 0.022 0.01 ]
 [0.016 0.032 0.056 0.084 0.107 0.116 0.107 0.084 0.056 0.032 0.016]
 [0.02  0.041 0.072 0.107 0.136 0.147 0.136 0.107 0.072 0.041 0.02 ]
 [0.022 0.044 0.077 0.116 0.147 0.159 0.147 0.116 0.077 0.044 0.022]
 [0.02  0.041 0.072 0.107 0.136 0.147 0.136 0.107 0.072 0.041 0.02 ]
 [0.016 0.032 0.056 0.084 0.107 0.116 0.107 0.084 0.056 0.032 0.016]
 [0.01  0.022 0.038 0.056 0.072 0.077 0.072 0.056 0.038 0.022 0.01 ]
 [0.006 0.012 0.022 0.032 0.041 0.044 0.041 0.032 0.022 0.012 0.006]
 [0.003 0.006 0.01  0.016 0.02  0.022 0.02  0.016 0.01  0.006 0.003]]
```

In [33]:

```python
def f_normal_2d(x, y, mu_x=0.0, mu_y=0.0, sigma_x=1.0, sigma_y=1.0, rho=0.0 ):
    '''
    Función normal 2D.
    Params:
       x, y  grid con valores de x,y del plano donde evaluar la función.
       mu_x, sigma_x, la media y desviación en el eje X de la distribución.
       mu_y, sigma_y, la media y desviación en el eje Y de la distribución.
       rho, coeficiente de correlación de los ejes X,Y.
    Retorna:
       matriz con valores de la función evaluada para cada par (x,y) del grid.
    '''
    z=np.zeros_like(x) #crea un array con la misma forma y tipo que x.

    #Pon tu código aquí
    #Sugerencia: usa el argumento 'out' en las ufuncs para mejorar la eficiencia.
    #No uses bucles, solo ufuncs.

    #

    return z

X, Y = np.meshgrid(np.linspace(-2, 2, 11), np.linspace(-2, 2, 11))
Z= f_normal_2d(X, Y, 0.0, 0.0, 1.0, 1.0, 0.0)
print('Tabla función normal 2D:\n', Z)
```

```
Tabla función normal 2D:
 [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

In [ ]: