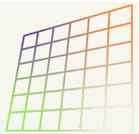
1803

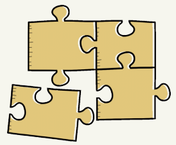
INFORMÁTICA II

Bioingeniería

Índice



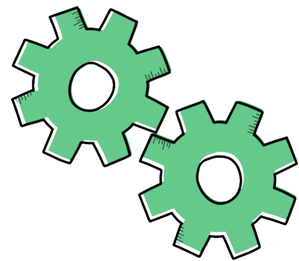
Unidad 2

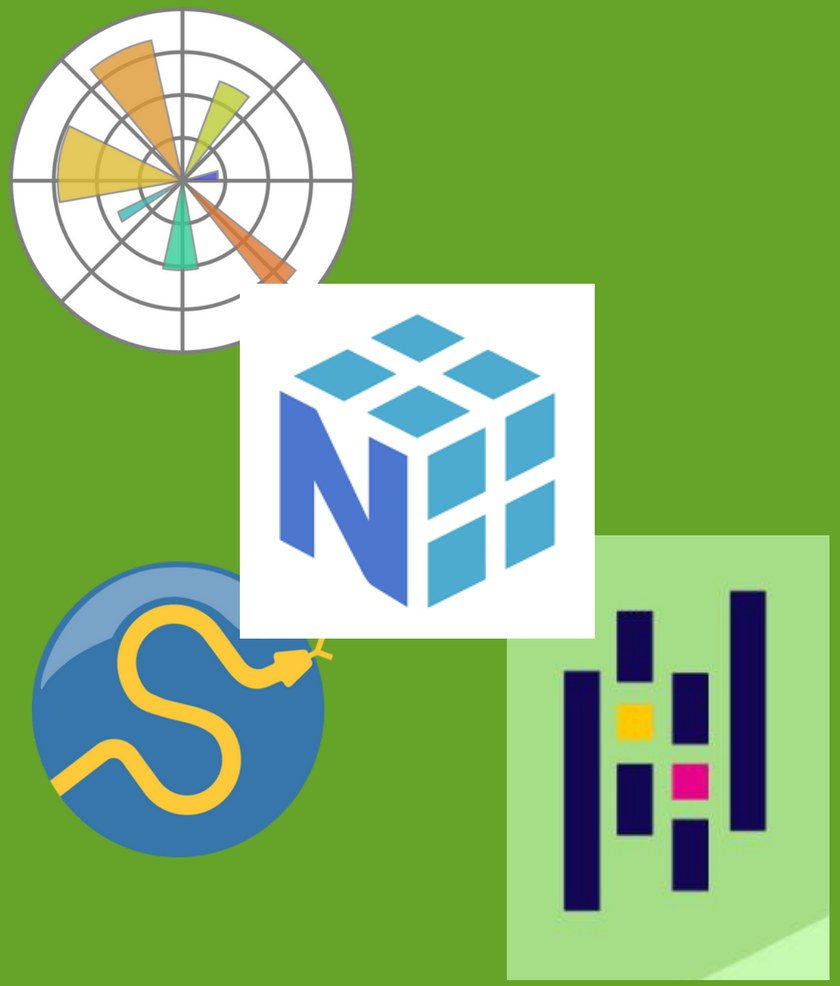


Arrays (matrices)

Indexación

Operaciones

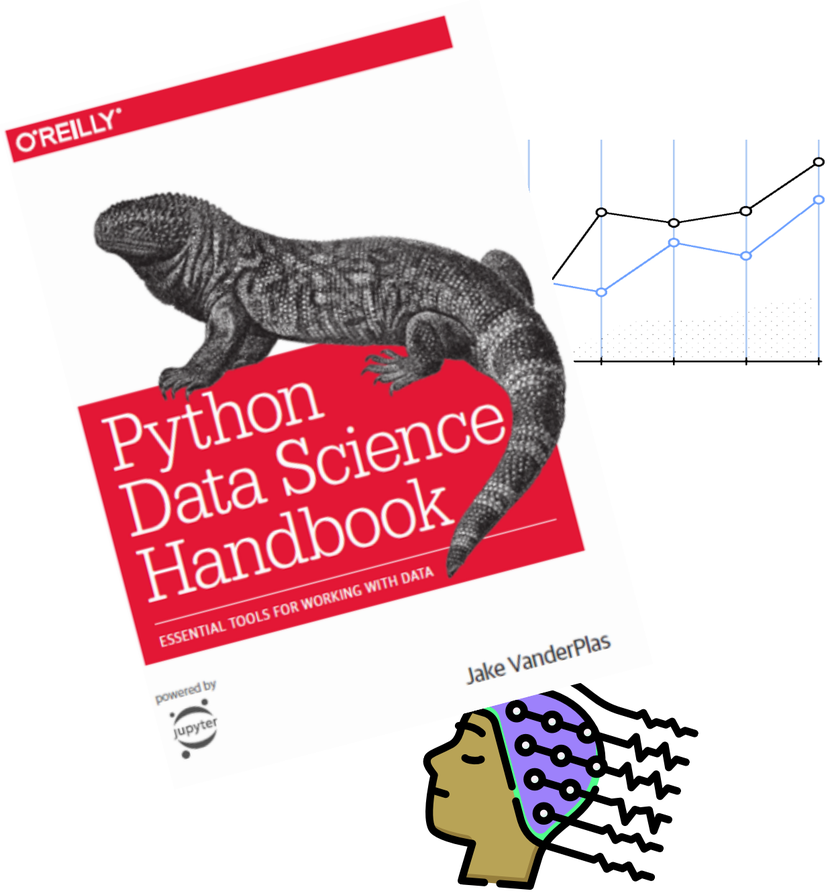




Unidad 2

Introducción a la computación en Bioingeniería

Introducción a Numpy

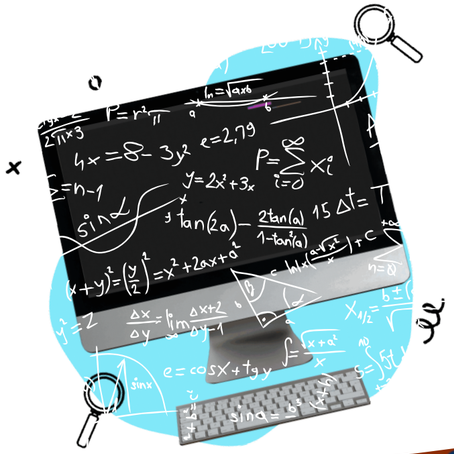


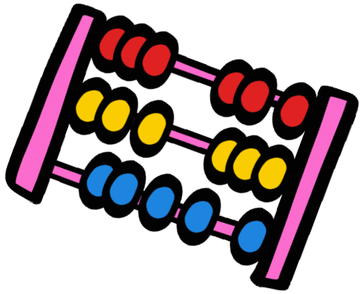
- No importa cuáles sean los datos, el primer paso para hacerlos analizables será transformarlos en matrices (arrays) de números.

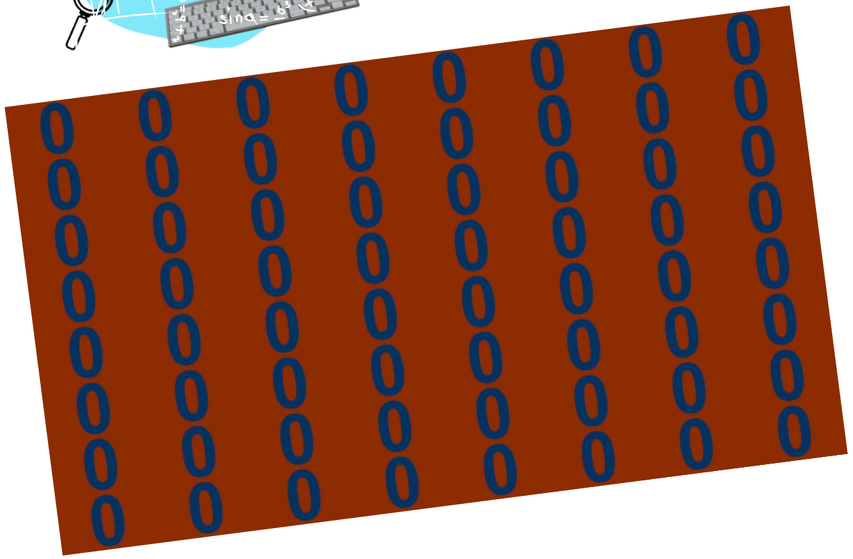
- El almacenamiento y la manipulación eficientes de matrices numéricas son absolutamente fundamentales para el proceso de hacer ciencia de datos.

Pagina oficial: **https://numpy.org/**

Introducción a Numpy



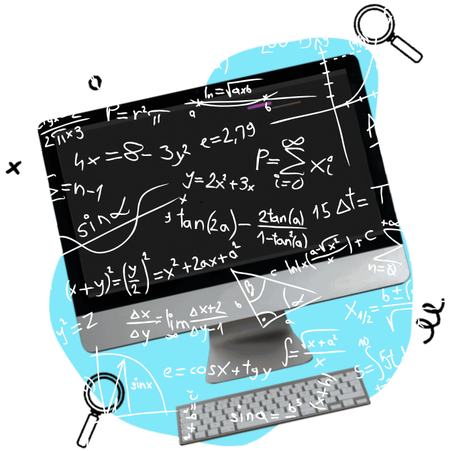




- NumPy (Numerical Python) proporciona una interfaz eficiente para almacenar y operar en búferes de datos densos.

- De alguna manera, las matrices NumPy son como el tipo de lista incorporada de Python, pero las matrices NumPy proporcionan **operaciones de almacenamiento y manipulación** de datos mucho más eficientes a medida que las matrices crecen en tamaño.

Introducción a Numpy



- Es un paquete para contener datos **multidimensionales homogéneos** y realizar funciones con los mismos, el uso de estos en algunos casos es igual que las listas de python.

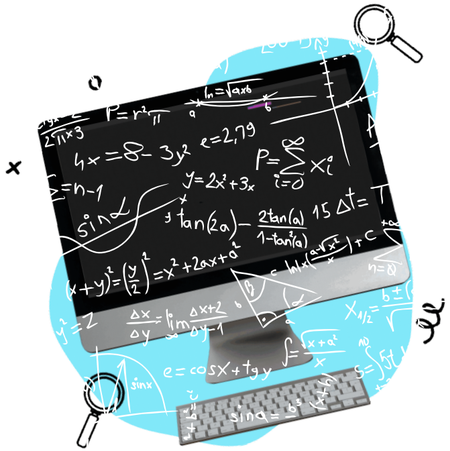
- Creamos un alias para acceder a las funciones del paquete:

```
>>> import numpy as np
>>> print(np.__version__)
```

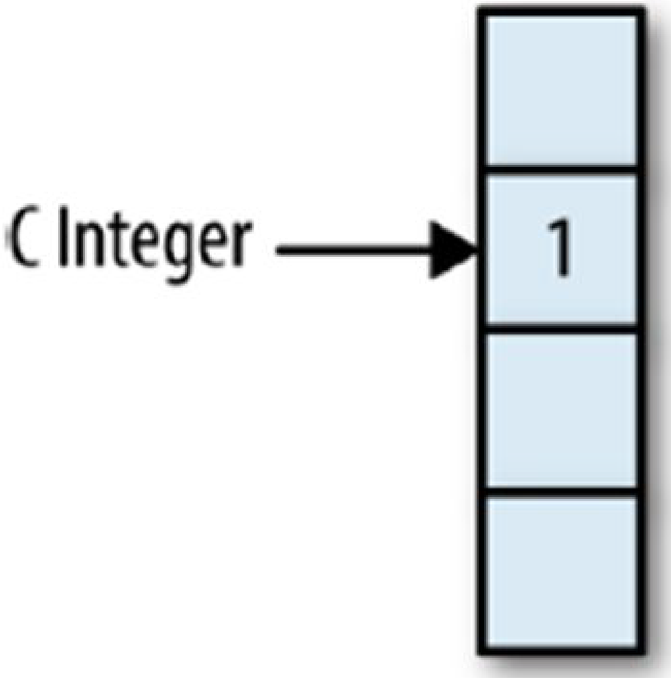
- Para obtener información:

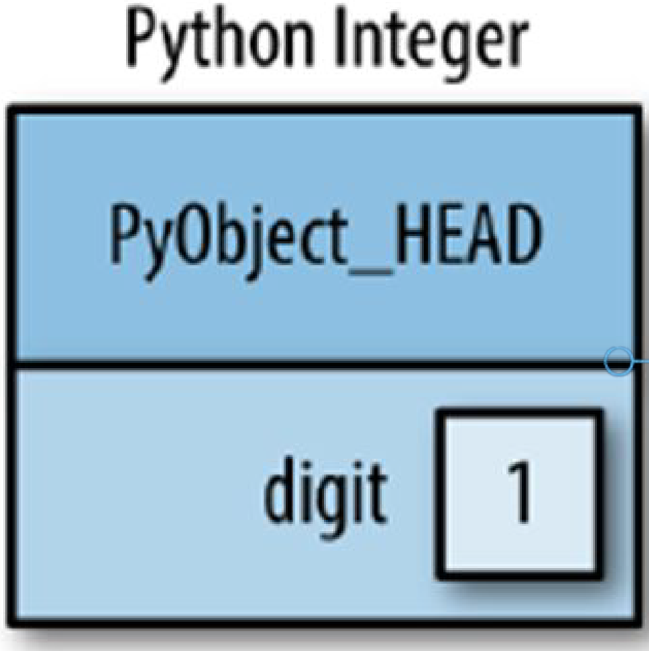
```
>>> np.lookfor('solve')
```

Tipos de datos en Numpy



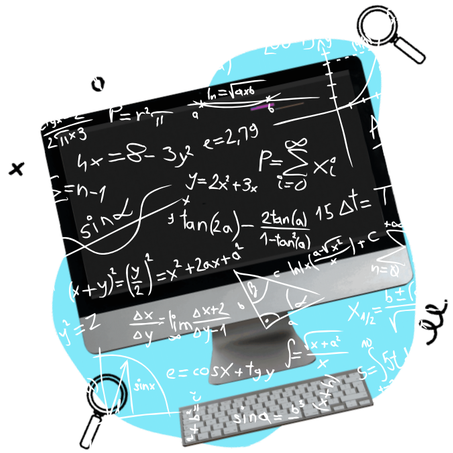
Como ya sabemos , Python es un lenguaje de programación orientado a objetos.

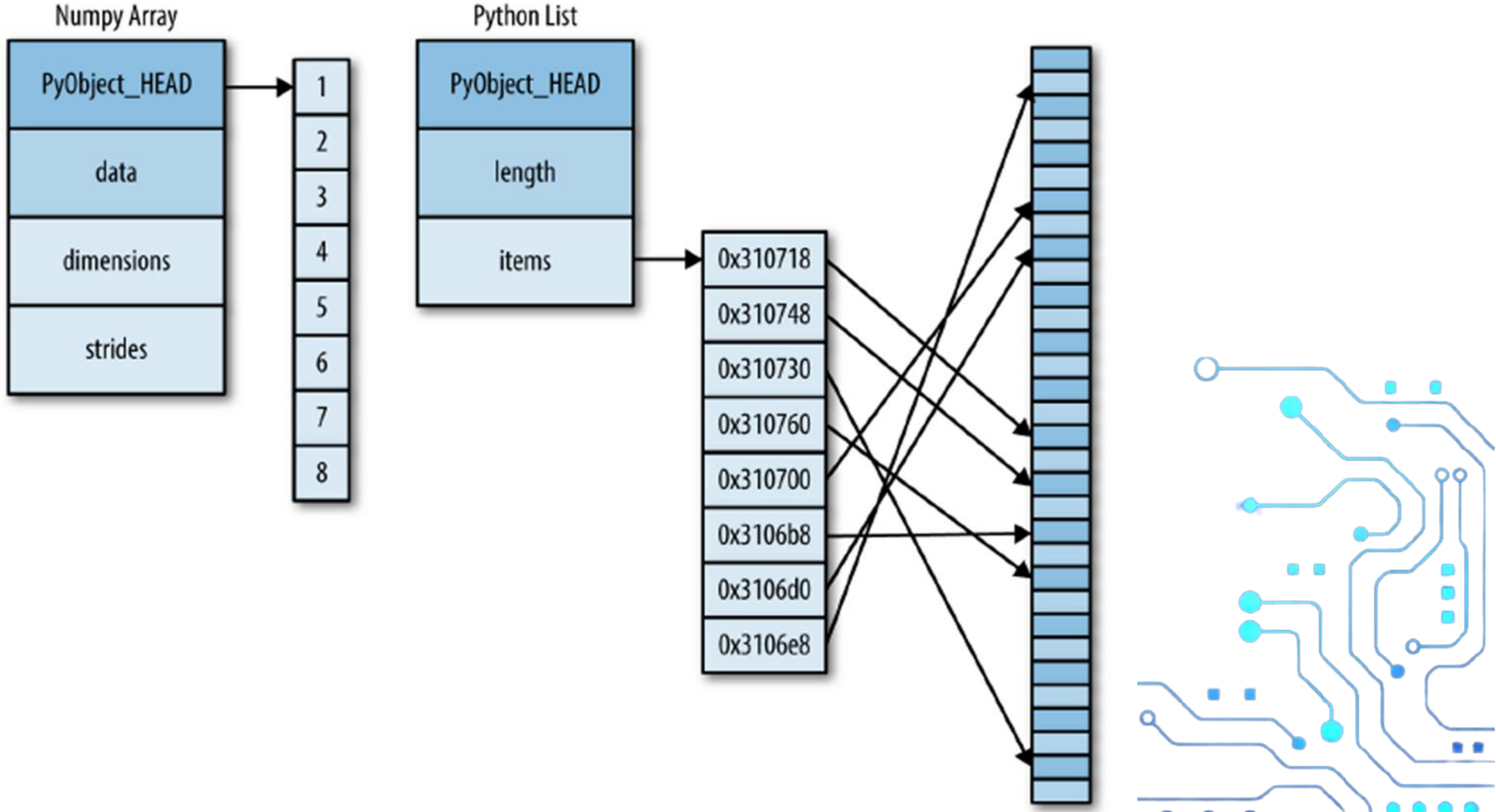




Tipos de datos en Numpy

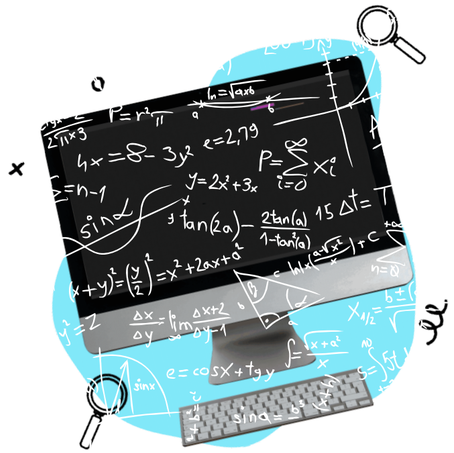
Como ya sabemos , Python es un lenguaje de programación orientado a objetos.



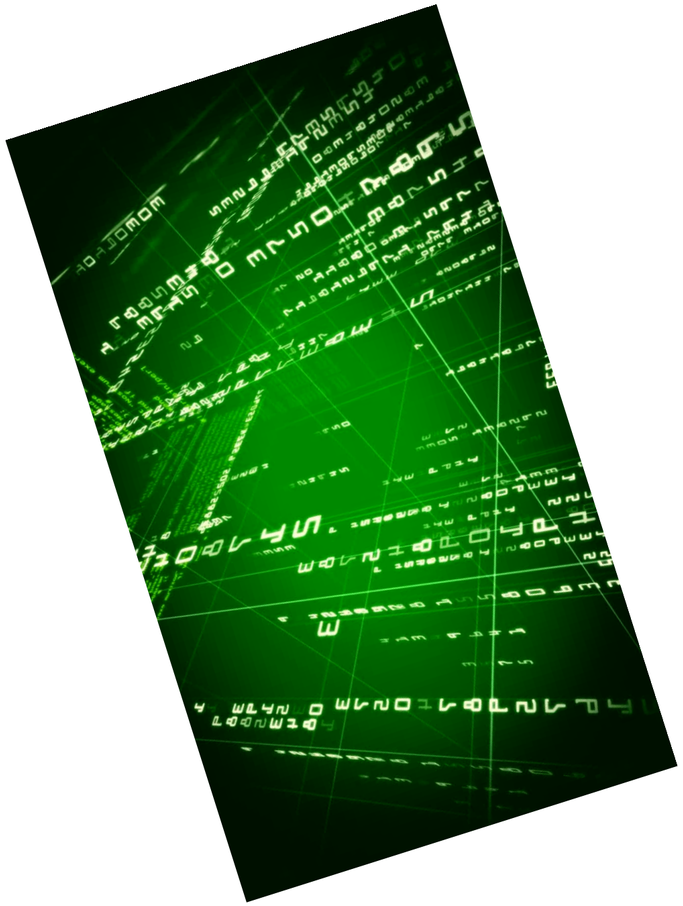


Tipos de datos en Numpy

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (–128 to 127)
`int16`	Integer (–32768 to 32767)
`int32`	Integer (–2147483648 to 2147483647)
`int64`	Integer (–9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`
`float16`	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats



Crear arrays



Para crear arreglos (matrices) desde python podemos usar las listas de python que ya conocemos.

```
>>> np.array([1,2,3,5,7,12])
```

NumPy está restringido a matrices que contienen **el mismo tipo.** Si los tipos no coinciden, NumPy los actualizará si es posible, es decir, los lleva a un mismo tipo de dato (como los mostrados en la tabla anterior)

Matrices (Arrays) en Numpy

Un array(arreglo o matriz) es una colección de N elementos

>> np.array(object, dtype=None)

Parámetros:

object: una colección de elementos

dtype: Tipo de salida del arreglo (opcional).

Si dtype no es dado, dtype='float'

```
In [1]: import numpy as np

In [2]: np.array([1, 2, 3])
Out[2]: array([1, 2, 3])

In [3]: np.array([1,2.,3])
Out[3]: array([ 1.,  2.,  3.])

In [4]: np.array([1,2,3], dtype=float)
Out[4]: array([ 1.,  2.,  3.])

In [5]: np.array([1,2,3], dtype=complex)
Out[5]: array([ 1.+0.j,  2.+0.j,  3.+0.j])

In [6]: a=np.array([1, 2, 3])

In [7]: a.astype('int')
Out[7]: array([1, 2, 3])
```

Matrices (Arrays) en Numpy


Para matrices más grandes, es más eficiente crear matrices desde cero utilizando **rutinas integradas** de NumPy.

Ejemplos:

```
>> np.zeros (157, dtype = ‘int’)
```

Matriz con ciento cincuentaisiete entradas en cero

```
>> np.ones ((2,13), dtype = ‘float’)
```

Matriz con , dos filas, trece columnas ,todas las entradas con valor de uno tipo flotantes

```
>> np.full ((2,19), 3.1416)
```

Matriz con, dos filas, diecinueve columnas, todas las entradas con el valor de 3.1416

Matrices (Arrays) en Numpy

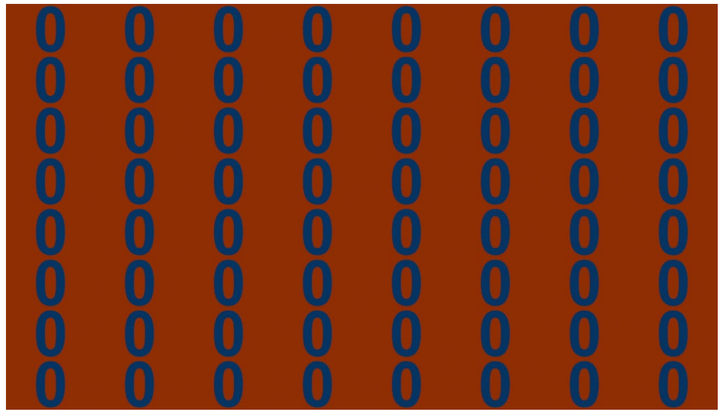
>> np.zeros(shape, dtype)

Parámetros:

Shape: Entero o secuencia de enteros

dtype: Tipo de salida del arreglo (opcional). Si dtype no es dado, dtype='float'

Retorna: Matriz de ceros de tamaño dado



Matrices (Arrays) en Numpy

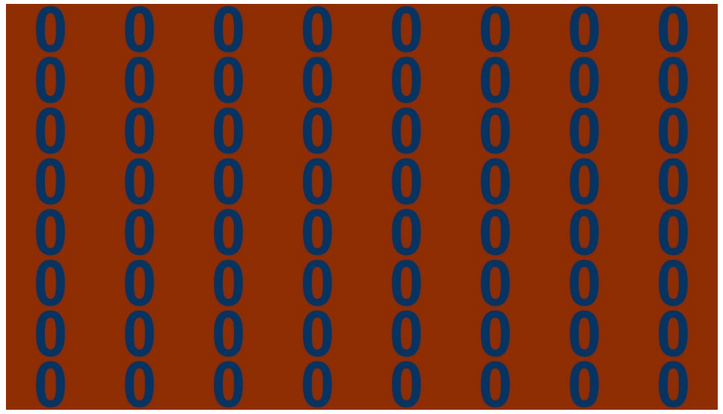
>> np.ones(shape, dtype)(shape, dtype)

Parámetros:

Shape: Entero o secuencia de enteros

dtype: Tipo de salida del arreglo (opcional). Si dtype no es dado, dtype='float'

Retorna: Matriz de unos de tamaño dado



Matrices (Arrays) en Numpy

>> np.identity(n, dtype)

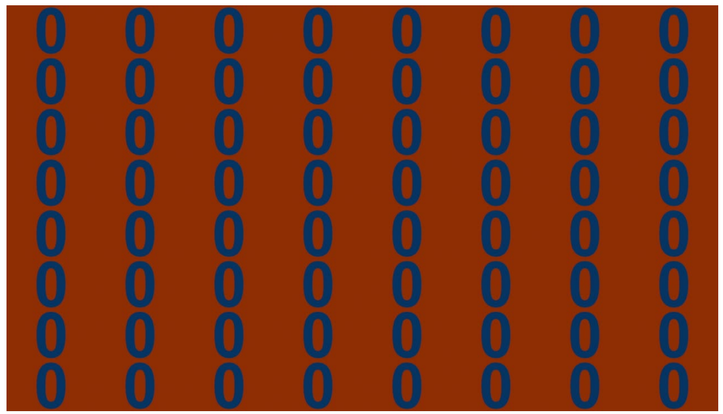
Parámetros:

n: Entero

dtype: Tipo de salida del arreglo (opcional).

Si dtype no es dado, dtype='float'

Retorna: Matriz identidad de tamaño *n x n*



```
In [6]: import numpy as np

In [7]: np.zeros([1,3])
Out[7]: array([[ 0.,  0.,  0.]])

In [8]: np.ones([3,4],dtype=int)
Out[8]:
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])

In [9]: np.identity(4)
Out[9]:
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

Matrices (Arrays) en Numpy

>> np.full(n,numero, dtype)

Parámetros:

n: Entero

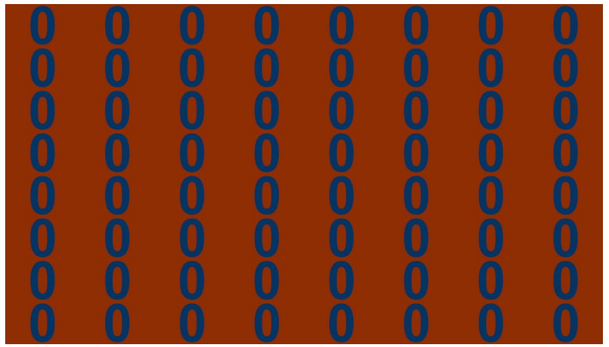
dtype: Tipo de salida del arreglo (opcional). Si dtype no es dado, dtype='float'

Retorna: Matriz de una forma determinada llena de elementos especificos

```
import numpy as np

np.full((12,2),4.32,"float32")
```

✓ 0.7s

```
array([[4.32, 4.32],
       [4.32, 4.32],
       [4.32, 4.32],
       [4.32, 4.32],
       [4.32, 4.32],
       [4.32, 4.32],
       [4.32, 4.32],
       [4.32, 4.32],
       [4.32, 4.32],
       [4.32, 4.32],
       [4.32, 4.32],
       [4.32, 4.32]], dtype=float32)
```



Matrices (Arrays) en Numpy

>>np.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None)

Retorna: Los números espaciados uniformemente en una escala logarítmica.
En el espacio lineal, la secuencia comienza en la base ** start (base a la potencia de inicio) y finaliza con la base ** stop (ver punto final a continuación).

Parámetros:
start: El valor de inicio de la secuencia (float)
stop: El valor de parada de la secuencia (float)
num: Número de muestras a generar (opcional)
endpoint: Si es verdadero, “stop” es el valor de la última muestra, en otro caso, no está incluida (opcional)
base: Base logaritmica (opcional)
dtype: Tipo de salida del arreglo (opcional). Si dtype no es dado, se infiere el tipo de dato de acuerdo a los valores de entrada

Matrices (Arrays) en Numpy


>>>np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)

Retorna: números espaciados uniformemente en un intervalo especificado.
El punto final del intervalo puede excluirse opcionalmente.

Parámetros:
start: El valor de inicio de la secuencia (Escalar)
stop: El valor de parada de la secuencia (Escalar)
num: Número de muestras a generar (opcional)
endpoint: Si es verdadero, “stop” es el valor de la última muestra, en otro caso, no está incluida (opcional)
retstep: Si es verdadero, retorna (muestras, espacio entre las muestras) (opcional)
dtype: Tipo de salida del arreglo (opcional). Si dtype no es dado, se infiere el tipo de dato de acuerdo a los valores de entrada

Matrices (Arrays) en Numpy

```
In [10]: np.linspace(0,1,10,dtype=complex)
Out[10]:
array([ 0.00000000+0.j,  0.11111111+0.j,  0.22222222+0.j,  0.33333333+0.j,
        0.44444444+0.j,  0.55555556+0.j,  0.66666667+0.j,  0.77777778+0.j,
        0.88888889+0.j,  1.00000000+0.j])

In [11]: np.logspace(0,3,20)
Out[11]:
array([    1.        ,     1.43844989,     2.06913808,     2.97635144,
           4.2813324 ,     6.15848211,     8.8586679 ,    12.74274986,
          18.32980711,    26.36650899,    37.92690191,    54.55594781,
          78.47599704,   112.88378917,   162.37767392,   233.57214691,
         335.98182863,   483.29302386,   695.19279618,  1000.        ])

In [12]: np.linspace(0,10,10,dtype=int)
Out[12]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8, 10])

In [13]: np.logspace(1,2,2)
Out[13]: array([  10.,  100.])
```

Matrices (Arrays) en Numpy

>> np.random.random(shape) ó random.random_sample (shape)

Retorna: números aleatorios con una forma determinada, entre 0 y 1

Parámetros:

shape: forma de la matriz (array)

>> np.random.randint (low, high=None, size=None, dtype='l')

ó

>> np.random.random_integer (low, high=None, size=None)

Retorna: números aleatorios enteros con una forma determinada entre low y high especificados excluyendo el numero high.

Parámetros:

low: número inicial (incluido)

high: númeto final (excluido)

size: tamaño del arreglo

dtype: tipo de datos del arreglo

Matrices (Arrays) en Numpy

>> np.arange(start, stop, step, dtype)

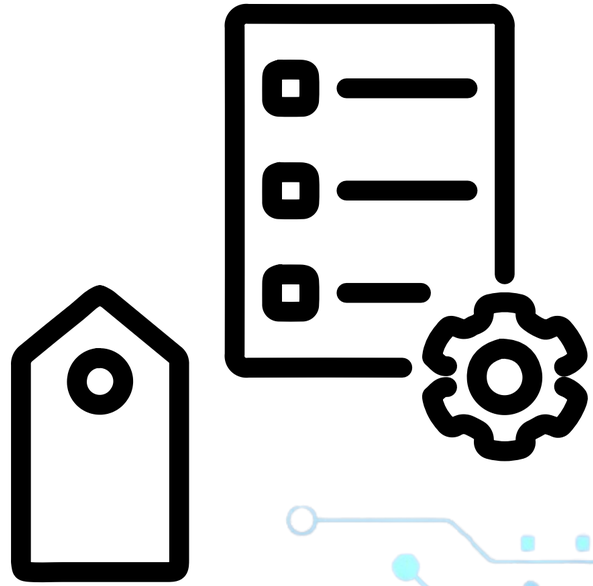
Retorna: Matriz con un valor inicial, final y un paso

Parámetros:

start: El valor de inicio (Número) (opcional)
stop: El valor de parada (Número)
step: Espacio entre los valores (opcional). Por defecto este valor es 1.
dtype: Tipo de salida del arreglo (opcional). Si dtype no es dado, se infiere el tipo de dato de acuerdo a los valores de entrada

Atributos de Matriz (Array)

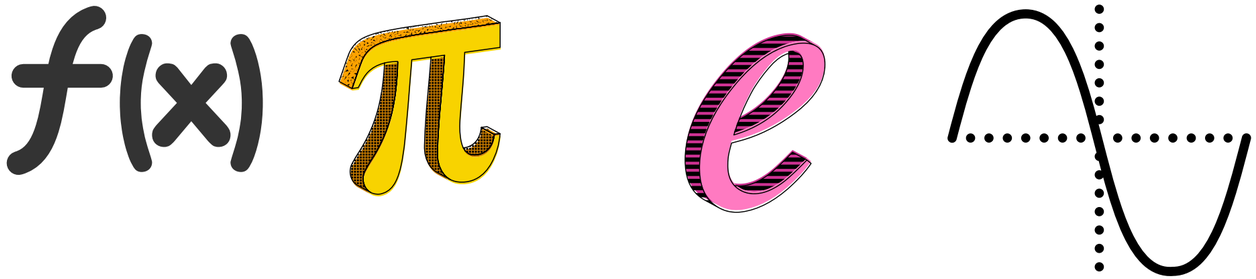
```
>> np.random.randint(seed, size=(x, y, z))

                ó

>>np.random.randint(low, high=None, size=None, dtype=int)

# Atributos

>> array_example = np.random.randint(10, size=(3, 4, 5))

>> print("Número de dimensiones: ", array_example.ndim)

>> print("Forma: ", array_example.shape)

>> print("Tamaño: ", array_example.size)

>> print("Tipo: ", array_example.dtype)

>> print("Tamaño en bytes de cada elemento del array: ", array_example.itemsize)

>> print("Tamaño total en bytes: ",array_example.nbytes)
```



Constantes y funciones matemáticas

- Constantes
  - **>> np.pi**
  - **>> np.e**
- Funciones matemáticas
  - **>>np.log**
  - **>>np.sin**
  - **>>np.cos**

```
IPython 6.1.0 -- An enhanced Interactive Python.

In [1]: import numpy as np

In [2]: np.sin(90)
Out[2]: 0.8939966636005785

In [3]: np.sin(np.radians(90))
Out[3]: 1.0

In [4]: np.cos(np.radians(90))
Out[4]: 6.123233995736766e-17
```



Constantes y funciones matemáticas

>> np.sin(x), >> np.cos(x), >> np.tan(x)

Parámetros

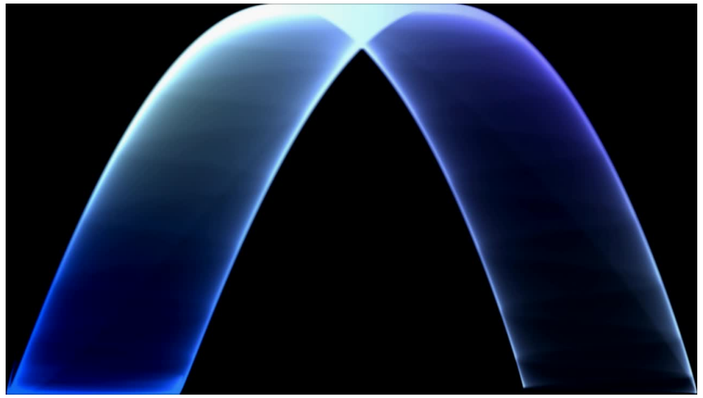
x: Ángulo en radianes

Retorna: La función correspondiente (seno, coseno..) de cada elemento de x

degrees(x): Convierte ángulos de radianes a grados

radians(x): Convierte ángulos de grados a radianes

$f(x)$



Numpy remuestreo

Construcción de arreglo de un número determinado de elementos y **remuestreo.**

```
In [1]: import numpy as np

In [2]: a=np.empty(5000)

In [3]: a
Out[3]:
array([  1.84577787e-317,   9.21460413e-316,   0.00000000e+000, ...,
         0.00000000e+000,   0.00000000e+000,   0.00000000e+000])

In [4]: a.reshape(50,100)
Out[4]:
array([[  1.84577787e-317,   9.21460413e-316,   0.00000000e+000, ...,
          0.00000000e+000,   0.00000000e+000,   0.00000000e+000],
       [  0.00000000e+000,   0.00000000e+000,   0.00000000e+000, ...,
          0.00000000e+000,   0.00000000e+000,   0.00000000e+000],
       [  0.00000000e+000,   0.00000000e+000,   0.00000000e+000, ...,
          0.00000000e+000,   0.00000000e+000,   0.00000000e+000],
       ...,
```

Numpy remuestreo

>> np.empty(shape, dtype)

Parámetros:

- **shape:** Entero, o tupla de enteros
- **dtype:** Tipo de salida del arreglo (opcional). Si dtype no es dado,dtype='float'
- **Retorna:** Un arreglo del tamaño dado y sin inicializar las entradas

NOTA: La función crea un array sin inicializar sus elementos, lo que significa que el contenido del array será impredecible y dependerá del estado de la memoria en el momento de la creación del array. Esta función no establece los valores de la matriz a cero y requiere que el usuario configure manualmente todos los valores en la matriz.

Numpy remuestreo

>> np.reshape(a, newshape)

Parámetros:

a: Arreglo

newshape: (int o tuplas de int) Nueva forma del arreglo. Debe ser compatible con el original.

>> values = np.arange(1, 10) matriz de nueve valores

>> grid = values.reshape((3, 3)) Usando el método de ndarray

>> print(grid)

>> print(id(values))

>> print(id(grid))

Numpy remuestreo

NOTA: Tenga en cuenta que el tamaño de la matriz inicial debe coincidir con el tamaño de la matriz remodelada. Siempre que sea posible, el método reshape utilizará una vista sin copia de la matriz inicial

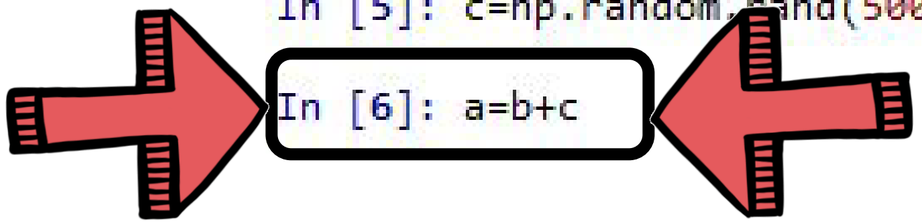
Forma alternativa para usar *reshape* desde la librería de numpy

```
>> values = np.arange(1, 10)
>> grid = np.reshape(values,(3, 3), order = ‘F’) # Usando la función de numpy
>> print(grid);
>> print(id(values));
>> print(id(grid));
```

Operaciones entre arreglos

Una de las grandes ventajas y por ende optimizaciones que trae consigo la librería **Numpy** es trabajar de manera matricial entre arreglos ,y realizar operaciones aritméticas, logaritmicas, lógicas, trigonometricas, etc

```
In [1]: import numpy as np

In [2]: n,m=50,100

In [3]: a=np.empty(5000).reshape(n,m)

In [4]: b=np.random.rand(5000).reshape(n,m)

In [5]: c=np.random.rand(5000).reshape(n,m)

In [6]: a=b+c
```



Indexación de arreglos

La indexación en NumPy es similar a la **indexación** en listas de Python. Para acceder a un elemento o conjunto de elementos de un array NumPy, se utiliza la notación de corchetes **([]).**

nombre_matriz[fila,columna]

nombre_matriz[fila,:]

nombre_matriz[fila,::n]

nombre_matriz[n1:n2,n3:n4]

nombre_matriz[nombre_matriz>2]

```
In [11]: b=np.array([[1,2,3,4,5,6,7],[24,56,23,45,67,98,10]])
In [13]: b[1,0:3]
Out[13]: array([24, 56, 23])
```

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

In [3]: a[0,0]
Out[3]: 1

In [4]: a[1,:]
Out[4]: array([4, 5, 6])

In [5]: a[:,0]
Out[5]: array([1, 4, 7])

In [6]: a[0,::2]
Out[6]: array([1, 3])

In [7]: a[0,::1]
Out[7]: array([1, 2, 3])

In [8]: a[1:2,0:2]
Out[8]: array([[4, 5]])
```

Indexación de arreglos, vistas sin copia

- Los cortes de matrices (slicing) retornas vistas de dicho corte, **más no** una copia de los datos
- Es decir, si modificamos la submatriz se cambia la **matriz original !!**
- Este comportamiento predeterminado es realmente bastante útil: cuando trabajamos con grandes conjuntos de datos, podemos acceder y procesar partes de estos conjuntos de datos sin la necesidad de copiar el búfer de datos subyacente.
- Para obtener copias , usamos el metodo *copy()*

```
>>> array_example2 = array_example[1:2: , 2:4: , ::-1].copy()
```

https://interactivechaos.com/es/manual/tutorial-de-numpy/indexado-y-seleccion

Cocatenación

- La concatenación, o unión de dos matrices en NumPy, se logra principalmente a través de las rutinas **np.concatenate, np.vstack y np.hstack**.
- **np.concatenate** toma una tupla o una lista de matrices como su primer argumento

```
>> x = np.array([1, 2, 3])
>> y = np.array([3, 2, 1])
>> np.concatenate([x, y])
```

Cocatenación vertical y horizontal


Al trabajar con matrices de dimensiones mixtas, puede ser más claro usar las funciones **np.vstack** (pila vertical) y **np.hstack** (pila horizontal)

- **>> x = np.array([1, 2, 3])**
- **>> grid = np.array([[9, 8, 7], [6, 5, 4]])**
- **>> np.vstack([x, grid]) # vertically stack the arrays**
- **>> y = np.array([[99], [99]])**
- **>> np.hstack([grid, y]) # horizontally stack the arrays**
-

https://interactivechaos.com/es/manual/tutorial-de-numpy/arrays-de-una-dimension

Separación (splitting) vertical y horizontal


Lo opuesto a la concatenación es la división, que es implementada por las funciones **np.split**, **np.hsplit y np.vsplit**. Para cada uno de estos, podemos pasar una lista de índices dando los puntos de división:

- **>> x = [1, 2, 3, 99, 99, 3, 2, 1]**
- **>> x1, x2, x3 = np.split(x, [3, 5]) Start to (end – 1)**
- **>> print(x1, x2, x3) → [1 2 3] [99 99] [3 2 1]**

Separación (splitting) vertical y horizontal

- División vertical
  - >> grid = np.arange(16).reshape((4, 4))
  - >> upper, lower = np.vsplit(grid, [2])
  - >> print(upper)
  - >> print(lower)
  -
- División Horizontal
  - >> left, right = np.hsplit(grid, [2])
  - >> print(left)
  - >> print(right)

Operaciones elemento a elemento

También es posible realizar operaciones a los elementos de una arreglo , de forma indaxada o general.

```
In [32]: c=np.arange(6).reshape(2,3)

In [33]: np.sqrt(c)
Out[33]:
array([[ 0.        ,  1.        ,  1.41421356],
       [ 1.73205081,  2.        ,  2.23606798]])

In [34]: d=np.arange(-3,3)
```

```
In [35]: np.sqrt(d)
__main__:1: RuntimeWarning: invalid value encountered in sqrt
Out[35]:
array([       nan,        nan,        nan, 0.        , 1.        ,
       1.41421356])
```

```
In [39]: d=d.astype(complex)

In [40]: d
Out[40]: array([-3.+0.j, -2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j])
```

```
In [41]: np.sqrt(d)
Out[41]:
array([ 0.00000000+1.73205081j,  0.00000000+1.41421356j,
        0.00000000+1.j        ,  0.00000000+0.j        ,
        1.00000000+0.j        ,  1.41421356+0.j        ])
```

Operaciones elemento a elemento

También es posible realizar operaciones a los elementos de una arreglo , de forma indaxada o general.

```
In [32]: c=np.arange(6).reshape(2,3)

In [33]: np.sqrt(c)
Out[33]:
array([[ 0.        ,  1.        ,  1.41421356],
       [ 1.73205081,  2.        ,  2.23606798]])

In [34]: d=np.arange(-3,3)
```

```
In [35]: np.sqrt(d)
__main__:1: RuntimeWarning: invalid value encountered in sqrt
Out[35]:
array([       nan,        nan,        nan,  0.        ,  1.        ,
       1.41421356])
```

```
In [39]: d=d.astype(complex)

In [40]: d
Out[40]: array([-3.+0.j, -2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j])
```

```
In [41]: np.sqrt(d)
Out[41]:
array([ 0.00000000+1.73205081j,  0.00000000+1.41421356j,
        0.00000000+1.j        ,  0.00000000+0.j        ,
        1.00000000+0.j        ,  1.41421356+0.j        ])
```

UFUNC


- La clave para hacer que el cálculo sea rápido es utilizar operaciones vectorizadas, generalmente implementadas a través de las **funciones universales de NumPy (ufuncs).**
  - **>> np.arange(5) / np.arange(1, 6)**
  - **>> x = np.arange(9).reshape((3, 3))**
  - **>> print(2 ** x)**
- Los cálculos que utilizan vectorización a través de ufuncs son casi siempre más eficientes que su contraparte implementada a través de bucles de Python, especialmente a medida que las matrices crecen en tamaño.
- Cada vez que vea un bucle de este tipo en un script de Python, debe considerar si se puede reemplazar con una expresión vectorizada.

UFUNC

Arithmetic operators implemented in NumPy

Operator	Equivalent ufunc	Description
+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

UFUNC

Aggregation functions available in NumPy

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute median of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

Comparaciones con UFUNC


- NumPy también implementa operadores de comparación como **< (menor que) y > (mayor que) como ufuncs**

- El resultado de estos operadores de comparación es siempre una matriz con un tipo de datos booleano.

- En cuanto a los operadores aritméticos, los operadores de comparación se implementan como ufuncs en NumPy (cuando se escribe x < 3, **internamente NumPy** usa **np.less(x, 3)).**

Comparaciones con UFUNC

Operator	Equivalent ufunc
==	np.equal
!=	np.not_equal
<	np.less
<=	np.less_equal
>	np.greater
>=	np.greater_equal

Comparaciones con UFUNC

```
In[4]: x = np.array([1, 2, 3, 4, 5])

In[5]: x < 3  # less than

Out[5]: array([ True,  True, False, False, False], dtype=bool)

In[6]: x > 3  # greater than

Out[6]: array([False, False, False,  True,  True], dtype=bool)

In[7]: x <= 3  # less than or equal

Out[7]: array([ True,  True,  True, False, False], dtype=bool)

In[8]: x >= 3  # greater than or equal

Out[8]: array([False, False,  True,  True,  True], dtype=bool)

In[9]: x != 3  # not equal

Out[9]: array([ True,  True, False,  True,  True], dtype=bool)

In[10]: x == 3  # equal

Out[10]: array([False, False,  True, False, False], dtype=bool)
```

Comparaciones con UFUNC


Al igual que en el caso de los ufuncs aritméticos, estos funcionarán en matrices de cualquier tamaño y forma.

```
In[12]: rng = np.random.RandomState(0)
        x = rng.randint(10, size=(3, 4))
        x

Out[12]: array([[5, 0, 3, 3],
                [7, 9, 3, 5],
                [2, 4, 7, 6]])

In[13]: x < 6

Out[13]: array([[ True,  True,  True,  True],
                [False, False,  True,  True],
                [ True,  True, False, False]], dtype=bool)
```

Comparaciones con UFUNC

Para contar el número de entradas True en una matriz booleana

```
In[16]: np.sum(x < 6)

Out[16]: 8
```

Contar en una dimensión específica:

```
In[17]: # how many values less than 6 in each row?
        np.sum(x < 6, axis=1)

Out[17]: array([4, 2, 2])
```

Any y All

Algún valor es verdadero ?

```
In[18]: # are there any values greater than 8?
        np.any(x > 8)

Out[18]: True

In[19]: # are there any values less than zero?
        np.any(x < 0)

Out[19]: False

In[20]: # are all values less than 10?
        np.all(x < 10)

Out[20]: True

In[21]: # are all values equal to 6?
        np.all(x == 6)

Out[21]: False
```

Comparaciones entre elementos de arrays

Operaciones elemento a elemento.

```
In [43]: a=np.arange(6)

In [44]: b=np.ones(6,dtype=int)

In [45]: a,b
Out[45]: (array([0, 1, 2, 3, 4, 5]), array([1, 1, 1, 1, 1, 1]))

In [46]: a<b
Out[46]: array([ True, False, False, False, False, False], dtype=bool)

In [47]: np.any(a<b)
Out[47]: True

In [48]: np.all(a<b)
Out[48]: False
```

Funciones sum(), any() y all()


- Python tiene funciones sum(), any() y all() integradas.

- Estos tienen una sintaxis diferente a las versiones de NumPy y, en particular, fallarán o producirán resultados no deseados cuando se usen en matrices multidimensionales.

- ¡Asegúrese de que está utilizando **np.**sum(), **np.**any() y **np.**all() para estos ejemplos!

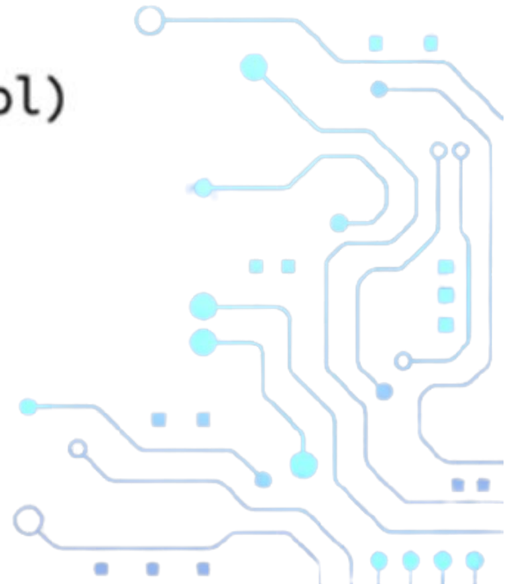
Operadores UFUNC para operaciones bit a bit

Operator	Equivalent ufunc
&	np.bitwise_and
|	np.bitwise_or
^	np.bitwise_xor
~	np.bitwise_not

Mascaras

```
In[26]: x

Out[26]: array([[5, 0, 3, 3],
                [7, 9, 3, 5],
                [2, 4, 7, 6]])

In[27]: x < 5

Out[27]: array([[False,  True,  True,  True],
                [False, False,  True, False],
                [ True,  True, False, False]], dtype=bool)
```

```
In[28]: x[x < 5]

Out[28]: array([0, 3, 3, 3, 2, 4])
```



Operdores booleanos and/or vs | / &

```
In[39]: x = np.arange(10)
        (x > 4) & (x < 8)

Out[39]: array([False, False, ...,  True,  True, False, False], dtype=bool)
```

```
In[40]: (x > 4) and (x < 8)

---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-40-3d24f1ffd63d> in <module>()
----> 1 (x > 4) and (x < 8)


ValueError: The truth value of an array with more than one element is...
```

Isclose y allclose

Las funciones isclose() y allclose() realizan comparaciones entre arreglos especificando una tolerancia.

```
In [50]: a=np.arange(6).astype(float)

In [51]: b=np.ones(6)

In [52]: a,b
Out[52]: (array([ 0.,  1.,  2.,  3.,  4.,  5.]), array([ 1.,  1.,  1.,  1.,  1.,  1.]))

In [53]: np.isclose(a,b)
Out[53]: array([False,  True, False, False, False, False], dtype=bool)

In [54]: np.allclose(a,b)
Out[54]: False
```

Isclose y allclose

>>> isclose(a, b, rtol=1e-05, atol=1e-08)

Parámetros:

a, b: Arreglos a comparar

rtol: Tolerancia relativa (float)

atol: Tolerancia absoluta(float)

Retorna: Una matriz booleana

>>> allclose(a, b, rtol=1e-05, atol=1e-08)

Parámetros:

a, b: Arreglos a comparar

rtol: Tolerancia relativa (float)

atol: Tolerancia absoluta(float)

Retorna: True, si todos los elementos de ambas matrices son iguales teniendo en cuenta la tolerancia

Ejercicios

1. Crear una matriz de ceros de tipo entero 3x4.
2. Crear una matriz de ceros de tipo entero 3x4 excepto la primera fila que será uno.
3. Crear una matriz de ceros de tipo entero 3x4 excepto la última fila que será el rango entre 5 y 8.
4. Crea un vector de 10 elementos, siendo los índices impares unos y los índices pares dos.
5. Crea un «tablero de ajedrez», con unos en las casillas negras y ceros en las blancas.
6. Crea una matriz aleatoria 5x5 y halla los valores mínimo y máximo.
7. Normalizar la matriz anterior

Referencias

https://docs.scipy.org/doc/numpy-dev/user/quickstart.html

http://cs231n.github.io/python-numpy-tutorial/

GRACIAS