NumPy

¿Qué es NumPy?

- NumPy (abreviatura de Python numérico) proporciona una interfaz eficiente para almacenar y operar con grandes cantidades de datos.
- Las matrices y vectores de NumPy son como el tipo ‘list’ incorporada de Python, pero las matrices NumPy proporcionan operaciones de almacenamiento y datos mucho más eficientes a medida que las matrices crecen en tamaño.
- Las matrices NumPy forman el núcleo de casi todo el ecosistema de herramientas de ciencia de datos en Python (pandas, SciPy, etc.) , por lo que aprender a usar NumPy será valioso sin importar en qué ámbito científico o técnico se trabaje.

Datos en Python

- La implementación estándar de Python está escrita en C. Eso significa que cualquier tipo de dato tiene una representación como una estructura de C. Si queremos crear en C un número entero grande haríamos:

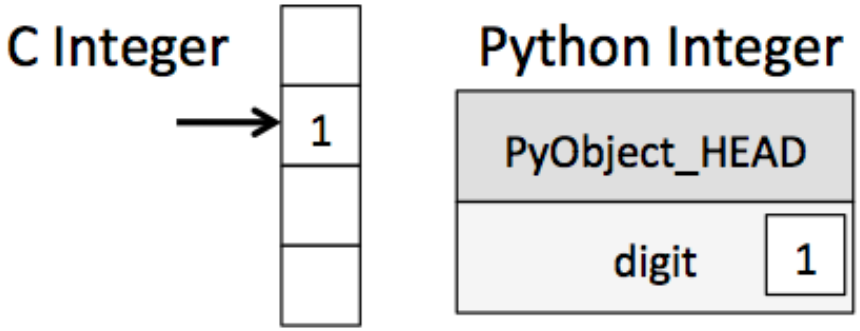
```
long x = 1;
```

- Sin embargo cuando creamos en Python un número entero grande (e.g. x = 100000) la representación de ese dato en C sería algo así:

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

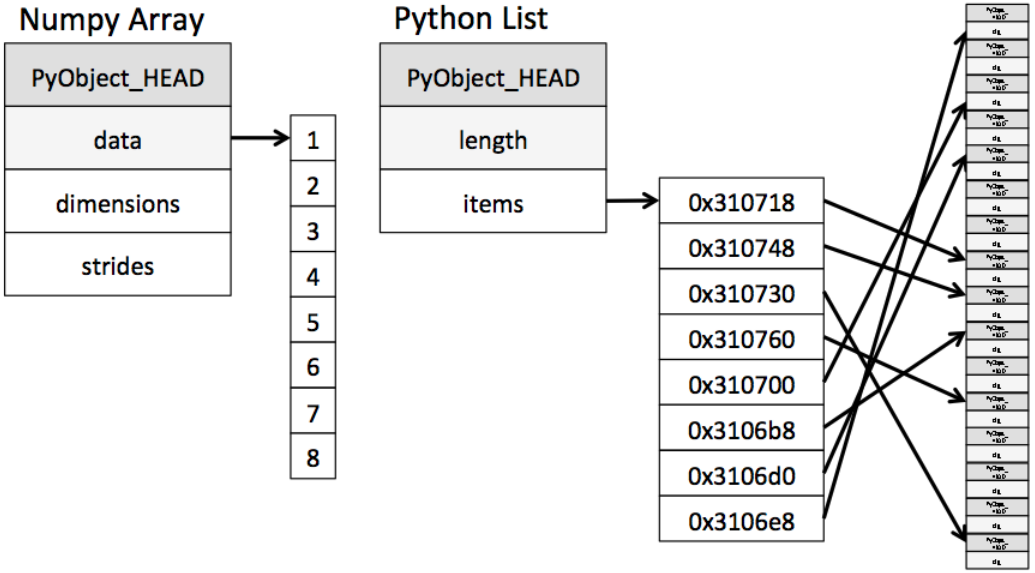
Datos en Python

- De esta forma cuando creamos un número en Python estamos creando algo más que un número.
- Esto se debe a que el tipado de los datos es dinámico. Los datos en Python son referencias a espacios de memoria con información sobre la información a la que apunta.



Datos en Python

- Al manejar más información, el trabajo del procesador es mayor. Esto se agrava con el tipo lista, pues la capacidad de generar listas de datos heterogéneos supone un coste de memoria enorme. La reimplementación de NumPy arregla este problema:



Array de Numpy

- La forma más sencilla de crear arrays en NumPy es a través de listas:

```
import numpy as np

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

- El array de NumPy no permite datos heterogéneos. Por ellos el tipo de los datos de la lista debe ser indicado o en su defecto se impondrá el más inclusivo.

```
y = np.array([1, 2, 3.4]) → array([1., 2., 3.4])

z = np.array([1, 2, 3], dtype = ‘int8’)
```

Array de Numpy

- La forma más sencilla de crear arrays en NumPy es a través de listas:

```
import numpy as np

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

- El array de NumPy no permite datos heterogéneos. Por ellos el tipo de los datos de la lista debe ser indicado o en su defecto se impondrá el más inclusivo.

```
y = np.array([1, 2, 3.4]) → array([1., 2., 3.4])

z = np.array([1, 2, 3], dtype = ‘int8’)
```

Array de Numpy

- Por supuesto también se pueden crear arrays de más de una dimensión:

```
import numpy as np

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

x[0] → array([1, 2])

x[0,0] → 1

x[:, 1] → array([2, 4, 6])
```

Arrays desde plantillas

- La forma más común de crear arrays será a través de plantillas que nos proporciona el módulo. Veremos las más importantes:
  - zeros

```
x = np.zeros((3,3)) → array([[0., 0., 0.],
                             [0., 0., 0.],
                             [0., 0., 0.]])
```

  - ones

```
x = np.ones(10000, dtype=int) → array([1, 1, 1, ..., 1, 1, 1])
```

Arrays desde plantillas

- full

```
x = np.full((2, 2), np.pi) → array([[3.14159265, 3.14159265],
                                    [3.14159265, 3.14159265]])
```

- arange

```
x = np.arange(5) → array([0, 1, 2, 3, 4])

y = np.arange(0, 10, 2) → array([0, 2, 6, 8])
```

- linspace

```
np.linspace(0, 1, 5) → array([ 0.0, 0.25, 0.5, 0.75, 1.0])
```

Arrays desde plantillas

- random

```
x = np.random.random(2,2) → array([[0.14896265, 0.84451222],
                                   [0.07854441, 0.50422254]])
```

- normal, exponential, gamma, etc.

```
x = np.random.normal(0,1,5) → array([-0.58092028,  2.40397492,
0.14483178, -0.28408127,  0.54038869]) # mean = 0, std = 1

y = np.random.exponential(0.25, 5) → array([0.33717166,
0.55792268, 0.35767695, 0.11521314, 0.12021909]) # lambda = 0.25
```

- randint

```
np.random.randint(0, 10, (3, 3)) → array([[2, 3, 4],
                                          [5, 7, 8],
                                          [0, 5, 0]])
```

Arrays desde plantillas

- eye

```
x = np.eye(3) → array([[ 1.,  0.,  0.],
                       [ 0.,  1.,  0.],
                       [ 0.,  0.,  1.]])
```

- empty

```
x = np.empty(10) → array([0.7565329 , 0.3939825 , 0.71197594,
0.93838362, 1.0232119 , 1.01689395, 0.08046008, 1.47431209,
1.3004099 , 0.30627926])
```

Esta es la forma más rápida de crear arrays pues no cambia los datos en la memoria.

Tipos en NumPy

- Como hemos visto cuando creamos los arrays podemos asociar un tipo de dato a los arrays. Podemos hacerlo a través de texto o con el objeto NumPy asociado:

```
x = np.zeros(10, dtype = 'int8')
y = np.zeros(10, dtype = np.int8)
```

Tipos 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

Atributos de los arrays

- Todos los arrays tienen 3 propiedades: dimensiones, forma y tamaño.

```
x1 = np.random.randint(10, size=6)  # Array de una dimensión.

x2 = np.random.randint(10, size=(3, 4))  # Array de dos dimensiones.

x3 = np.random.randint(10, size=(3, 4, 5))  # Array de tres dimensiones.

print(x3.ndim) → 3

print(x3.shape) → (3, 4, 5)

print(x3.size) → 60
```

Atributos de los arrays

- Todos los arrays tienen 3 propiedades: dimensiones, forma y tamaño.

```
x1 = np.random.randint(10, size=6)  # Array de una dimensión.

x2 = np.random.randint(10, size=(3, 4))  # Array de dos dimensiones.

x3 = np.random.randint(10, size=(3, 4, 5))  # Array de tres dimensiones.

print(x3.ndim) → 3

print(x3.shape) → (3, 4, 5)

print(x3.size) → 60
```

Copiar arrays

- ¡Cuidado! Si simplemente asignamos un subarray a una variable estamos haciendo una referencia al objeto original.

```
x = np.zeros(10, dtype= np.int) →
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

y = x[::2]

for i in range(len(y)):
    y[i] = 1

print(x) → [1 0 1 0 1 0 1 0 1 0]
```

Copiar arrays

- Para copiar arrays hacemos uso del método copy(). El mismo ejemplo:

```
x = np.zeros(10, dtype= np.int) →
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

y = x[::2].copy()

for i in range(len(y)):
   y[i] = 1

print(x) → [0 0 0 0 0 0 0 0 0 0]

print(y) → [1 1 1 1 1]
```

Copiar arrays

- Para copiar arrays hacemos uso del método copy(). El mismo ejemplo:

```
x = np.zeros(10, dtype= np.int) →
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

y = x[::2].copy()

for i in range(len(y)):
   y[i] = 1

print(x) → [0 0 0 0 0 0 0 0 0 0]

print(y) → [1 1 1 1 1]
```

UFuncs

- En NumPy existen una serie de funciones que nos permiten trabajar con cada elemento del array sin tener que acceder a través de un for (lo que es muy costoso). Llaman a estas funciones ‘UFuncs’.
- Por ejemplo, si quisieramos encontrar el inverso de todos los elementos de un array usaríamos una función como esta:

```
def inverso(x: np.array) -> np.array:
    out = np.empty(len(x))
    for i in range(len(x)):
        out[i] = 1 / x[i]
    return out
```

UFuncs

- Sin embargo esta función es muy ineficiente.

```
x = np.arange(1, 101, dtype=np.float)
y = inverso(x)
```

- Podemos hacer lo mismo a mayor velocidad simplemente con:

```
x = np.arange(1, 101, dtype=np.float)
y = 1 / x
```

UFuncs

- Todas las operaciones básicas están definidas como métodos mágicos de np.array así que podemos hacer cosas como estas:

```
x = np.arange(5, dtype=int) → [0 1 2 3 4]

x + 2 → [2 3 4 5 6]

x * 3 → [0 3 6 9 12]

x **2 → [0 1 4 9 16]

x / 2 → [0.0 0.5 2.0 4.5 8]

2 / x → [inf 2.0 1.0 0.66666666666 0.5]
```

UFuncs

- Tabla aritmética:

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`)

UFuncs

- Valor absoluto:

```
x = np.array([-2, -1, 0, 1, 2])
abs(x) → [2 1 0 1 2]
```

- Funciones trigonométricas.

```
theta = np.linspace(0, 2*np.pi, 100)
from matplotlib.plot import plot, clf
plot(np.sin(theta))
plot(np.cos(theta))
clf()
plot(np.tan(theta))
```

UFuncs

- Valor absoluto:

```
x = np.array([-2, -1, 0, 1, 2])
abs(x) → [2 1 0 1 2]
```

- Funciones trigonométricas.

```
theta = np.linspace(0, 2*np.pi, 100)
from matplotlib.plot import plot, clf
plot(np.sin(theta))
plot(np.cos(theta))
clf()
plot(np.tan(theta))
```

UFuncs

- Potencias y logaritmos:

```
x = np.arange(1, 11)
```

np.exp(x) → e^x

np.pow(x, 3) → x^3

np.pow(3, x) → 3^x

```
np.log2(x)
np.log10(x)
```

np.log(x) → ln(x)

UFuncs

- En todas las Ufuncs podemos especificar la salida:

```
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y) → [0. 10. 20. 30. 40.]
```

UFuncs

- Agregados:

```
x = np.arange(1, 10)
np.sum(x) → 45
np.min(x) → 1
np.max(x) → 9
m = np.arange(1,10).reshape((3,3))
np.sum(m) → 45
np.sum(m, axis=0) → [12 15 18]
np.sum(m, axis=1) → [6 15 24]
```

UFuncs

- Tabla agregados:

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 mean 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

UFuncs

- Booleanos:

```
x = np.arange(10) ** 2

x > 25 → [False, False, False, False,
False, False,  True,  True,  True, True]

np.any(x % 2 == 0) → True

np.count_nonzero(x < 6) → 3

np.sum(x > 25) → 4

np.sum((x>25) & (x<75)) → 3
```