**Clase 04. Programación con arrays: introducción a NumPy**

# Estructuras de datos. Arrays

## Estructuras de Datos

Habitualmente, cuando trabajamos en Python con datos orientados a Data Science, vemos que estos vienen en distintos formatos o tipos de datos. Estos conjuntos de datos están organizados de una determinada manera, y gracias a ello podemos acceder a estos datos, operar con ellos, mostrarlos, y en general, manipularlos. Tal como vimos en la clase pasada, hablaremos aquí de **estructuras de datos** . No pensemos aquí solamente en datos que podríamos ver, por ejemplo, en una planilla. Este análisis puede extenderse a datos de otras clases, como por ejemplo archivos de audio, o imágenes. Estos tipos diversos de datos tienen que ser transformados a números para poder ser interpretados por los algoritmos.

Por otra parte, para trabajar con estructuras de datos es necesario realizar operaciones que son habituales para este tipo de trabajo. Python ya cuenta con una serie de librerías especializadas para realizar esos trabajos. En esta clase y en la siguiente veremos los dos paquetes principales: NumPy (Numerical Python) y Pandas. Éstos paquetes proveen funciones avanzadas para trabajar con conjuntos de datos de tamaños diversos. De alguna forma, es similar a trabajar con listas, tal como vimos en la clase anterior, pero en estos casos el almacenamiento es más eficiente, y funciona muy bien con conjuntos de datos grandes.

## Arrays

Recordemos de la Clase 02 que en Python existen varios tipos de estructuras de datos, tales como list, tuple, dict y set. Cada una de estas estructuras contaba con alguna características particulares. La realidad es que habitualmente estas características no alcanzan para cubrir las necesidades del data scientist.

A continuación extenderemos la aplicación de estos tipos de estructura de datos, agregando el tipo de dato array.

Comparado con el tipo list, ambos tipos de datos sirven para guardar conjuntos de datos ordenados en memoria. No obstante, tienen algunas diferencias. Mientras que el tipo de dato list puede guardar datos de diferentes tipos, el tipo de dato array guarda datos de un único tipo. Esto le permite ser más eficiente, especialmente al trabajar con conjuntos de datos grandes. Además, para definir un dato de tipo array, debemos importar previamente el módulo array. Veamos su aplicación en código:

| L = list(range(10)) L # Muestra una lista de 10 elementos import array as arr #Necesitamos importar el módulo array para crear arrays A = arr.array("i",range(10)) A # Muestra un array de 10 elementos |
| --- |

# NumPy y ndarrays. Creación de ndarrays

¿Cómo lograr una mejora en el uso de estas estructuras? El paquete NumPy nos da la respuesta. NumPy define un nuevo tipo de dato denominado ndarray, que funciona como un array con vitaminas. Comencemos creando una estructura de este tipo:

| import numpy as np # importamos la librería NumPy Npa = np.array(range(10)) # creamos un ndarray de 10 elementos Npa |
| --- |

También tenemos funciones predeterminadas para crear ndarrays ya rellenos con datos. Mostramos algunas a continuación:

| Np\_cero = np.zeros(10) # Ndarray de ceros Np\_cero\_int = np.zeros(10, dtype=int) # Ndarray de ceros, pero especificando que sean enteros Np\_uno = np.ones(10) # Ndarray de unos print(Np\_cero, Np\_cero\_int, Np\_uno) Np\_relleno = np.full(10,256) # Rellenar un array de 10 elementos con el número 256 Np\_relleno\_dimensiones = np.full((2,3),256) # Armar una matriz de 2 filas por 3 columnas y rellenarla con el número 256 print(Np\_relleno) print(Np\_relleno\_dimensiones) Np\_rango = np.arange(10) # Armar de forma compacta un array con números ordenados Np\_random\_dimensiones = np.random.randint(10, size=(3, 4)) # Array random de 3 filas por 4 columnas print(Np\_random\_dimensiones) |
| --- |

De esta manera podemos crear arrays en NumPy

# Tipos de datos y atributos de arrays

## Tipos de datos

Los objetos de tipo ndarray pueden ser de un solo tipo. Como vimos anteriormente, podemos dejar el tipo vacío, con lo cual Python decide el tipo basándose en los elementos ingresados; o bien, podemos especificar el tipo, por ejemplo cuando sabemos a priori qué tipo de datos contendrá el ndarray, de tal forma de hacer más eficiente el uso de la memoria.

Para especificar el tipo de dato en la creación del ndarray, podemos consultar los tipos de la lista oficial publicada en este link: <https://numpy.org/devdocs/user/basics.types.html>

## Atributos

Al ser objetos de Python, los ndarrays tienen atributos. Estos atributos se ocupan de tomar registro del tamaño, forma, consumo de memoria y tipo de datos del ndarray.

| print(Np\_cero.ndim) # Vector de una dimensión print(Np\_relleno\_dimensiones.ndim) # Matriz de 2 dimensiones (filas y columnas) print(Np\_relleno\_dimensiones.shape) # 2 filas por 3 columnas print(Np\_relleno\_dimensiones.size) # 6 elementos en total print(Np\_cero.dtype) # Punto flotante print(Np\_cero\_int.dtype) # Enteros print(Np\_cero.itemsize) # Tamaño de cada elemento print(Np\_cero.nbytes) # Tamaño total del array print(Np\_cero\_int.nbytes) # Los enteros ocupan menos que los elementos de punto flotante |
| --- |

# Indexado y acceso

## Acceso a elementos

Anteriormente dijimos que los arrays y ndarrays, al igual que las estructuras de tipo list, contienen datos ordenados. Esto significa que el orden de cada elemento es siempre el mismo, con lo que podemos referirnos a un elemento (o *referenciarlo*) a partir de su índice o número de posición. Como particularidad, en Python los índices comienzan desde el número 0. Con lo cual, un conjunto de 10 elementos tendrá índices desde el 0 hasta el 9. Veamos un ejemplo:

| rango = range(1,11) # La función range incluye al primer número y excluye al último Np\_diez\_numeros = np.array(rango) # Crear un ndarray que contenga al rango generado print(Np\_diez\_numeros) print(Np\_diez\_numeros[0]) # Primer elemento print(Np\_diez\_numeros[4]) # Quinto elemento print(Np\_diez\_numeros[-1]) # Cuento en reversa, comenzando por el -1 (último elemento) print(Np\_diez\_numeros[-2]) # Penúltimo elemento print(Np\_random\_dimensiones) print(Np\_random\_dimensiones[2,1]) #Tercera fila, segunda columna |
| --- |

## Acceso a subarrays

En Python podemos “recortar” porciones en conjuntos de datos usando la operación de *slice*. Para ello usamos la siguiente notación: objeto[desde:hasta:tamaño\_de\_paso]. Por defecto los valores se recorren desde 0 hasta el tamaño del objeto, de a un paso (de a un elemento). Veamos algunos ejemplos a continuación:

| print(Np\_diez\_numeros[:4]) # Primeros cuatro elementos print(Np\_diez\_numeros[3:]) # Desde el cuarto elemento hasta el final print(Np\_diez\_numeros[4:7]) # Ojo! Desde el quinto elemento hasta el séptimo  print(Np\_diez\_numeros[::2]) # Desde el primer elemento, saltando de a dos print(Np\_diez\_numeros[1::2]) # Desde el segundo elemento, saltando de a dos print(Np\_diez\_numeros[::-2]) # Si el step es negativo, camino hacia atrás |
| --- |

En el caso de contar con más de una dimensión, accedemos en forma ordenada a cada dimensión de la forma objeto[dimensión\_1,dimensión\_2,...]. En cada dimensión podemos hacer un slice de la misma forma que lo hacíamos anteriormente.

| print(Np\_random\_dimensiones[2,]) # Tercera fila, todas las columnas print(Np\_random\_dimensiones[:2,:2]) # Subconjunto de primeras dos filas y primeras dos columnas print(Np\_random\_dimensiones[2,3]) # Tercera fila, cuarta columna |
| --- |

# Operaciones básicas: reshape, concatenación, splitting

Existen muchas operaciones para poder explotar al máximo el potencial que brinda NumPy. Podemos organizar las principales operaciones en tres grandes grupos: operaciones básicas, agregaciones y operaciones vectorizadas. Existen varios tipos más de operaciones, pero aquí veremos las básicas que permitan introducirnos en el mundo del Data Science.

## Reshape

Hasta ahora, no vimos una manera rápida de organizar elementos en un array de varias dimensiones. Supongamos por ejemplo que queremos rellenar un tablero de ajedrez (de 8 x 8 casilleros) con números, y tenemos a mano la función range() que nos permite generar los números sin necesidad de escribirlos a mano.

Si tratamos de hacer un tablero de ajedrez relleno de ceros, o de cualquier número fijo, con lo que conocemos hasta ahora es posible hacerlo fácilmente.

| Ajedrez\_ceros = np.zeros((8,8)) print(Ajedrez\_ceros) Ajedrez\_num = np.full((8,8),35) print(Ajedrez\_num) |
| --- |

En cambio, si queremos rellenarlo con números consecutivos del 1 al 64, nos dará un error.

| rango\_ajedrez = range(1,65) Ajedrez = np.array((8,8),rango\_ajedrez) # Esto no funciona |
| --- |

Para solucionar esta situación y similares, NumPy define una operación denominada *reshaping*. Básicamente, tomamos un ndarray como entrada, y le “cambiamos la forma”, en términos de dimensiones. Veamos algunos ejemplos:

| Ajedrez\_64 = np.arange(1,65) # Una array de números de 1 a 64  print(Ajedrez\_64) Ajedrez\_64 = np.arange(1,65).reshape(8,8) # Organizarlo en 8 x 8 print(Ajedrez\_64) Ajedrez\_64 = Ajedrez\_64.reshape(8,8) # Equivalente al comando anterior print(Ajedrez\_64) |
| --- |

## Concatenación

Para “enganchar” o concatenar dos o más arrays, utilizamos las funciones concatenate y vstack / hstack. Veamos algunos ejemplos:

| Array\_1 = np.random.randint(10,size=5) Array\_2 = np.random.randint(10,size=5) print(Array\_1, Array\_2) # Dos arrays de 5 elementos, con números aleatorios del 0 al 9 Arrays\_concatenados = np.concatenate([Array\_1, Array\_2]) # Un array que contiene los 10 números concatenados print(Arrays\_concatenados) |
| --- |

Si queremos combinar los arrays en forma explícita de manera horizontal o vertical, usaremos las funciones vstack y hstack

| Array\_1 = np.random.randint(10,size=5) Array\_2 = np.random.randint(10,size=5) print(Array\_1, Array\_2) # Dos arrays de 5 elementos, con números aleatorios del 0 al 9 Arrays\_concatenados = np.concatenate([Array\_1, Array\_2]) # Un array que contiene los 10 números concatenados print(Arrays\_concatenados) Array\_extra = np.array([[10],[20]]) # Un array de dos dimensiones (vertical) con los números 10 y 20 print(Array\_extra) Array\_apilados\_v\_h = np.hstack([Array\_extra,Array\_apilados\_v]) # Concatenar el array vertical con el array apilado print(Array\_apilados\_v\_h) print(np.shape(Array\_apilados\_v\_h)) # ¿Cuántas filas y columnas tiene? |
| --- |

De esta forma podemos combinar arrays.

## Splitting

Por otra parte, podemos “partir” los arrays utilizando las funciones split, hsplit y vsplit. Estas funciones reciben como argumento el array a “partir” y una lista con los puntos de corte. Podemos pensar los puntos de corte como los “huecos” existentes entre los elementos del array. Veamos los siguientes ejemplos:

| Array\_partido = np.split(Arrays\_concatenados,[2]) # Parte el array entre el segundo y tercer elemento print(Array\_partido) Array\_partido\_2 = np.split(Arrays\_concatenados,[2,8]) # Parte el array entre el segundo y tercero, y entre el octavo y noveno elemento print(Array\_partido\_2) Parte\_1, Parte\_2, Parte\_3 = Array\_partido\_2 # Guarda cada elemento en una variable. Muy útil print(Parte\_1, Parte\_2, Parte\_3) Ajedrez\_partido\_1 = np.hsplit(Ajedrez\_64,[4]) # Parte el tablero de ajedrez por la mitad, de forma vertical print(Ajedrez\_partido\_1) Ajedrez\_partido\_2 = np.vsplit(Ajedrez\_64,[4]) # Parte el tablero de ajedrez por la mitad, de forma horizontal print(Ajedrez\_partido\_2) |
| --- |

De esta manera podemos trabajar con operaciones básicas sobre arrays.

# Agregaciones

¿Cómo haríamos para calcular un promedio de diez números dentro de un array? Simple, sumamos los números y dividimos el resultado por 10. Hagámoslo en Python:

| Array\_aleatorio = np.random.randint(10,size=10) print(Array\_aleatorio) suma = 0 for i in Array\_aleatorio:  suma += i promedio = suma / np.size(Array\_aleatorio) print(promedio) |
| --- |

Esta resolución es bonita y elegante. No obstante, es posible que debamos calcular frecuentemente este tipo de medidas resumen, como el promedio, la varianza, el desvío, la mediana, y similares. Para estos casos frecuentes, NumPy tiene atajos que son muy útiles a la hora de realizar los cálculos.

| print(Array\_aleatorio.sum()) # Suma de elementos print(Array\_aleatorio.mean()) # Promedio de elementos print(Array\_aleatorio.max()) # Elemento más grande. print(np.median(Array\_aleatorio)) # Mediana print(np.std(Array\_aleatorio)) # Desvío estándar print(np.var(Array\_aleatorio)) # Varianza |
| --- |

Todas las funciones pueden escribirse como np.nombre\_de\_función(objeto). Las funciones más comunes (sum, mean, max), pueden escribirse equivalentemente como métodos aplicados sobre el objeto (por ejemplo objeto.mean().

De esta manera NumPy organiza las funciones de agregación, o resumen, de forma compacta y en una sola línea de código.

Es necesario aclarar que Python ofrece estas funciones por defecto, pero en NumPy están optimizadas para trabajar con grandes volúmenes de datos.

# Operaciones vectorizadas

Finalmente, NumPy trae un conjunto de funciones para operaciones optimizadas para arrays, de tal forma de trabajar estas estructuras a la manera de vectores o matrices matemáticas. A estas funciones se las llama operaciones vectorizadas, o funciones universales (ufuncs). Muchas veces es necesario aplicar operaciones sobre cada elemento, como por ejemplo sumar una constante, o bien sumar dos vectores elemento a elemento, o realizar un producto matricial. Para ello podemos operar utilizando estas funciones con los ndarrays de NumPy. Veamos algunos ejemplos:

| print(Array\_1, Array\_2)  print(Array\_1 + 5) # Suma 5 a cada elemento de Array\_1 print(Array\_1 + Array\_2) # Suma Array\_1 y Array\_2, elemento a elemento print(np.add(Array\_1,Array\_2)) # Igual al anterior print(np.matmul(Array\_1, Array\_2)) # Producto vectorial: Se multiplican los elementos en la misma posición de Array\_1 y Array\_2 (se multiplican los dos primeros, luego los dos segundos, etc.), y luego se suman todos los resultados. |
| --- |

La guía completa de referencia de las ufuncs está disponible en https://numpy.org/doc/stable/reference/ufuncs.html#available-ufuncs

Siempre que se trabaje con ndarrays de NumPy se recomienda utilizar estas funciones, porque además de ser más compactas que realizar la programación a mano, están construidas de tal forma que funcionan más rápidamente. Esto es especialmente importante cuando utilizamos conjuntos grandes de datos.

# Ejercicios

1. Generar un array aleatorio de 100 elementos. Calcular la mediana correspondiente.
2. Recordar los ejercicios con funciones para cálculo de factorial y suma de serie. Repetir ambos ejercicios (puede ser sobre el mismo código), pero ahora utilizar las nuevas operaciones aprendidas con los ndarrays.
3. En [este link](https://docs.google.com/spreadsheets/u/0/d/15Stnum8mI9QdwF9ZiPFCe3_qG9XyZyNzmOSIDunzPjA/edit) se provee un archivo con los resultados de la Encuesta de Sueldos de Openqube (<https://sueldos.openqube.io/encuesta-sueldos-2020.02/>), en este caso de Febrero 2020. Calcular y comparar media y mediana de los sueldos netos.