MÓDULO 3: HERRAMIENTAS FUNDAMENTALES PARA LA PREPARACIÓN Y ANÁLISIS DE DATOS

Unidad 1: Conceptos básicos de NumPy: matrices y operaciones con vectores

En este módulo, sentaremos las bases más elementales y necesarias que un *data scientist* o *data analyst* necesitan para trabajar en el análisis y la explotación de los datos. Desde un análisis descriptivo hasta un complejo modelo de *machine learning*, requieren un importante trabajo de análisis de las fuentes de datos, limpieza y depuración de variables basura, imputación de valores faltantes, comprensión estadística de las variables, distribución, creación de nuevas variables, etcétera. Se estima que estas tareas llevan aproximadamente el 80 % del tiempo y que solo en el 20 % restante (que abordaremos en el próximo módulo) sucede el proceso de entrenamiento de un algoritmo para la creación de un modelo predictivo o la elaboración de los gráficos que permitirán analizar los datos y llevarlos en un reporte para que el *management* tome las decisiones.

3.1.1 Introducción a NumPy

NumPy es la versión abreviada de **Num**erical **Py**thon, una de las bibliotecas más utilizadas por la analítica de datos. Provee estructuras de datos y diferentes tipos de operaciones para realizar procesamientos en paralelo. En *data science*, en general, se trabaja con grandes cantidades de datos que están ordenados por tablas, por filas o columnas. Entonces, muchas veces es conveniente realizar operaciones sobre toda una fila al mismo tiempo y no elemento por elemento. NumPy provee al científico de datos de herramientas para hacer más eficientes estas ejecuciones.

Estas son algunos de los objetos y herramientas que se pueden encontrar en la librería NumPy:

- **Ndarray (arreglos de *n* dimensiones)**: es una matriz multidimensional (generalmente, de tamaño fijo) de elementos del mismo tipo y tamaño. El número de dimensiones y elementos en una matriz se define por su forma, que es una tupla de *n* enteros positivos que

especifican los tamaños de cada dimensión. El tipo de elementos en la matriz se especifica mediante un objeto de tipo de datos (*dtype*) separado, uno de los cuales está asociado con cada *ndarray*.
- Funciones matemáticas para operaciones rápidas en matrices enteras de datos sin tener que escribir bucles.
- Herramientas para leer y escribir datos de matriz en el disco y trabajar con archivos mapeados en memoria.
- Álgebra lineal, generación de números aleatorios y transformación de Fourier.

Si bien NumPy por sí solo no proporciona la posibilidad de hacer modelos o ninguna funcionalidad de cálculo científica, comprender los arreglos NumPy y los cálculos orientados a estos facilita el entendimiento de otras librerías con semántica similar —como por ejemplo, Pandas— de manera mucho más efectiva.

Para la mayoría de las aplicaciones de análisis de datos, las principales funcionalidades en las que nos centraremos son:

- operaciones de matriz vectorizadas rápidas para limpieza de datos, subconjunto y filtrado, transformación, *munging* y cualquier otro tipo de cálculo;
- algoritmos de matriz comunes, como operaciones de clasificación, únicas y de conjuntos;
- estadísticas descriptivas eficientes y datos de agregación y resumen;
- alineación de datos y manipulaciones de datos relacionales para fusionar y unir conjuntos de datos heterogéneos;
- expresiones de lógica condicional con matrices;
- manipulaciones de datos grupales (agregación, transformación, aplicación de funciones).

Data munging

Hace referencia a la limpieza y depuración de los datos que permite que sean más aptos para el análisis e involucra una serie de actividades.

- enriquecer los datos;
- estandarizar los datos:
- normalización;
- aplicar una macro;
- Buscar patrones
- ordenación;
- filtrado;
- unión;
- transposición;
- analizar gramaticalmente;
- transformación de tipos de datos;
- imputación de datos faltantes.

3.1.2 NumPy ndarray: el arreglo multidimensional de NumPy

Características generales de los arreglos multidimensionales

Una de las características clave de NumPy es su objeto de arreglo *n*-dimensional o *ndarray*, que es un contenedor rápido y flexible para grandes conjuntos de datos en Python. Los arreglos o matrices le permiten realizar operaciones matemáticas en bloques completos de datos utilizando una sintaxis similar a las operaciones equivalentes entre elementos escalares.

Para dar una idea de cómo NumPy habilita los cálculos por lotes, primero importemos NumPy y luego generemos una pequeña matriz de datos aleatorios llamada *data*, y probemos ejecutar operaciones matemáticas con la matriz resultante.

Figura 1: Cómo NumPy habilita cálculos por lotes

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

In [2]: data = np.random.randn(2, 3)
        data

Out[2]: array([[-0.18280995, -0.55867547, -1.52492821],
               [-1.39116654, -0.52552515,  0.86453007]])

In [3]: data * 2

Out[3]: array([[-0.3656199 , -1.11735094, -3.04985642],
               [-2.78233307, -1.05105029,  1.72906014]])

In [4]: data + 5

Out[4]: array([[4.81719005, 4.44132453, 3.47507179],
               [3.60883346, 4.47447485, 5.86453007]])

In [5]: data + data

Out[5]: array([[-0.3656199 , -1.11735094, -3.04985642],
               [-2.78233307, -1.05105029,  1.72906014]])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Vea cómo el valor escalar se multiplica por cada uno de los elementos de la matriz. En el caso de la suma de matrices, se adicionan los elementos posicionados en la misma ubicación de la matriz.

Un **ndarray** es un contenedor multidimensional genérico para datos homogéneos; es decir, todos los elementos deben ser del mismo tipo. Cada matriz tiene dos características intrínsecas: una forma (***shape***), que indica el tamaño de cada dimensión, y un **dtype**, es decir, un objeto que describe el tipo de datos de la matriz:

```
In [6]: data.shape

Out[6]: (2, 3)

In [7]: data.dtype

Out[7]: dtype('float64')
```

Creación de arreglos

La forma más fácil de crear un arreglo es usar la función *array*. Esto acepta cualquier secuencia de objetos similares (incluidas otras matrices) y produce una nueva matriz NumPy que contiene los datos provistos. Por ejemplo, una lista es un buen candidato para la conversión a arreglo:

Figura 2: Creación de arreglos

```
In [18]: lista = [0.5, 1.5, 2, 3.5, 5.5, 9, 14.5]

In [19]: arreglo = np.array(lista)

In [20]: arreglo

Out[20]: array([ 0.5,  1.5,  2. ,  3.5,  5.5,  9. , 14.5])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Las secuencias anidadas, como una lista de listas de igual longitud, se convertirán en una matriz multidimensional:

Figura 3: Matriz multidimensional

```
In [33]: lista1= [[0,1,2],[0.5,4,32]]

In [34]: Matriz= np.array(lista1)
         Matriz

Out[34]: array([[ 0. ,  1. ,  2. ],
                [ 0.5,  4. , 32. ]])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

También tenemos otras formas de crear arreglos multidimensionales. La función **zeros** crea arreglos de ceros, en donde hay que insertar la forma de la matriz deseada.

Figura 4: Función zeros

```
In [40]: np.zeros((2,3))

Out[40]: array([[0., 0., 0.],
                [0., 0., 0.]])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Además, tenemos la función **empty**, a la que se le asigna la forma deseada sin inicializar las entradas, por lo que requiere que el usuario establezca manualmente todos los valores en la matriz.

Figura 5: Función empty

```
In [2]: vacia=np.empty((2,3,2))
        vacia

Out[2]: array([[[0., 0.],
                [0., 0.],
                [0., 0.]],

               [[0., 0.],
                [0., 0.],
                [0., 0.]]])

In [3]: vacia[[0],0,1]=3
        vacia

Out[3]: array([[[0., 3.],
                [0., 0.],
                [0., 0.]],

               [[0., 0.],
                [0., 0.],
                [0., 0.]]])
```

```
In [4]: vacia[[1],2,0]=5
        vacia

Out[4]: array([[[0., 3.],
                [0., 0.],
                [0., 0.]],

               [[0., 0.],
                [0., 0.],
                [5., 0.]]])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Puede suceder también que dicha función traiga *valores basura*. Esto sucede porque el valor de sus elementos es el resultado de lo que ha habido antes almacenado en esa parte de la RAM (*random access memory*).

Figura 6: Función con valores basura

```
In [34]: data=np.random.randn(2,3)
         data

Out[34]: array([[ 1.3187989 , -0.37692257,  1.23064367],
                [ 0.45373581, -1.03157843, -0.00788164]])

In [39]: np.empty((2,3))

Out[39]: array([[1.3187989 , 0.37692257, 1.23064367],
                [0.45373581, 1.03157843, 0.00788164]])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

En el caso anterior, la función **empty** asignó los valores de una matriz del mismo modo que antes. No se recomienda utilizarla porque, si no se inicializan todos los valores después, puede generar que las operaciones posteriores den resultados no deseados a causa de los valores basura que a dicha matriz le puedan quedar sin inicializar.

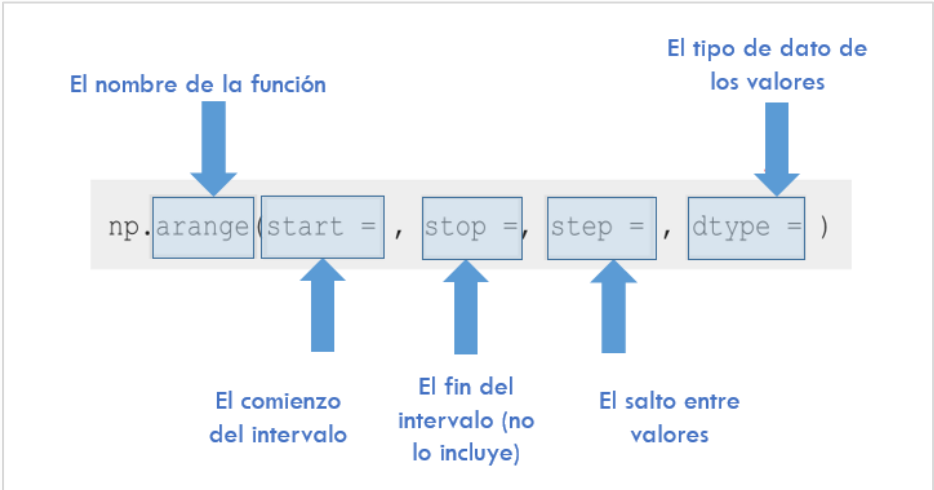
Por otro lado, el equivalente de la función de Python **range** se encuentra dentro de la biblioteca de NumPy. Se expresa ***arange*** y cumple la misma función: arroja un arreglo unidimensional al que puede cambiársele la dimensión con la función *shape*.

Figura 7: Función arange

```
In [52]: rango = np.arange(9)
         rango

Out[52]: array([0, 1, 2, 3, 4, 5, 6, 7, 8])

In [54]: rango.shape=(3,3)
         rango

Out[54]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Figura 8: Parámetros de la función arange de NumPy



Fuente: elaboración propia.

Otra función muy utilizada para crear arreglos es la función **random**. Como su nombre lo indica, devuelve una matriz con números aleatorios con la forma solicitada.

Figura 9: Función random

```
In [34]: data=np.random.randn(2,3)
         data
Out[34]: array([[ 1.3187989 , -0.37692257,  1.23064367],
                [ 0.45373581, -1.03157843, -0.00788164]])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Otra función para la creación de un arreglo de *n* dimensiones es **identity**, la cual crea una matriz de identidad de dimensión $n \times n$. En álgebra, las *matrices de identidad* son aquellas en las cuales cualquier matriz multiplicada por ellas no produce ningún efecto, es decir, son neutras.

Figura 10: Función identity

```
In [65]: identidad=np.identity(4)
         identidad
Out[65]: array([[1., 0., 0., 0.],
                [0., 1., 0., 0.],
                [0., 0., 1., 0.],
                [0., 0., 0., 1.]])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Todos los ndarray nacen con un dtype (o tipo de datos) que se asigna automáticamente, dependiendo del tipo de datos que se le asigna al arreglo, a no ser que específicamente se fuerce un formato al conjunto de datos dado. Por ejemplo, en la función arange existe el parámetro dtype, a partir del cual se puede asignar un tipo de dato específico al tipo de datos que se está introduciendo en la información de intervalos.

Figura 11: Función arange con dato específico

```
In [86]: np.arange(2, 6, dtype='float64')
Out[86]: array([2., 3., 4., 5.])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

El tipo de datos o dtype es un objeto especial que contiene la información (o metadatos, datos sobre datos) que el ndarray necesita para interpretar una porción de memoria como un tipo particular de datos.

Se puede convertir o truncar valores de una matriz de un dtype a otro utilizando el método **astype** de ndarray. En las siguientes instrucciones, tenemos unos ejemplos.

Figura 12: Método astype

```
In [89]: np.array([7.8, 9, 15.25, 11], dtype='int32')
Out[89]: array([ 7,  9, 15, 11])

In [92]: arr=np.array([1.1, 2.2, 3.1, 4.2])
         arr.dtype
Out[92]: dtype('float64')

In [97]: arr= arr.astype('int32')
         arr
Out[97]: array([1, 2, 3, 4])

In [98]: arr.dtype
Out[98]: dtype('int32')
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Muchas veces sucede que importamos *datasets* donde los formatos de algunos campos numéricos vienen como texto. En estos casos, con NumPy podemos convertir el tipo de dato en numérico con la misma función **astype**.

Figura 13: Función astype: conversión de datos numéricos que vienen como texto

```
In [115]: celulares=np.array(['156115544','155444445','153112211'], dtype=np.string_)
          celulares.dtype
Out[115]: dtype('S9')

In [116]: celulares.astype('int')
Out[116]: array([156115544, 155444445, 153112211])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Operaciones básicas con arreglos

Los arreglos "permiten expresar operaciones por lotes en datos sin escribir ninguna para los bucles. Esto generalmente se llama ***vectorización***" (McKinney, en Solomon, 2017, https://www.it-swarm.dev/es/python/que-es-la-vectorizacion/834957754/). Es decir que NumPy implementa funciones matemáticas, de modo que, cuando una función actúa en una matriz, la operación matemática se aplica a cada entrada de esa matriz.

Figura 14: Arreglos

```
In [6]: arreglo=np.array([[2,2,2],[5,6,7]])
        arreglo
Out[6]: array([[2, 2, 2],
               [5, 6, 7]])

In [7]: arreglo * arreglo
Out[7]: array([[ 4,  4,  4],
               [25, 36, 49]])

In [21]: comp1=np.sqrt(arreglo)
         comp1
Out[21]: array([[1.41421356, 1.41421356, 1.41421356],
                [2.23606798, 2.44948974, 2.64575131]])

In [25]: comp2=10/arreglo
         comp2
Out[25]: array([[5.        , 5.        , 5.        ],
                [2.        , 1.66666667, 1.42857143]])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Las comparaciones entre matrices del mismo tamaño producen matrices booleanas:

Figura 15: Matrices booleanas

```
In [26]: comp1>comp2
Out[26]: array([[False, False, False],
                [ True,  True,  True]])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Indexación y slicing de arreglos

La indexación de un arreglo multidimensional de NumPy es un tema amplio, ya que hay muchas formas en que se puede querer seleccionar un subconjunto de sus datos o elementos individuales. Las matrices unidimensionales son simples, actúan de manera similar a las listas de Python. Pero, cuando tenemos *n* dimensiones, se vuelve algo más complejo. Repasemos primero con unos ejemplos simples de matriz unidimensional.

Figura 16: Matriz unidimensional

```
In [30]: arr = np.arange(11,20)
         arr
Out[30]: array([11, 12, 13, 14, 15, 16, 17, 18, 19])

In [31]: arr[5]
Out[31]: 16

In [32]: arr[6:]
Out[32]: array([17, 18, 19])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Como se puede ver, creamos un arreglo de 9 elementos y, para acceder a un elemento dentro de dicho arreglo, solo basta con identificar su posición (se debe recordar que el primer elemento se cuenta como posición 0).

Siguiendo con este ejemplo, veamos cómo asignamos un valor a una porción del arreglo.

Figura 17: Asignación de un valor a una porción del arreglo

```
In [55]: arr[3:6]=99
         arr
Out[55]: array([11, 12, 13, 99, 99, 99, 17, 18, 19])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Como se puede ver, si asigna un valor a un segmento, como en arr[3:6] = 99, el valor se propaga a toda la selección. Una primera distinción importante con respecto a las listas de Python es que los segmentos modificados del arreglo impactan en el arreglo original. Esto significa que los datos no se copian y cualquier modificación de la vista se reflejará en el arreglo original.

Siguiendo con el ejemplo anterior, creamos un arreglo llamado **arr_slice**, que es una porción de arreglo original arr (la misma que modificamos con el 99), y le asignamos al valor en posición 1 el número 22. Esta modificación impacta en el nuevo arreglo y, por consiguiente, en el original, como se puede ver a continuación.

Figura 18: *Slicing*

```
In [56]: arr_slice=arr[3:6]
         arr_slice
Out[56]: array([99, 99, 99])

In [57]: arr_slice[1]=22
         arr_slice
Out[57]: array([99, 22, 99])

In [58]: arr
Out[58]: array([11, 12, 13, 99, 22, 99, 17, 18, 19])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Esto se denomina comúnmente como ***slicing*** y se refiere a las operaciones en las que se trabaja con cortes o porciones de un arreglo.

Con arreglos matriciales, es decir, de más dimensiones, hay muchas más opciones. En una matriz bidimensional, los elementos en cada índice ya no son escalares, sino más bien arreglos unidimensionales:

Figura 19: Arreglos unidimensionales en una matriz bidimensional

```
In [66]: arr_2 = np.array([['a', 'b', 'c'], [4, 5, 6], [7, 'x', 9]])
         arr_2

Out[66]: array([['a', 'b', 'c'],
                ['4', '5', '6'],
                ['7', 'x', '9']], dtype='<U1')

In [67]: arr_2[2]

Out[67]: array(['7', 'x', '9'], dtype='<U1')
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Por lo tanto, a los elementos individuales se puede acceder de forma recursiva. Pero eso es demasiado trabajo, por lo que se puede pasar una lista de índices separados por comas para seleccionar varios elementos individuales.

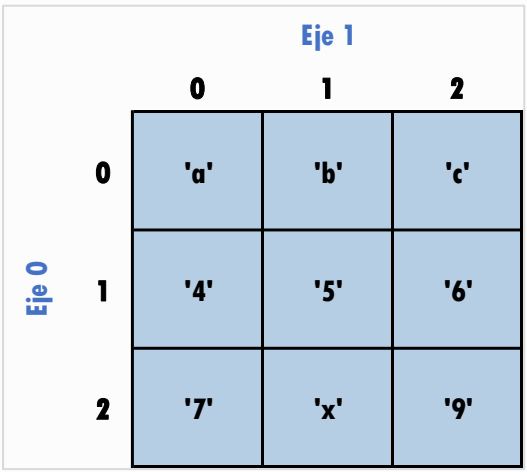
Figura 20: Lista de índices separados por coma

```
In [68]: arr_2[0][2]

Out[68]: 'c'

In [69]: arr_2[0,2]

Out[69]: 'c'

In [71]: arr_2[0,[0,1,2]]

Out[71]: array(['a', 'b', 'c'], dtype='<U1')
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

La siguiente figura puede resultar útil para pensar la forma de indexar matrices. Las filas se pueden interpretar como el eje 0 y las columnas como el eje 1.

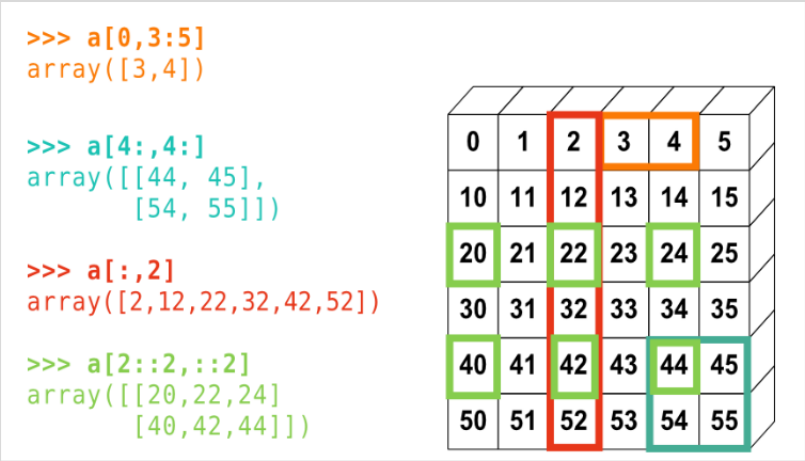
Figura 21: Una matriz de dos dimensiones



Fuente: elaboración propia.

Podemos ver otros métodos para indexar matrices de dos dimensiones en la siguiente figura, un poco más ilustrativa, en donde se parte de una matriz llamada "a" de 6 × 6.

Figura 22: Matriz a de 6 × 6



Fuente: Ávila, s. f., https://damianavila.github.io/Python-Cientifico-HCC/3_NumPy.html

La instrucción **"::2"** en NumPy saltea, en este caso, dos posiciones, por lo que en el último ejemplo (el verde) la instrucción sería: "En el eje 0, a partir de la segunda posición, seleccionamos las filas, y en el eje 1, a partir de la posición cero, seleccionamos cada 2 elementos".

Ahora veamos cómo es el proceso de indexación en matrices de más de dos dimensiones. Comencemos creando un arreglo de tres dimensiones.

Figura 23: Arreglo de tres dimensiones

```
In [3]: arr_3dim = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9]], [['a', 'b', 'c'],['d', 'e', 'f'],['g', 'h', 'i']]])
        arr_3dim

Out[3]: array([[['1', '2', '3'],
                ['4', '5', '6'],
                ['7', '8', '9']],

               [['a', 'b', 'c'],
                ['d', 'e', 'f'],
                ['g', 'h', 'i']]], dtype='<U11')
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Para ser más gráficos y comprender mejor el resultado de la matriz de tres dimensiones, esta se vería así:

Figura 24: Ilustración de una matriz de tres dimensiones

Fuente: elaboración propia.

A continuación, veremos cómo se indexa cada cara del cubo, cada una de las cuales corresponde a una matriz simple de dos dimensiones. Así, tenemos que 0 es la matriz de dos dimensiones que corresponde a los elementos numéricos (aunque tienen un dtype = '<U11', es decir, *unicode string*), y que 1 corresponde a la posición de la matriz de dos dimensiones con las letras.

Figura 25: Indexación de cada cara del cubo

```
In [4]: arr_3dim[0]

Out[4]: array([['1', '2', '3'],
               ['4', '5', '6'],
               ['7', '8', '9']], dtype='<U11')

In [5]: arr_3dim[1]

Out[5]: array([['a', 'b', 'c'],
               ['d', 'e', 'f'],
               ['g', 'h', 'i']], dtype='<U11')
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Ya determinando esto, será más fácil identificar de qué manera llevar un elemento dentro de cada matriz dimensional. Con la misma lógica que vimos antes, quedaría así:

Figura 26: Llevar un elemento a cada matriz dimensional

```
In [25]: arr_3dim[1,1,2]

Out[25]: 'f'

In [22]: arr_3dim[1,:2,::2]

Out[22]: array([['a', 'c'],
                ['d', 'f']], dtype='<U11')
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Hasta aquí hemos conocido lo básico de NumPy. A partir de ahora, podremos empezar a conocer con más profundidad las funcionalidades de NumPy, que son las que más adelante nos permitirán hacer el análisis de datos de nuestro interés.

3.1.3 Operaciones condicionales con arrays

En NumPy tenemos una función análoga a la cláusula if/elif/else que vimos en el módulo anterior dentro del lenguaje de Python, que es la función **numpy.where**. Veamos cómo funciona con un ejemplo. Supongamos que tenemos 3 arreglos, uno booleano y dos con valores de nombres, todos de la misma forma (*shape*).

Figura 27: Tres arreglos

```
In [33]: nombres1 = np.array(['Ana','Juan','Pedro','Boris','Sofia'])
         nombres2 = np.array(['Ringo','Nina','Javier','Paula','Pilar'])
         bool = np.array([True, False, False, True, True])

In [34]: nombres1
Out[34]: array(['Ana', 'Juan', 'Pedro', 'Boris', 'Sofia'], dtype='<U5')

In [35]: nombres2
Out[35]: array(['Ringo', 'Nina', 'Javier', 'Paula', 'Pilar'], dtype='<U6')

In [36]: bool
Out[36]: array([ True, False, False,  True,  True])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Supongamos que deseamos tomar un valor de *nombres1*, siempre que el valor correspondiente en *bool* sea verdadero (*true*), y, de lo contrario, tomar el valor de *nombres2*. Con la función **where**, lo podemos resolver de la siguiente forma:

Figura 28: Función where

```
In [38]: resultado = np.where(bool, nombres1, nombres2)
         resultado
Out[38]: array(['Ana', 'Nina', 'Javier', 'Boris', 'Sofia'], dtype='<U6')
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Pero las matrices que utilizamos con numpy.where pueden ser más que simples matrices o escalares de igual tamaño. Veamos algunos ejemplos:

Figura 29: Matrices con numpy.where

```
In [3]: arreglo=np.random.randn(3,3)
        arreglo

Out[3]: array([[-1.6530774 , -0.15195163,  3.14688415],
               [ 0.63950685, -0.32327973, -1.23129283],
               [-0.56436707, -0.16712834,  0.0401732 ]])

In [4]: arreglo>0

Out[4]: array([[False, False,  True],
               [ True, False, False],
               [False, False,  True]])

In [7]: np.where(arreglo>0,10,-2)

Out[7]: array([[-2, -2, 10],
               [10, -2, -2],
               [-2, -2, 10]])

In [9]: np.where(arreglo>0,99,arreglo)

Out[9]: array([[-1.6530774 , -0.15195163, 99.        ],
               [99.        , -0.32327973, -1.23129283],
               [-0.56436707, -0.16712834, 99.        ]])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

3.1.4 Métodos matemáticos y estadísticos

Se puede acceder a un conjunto de funciones estadísticas sobre una matriz completa o sobre los datos a lo largo de un eje de la matriz con distintos métodos (remitirse al punto 2.4 del Módulo 2 del curso: "Atributos y métodos"). Se pueden hacer agregaciones (a menudo llamadas *reducciones*) como sum, mean y std (desviación estándar), llamando al método del arreglo, pero también a menudo se usan las correspondientes funciones de la librería NumPy. Veamos los ejemplos en el código a partir de un arreglo de dos dimensiones llamado **cálculos**.

Figura 30: Arreglo de dos dimensiones: cálculos

```
In [31]: calculos = np.array([[2,4,6],[1,3,5],[10,20,30],[7,7,7]])
         calculos

Out[31]: array([[ 2,  4,  6],
                [ 1,  3,  5],
                [10, 20, 30],
                [ 7,  7,  7]])

In [32]: calculos.mean() # que es lo mismo que hacer>>> np.mean(calculos)

Out[32]: 8.5

In [33]: calculos.sum() # que es lo mismo que hacer>>> np.sum(calculos)

Out[33]: 102

In [34]: calculos.std() # que es lo mismo que hacer>>> np.std(calculos)

Out[34]: 8.0156097709407
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Las operaciones también pueden hacerse sobre los ejes y determinar los cálculos por cada fila o columna de la matriz, lo que dará como resultado un arreglo nuevo, con una dimensión menos, con los resultados.

Figura 31: Operaciones sobre los ejes

```
In [35]: calculos.mean(axis=0)

Out[35]: array([ 5. ,  8.5, 12. ])

In [36]: calculos.mean(axis=1)

Out[36]: array([ 4.,  3., 20.,  7.])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Otros métodos, como el **np.cumsum()** y el **np.cumprod()**, también generan como resultado un arreglo nuevo de una dimensión inferior a la del original con los resultados, y también pueden realizarse por fila o columna (con el parámetro *axis*=), o a nivel agregado total, de la misma manera que en los ejemplos anteriores.

Tarea: Se le deja al alumno la comprobación práctica de estos dos métodos en un Jupyter Notebook y la verificación del tipo de operación al que corresponden.

3.1.5 Operaciones con matrices de álgebra lineal

Hasta ahora hemos visto operaciones con matrices, pero las operaciones no siguen las reglas de cálculo de matrices del álgebra. Por ejemplo, la multiplicación de matrices a través del operador * da como resultado una multiplicación lineal de filas de la matriz 1 con las mismas filas de la matriz 2.

Figura 32: Multiplicación de matrices a través de *

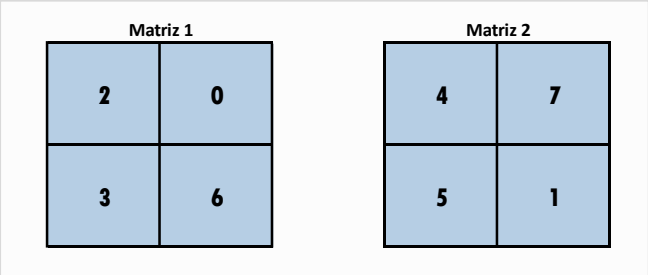
```
In [45]: array1=np.array([[2,0],[3,6]])
         array2=np.array([[4,7],[5,1]])

In [46]: array1

Out[46]: array([[2, 0],
                [3, 6]])

In [47]: array2

Out[47]: array([[4, 7],
                [5, 1]])

In [48]: array1*array2

Out[48]: array([[ 8,  0],
                [15,  6]])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Sin embargo, en álgebra, la multiplicación de matrices cumple una lógica diferente, según la cual la cantidad de filas de la matriz 1 y la cantidad de columnas de la matriz 2 deben coincidir. Vamos a comentar esto de manera muy global. Quien lo desee podrá ampliar el tema por su cuenta, ya que no es el principal de este curso.

Cada elemento de las filas de la matriz 1 se multiplica por todos los elementos de las columnas de la matriz 2 y se suman los resultados. Volvamos al ejemplo. Para que sea más fácil comprender la lógica, tenemos dos matrices de 2 × 2, por lo cual cumplen la condición de que la cantidad de filas coincide con la cantidad de columnas de la segunda matriz.

Figura 33: Dos matrices de 2 × 2



Fuente: elaboración propia.

La multiplicación de matrices comienza con el primer elemento de la primera fila de la matriz 1 y lo multiplica por el primer elemento de la primera columna de la matriz 2, y luego sigue

el segundo elemento de la primera fila de la matriz 1 con el segundo elemento de la columna de la matriz 2. Y luego se suman los resultados.

En el ejemplo sería: **(2*4) + (0*5) = 8.**

Este resultado es el primer elemento de la matriz de resultado. Luego se continúa manteniendo la primera fila de la matriz 1 como *pivot*, siguiendo con la segunda columna de la matriz 2 y haciendo lo mismo.

Quedaría la operación: **(2*7) + (0*1) = 14**

Así, tenemos el segundo elemento de la primera fila de la matriz de resultado. En el paso siguiente, se realiza lo mismo con la segunda fila de la matriz 1:

(3*4) + (6*5) = 42

(3*7) + (6*1) = 27

Esto dará como resultado el primer y segundo elemento de la segunda fila de la matriz de resultado.

Figura 34: Matriz de resultado

Matriz R

8	14
42	27

Fuente: elaboración propia.

En NumPy, la multiplicación algebraica de matrices no se realiza utilizando el operador *, sino a través de la función numpy.dot(x,y). Veamos cómo quedaría el ejemplo en Python.

Figura 35: Función numpy.dot(x,y) para la multiplicación algebraica de matrices

```
In [49]: array1.dot(array2)

Out[49]: array([[ 8, 14],
                [42, 27]])
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Quien desee ampliar más sobre operaciones de álgebra lineal en la librería de NumPy puede consultar la documentación oficial en https://numpy.org/doc/stable/reference/routines.linalg.html?highlight=linalg

Unidad 2: Biblioteca Pandas: conceptos básicos

3.2.1 Introducción a Pandas

Pandas es una de las bibliotecas más importantes para trabajar con los datos. Provee herramientas de manipulación de datos diseñadas para hacer que la limpieza y el análisis de estos sean rápidos y fáciles en Python. A menudo, Pandas se usa en conjunto con herramientas informáticas numéricas, como NumPy y SciPy (para análisis exploratorio de datos), bibliotecas analíticas, como Statsmodels y Scikit-learn (sobre todo para *machine learning*), y bibliotecas de visualización de datos, como Matplotlib. Pandas adopta partes significativas del estilo de procesamiento basado en arreglos de NumPy, por lo que tener conocimiento de este último permite comprender mejor y más rápido la forma de trabajar con Pandas.

Mientras que Pandas adopta muchos modismos de codificación de NumPy, la mayor diferencia es que Pandas está diseñado para trabajar con datos tabulares o heterogéneos. NumPy, por el contrario, es el más adecuado para trabajar con datos de matrices numéricas homogéneas.

Desde que se convirtió en un proyecto de código abierto en 2010, Pandas maduró bastante hasta convertirse en una biblioteca bastante grande que es aplicable a un amplio conjunto de casos de uso de la vida real.

Siempre que vayamos a trabajar con Pandas, no hay que olvidar (al igual que con NumPy) importar la librería al espacio de trabajo.

```
>>> import Pandas as pd
```

Y también conviene importar DataFrame y Series, que son los dos principales objetos o estructuras de datos sobre los que trabaja esta librería.

```
>>> from pandas import Series, DataFrame
```

Muy importante prestar atención aquí a las mayúsculas porque los nombres de las librerías son *case sensitive*.

3.2.2 Estructuras de datos de Pandas

Para comenzar a trabajar con Pandas, hay que familiarizarse primero con los dos tipos de estructuras de datos con los que trabaja: las Series y los DataFrame. Si bien no son una solución universal para cada problema, proporcionan una base sólida y fácil de usar para la mayoría de las aplicaciones.

Series

Una Serie es un objeto tipo matriz unidimensional que contiene una secuencia de valores (de tipos similares a los de NumPy) y una matriz asociada de etiquetas de datos, denominada *índice*. La Serie más simple está formada por solo una matriz de datos:

Figura 36: Serie simple

```
In [1]: import pandas as pd
        from pandas import DataFrame, Series

In [2]: objeto1 = pd.Series([1, 3, -5, -7])
        objeto1

Out[2]: 0    1
        1    3
        2   -5
        3   -7
        dtype: int64
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Ya podemos ver una diferencia sustancial respecto de la visualización de los arreglos de NumPy. La representación de cadena de una Serie, como se muestra en el ejemplo anterior, muestra el índice a la izquierda y los valores a la derecha. Como no especificamos un índice para los datos, se crea automáticamente uno predeterminado que consta de los enteros 0 a *n* - 1 (donde *n* es la longitud de los datos). Se puede obtener la representación tipo arreglo de los valores de la Serie y los atributos del índice respectivamente:

Figura 37: Representación de los valores de la Serie y atributos del índice

```
In [3]: objeto1.values
Out[3]: array([ 1,  3, -5, -7], dtype=int64)

In [5]: objeto1.index  # se muestra como los parámetros de la función: range()
Out[5]: RangeIndex(start=0, stop=4, step=1)
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

A menudo, puede ser útil crear una Serie con un índice que identifique cada elemento de datos con una etiqueta:

Figura 38: Etiquetas

```
In [6]: objeto2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
        objeto2

Out[6]: d    4
        b    7
        a   -5
        c    3
        dtype: int64

In [7]: objeto2.index

Out[7]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

En comparación con las matrices NumPy, se pueden usar las etiquetas del índice al seleccionar los elementos de la Serie o un conjunto de elementos:

Figura 39: Selección de elementos o conjunto de elementos de la Serie

```
In [9]: objeto2['b']

Out[9]: 7

In [11]: objeto2[['b','c']]

Out[11]: b    7
         c    3
         dtype: int64
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

El uso de funciones NumPy u operaciones similares a NumPy, como el filtrado con una matriz booleana, la multiplicación o la aplicación de funciones matemáticas, preservará el valor índice de los elementos:

Figura 40: Preservación del valor índice de los elementos

```
In [12]: objeto2<0

Out[12]: d    False
         b    False
         a     True
         c    False
         dtype: bool

In [14]: objeto2*3

Out[14]: d    12
         b    21
         a   -15
         c     9
         dtype: int64
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Si se tiene un diccionario de Python, se puede crear una Serie a partir de él:

Figura 41: Serie a partir de diccionario de Python

```
In [23]: dict = {'kilobyte': 1024 , 'megabyte': 1048576, 'gigabyte': 1073741824}

In [24]: objeto3 = pd.Series(dict)
         objeto3

Out[24]: kilobyte          1024
         megabyte       1048576
         gigabyte    1073741824
         dtype: int64
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Las funciones **isnull** y **notnull** en Pandas se usan para detectar datos faltantes. Veamos un ejemplo con una Serie de nombres y edades en el cual, para el índice de etiqueta *Teo*, no se tiene el valor de edad en la Serie.

Figura 42: Función isnull

```
In [35]: objeto4=pd.Series([19,24,33,None,29], index=['Ana','Juan','Ema','Teo','Luis'])
         objeto4

Out[35]: Ana     19.0
         Juan    24.0
         Ema     33.0
         Teo      NaN
         Luis    29.0
         dtype: float64

In [36]: pd.isnull(objeto4)

Out[36]: Ana     False
         Juan    False
         Ema     False
         Teo      True
         Luis    False
         dtype: bool
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Identificar los valores faltantes en una Serie es un tema muy importante para la limpieza de datos anterior al armado, por ejemplo, de un modelo predictivo.

Con respecto a las operaciones que se pueden realizar con las Series, las matemáticas, como la suma, no pueden realizarse entre Series que poseen distinto índice. Si bien no arroja error, los índices de las Series se apilan, ya que, al no encontrar un punto en común en el otro índice de Serie, considera que no se pueden sumar dichos valores y arroja valores NaN. En el caso de coincidir algún índice entre ambas Series, el resultado será la suma de los índices coincidentes y la unión externa del resto de los índices no coincidentes, pero con valores NaN como resultado de la sumatoria de ellos.

Figura 43

```
In [42]: objeto2
Out[42]: d    4
         b    7
         a   -5
         c    3
         dtype: int64

In [43]: objeto1
Out[43]: 0    1
         1    3
         2   -5
         3   -7
         dtype: int64
```

```
In [44]: objeto1+objeto2
Out[44]: 0   NaN
         1   NaN
         2   NaN
         3   NaN
         a   NaN
         b   NaN
         c   NaN
         d   NaN
         dtype: float64
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Pero, si alteráramos la etiqueta del índice de una Serie para que coincida con la otra, se habilitará la posibilidad de hacer las operaciones que se deseen entre ambas.

Figura 44: Alteración de etiquetas

```
In [46]: objeto2.index=[0,1,2,3]
         objeto2
Out[46]: 0    4
         1    7
         2   -5
         3    3
         dtype: int64

In [47]: objeto1+objeto2
Out[47]: 0     5
         1    10
         2   -10
         3    -4
         dtype: int64
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Otro punto para tener en cuenta sobre las Series es que tanto el objeto Serie en sí como su índice tienen un atributo de nombre que se integra con otras áreas clave de las funcionalidades de pandas:

Figura 45: Atributo de nombre

```
In [39]: objeto4.index.name='nombre'

In [40]: objeto4.name='edad'
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

DataFrame

Un DataFrame representa una tabla rectangular de datos y contiene una colección ordenada de columnas, cada una de las cuales puede ser un tipo de valor diferente (numérico, cadena, booleano, etc.). El DataFrame tiene un índice de fila y de columna, y se puede considerar como un diccionario de Series que comparten el mismo índice.

Hay muchas formas de construir un DataFrame, aunque una de las más comunes es a partir de un diccionario de listas de igual longitud o matrices NumPy:

Figura 46: Diccionario de listas

```
In [55]: df=pd.DataFrame(data)
         df
```

Out[55]:

	provincia	poblacion	superficie en m/km
0	Buenos Aires	31250168	615.1
1	Catamarca	367828	102.6
2	Cordoba	3308876	165.3
3	Santa Fe	3194537	133.0
4	Mendoza	1738929	148.0
5	Capital Federal	2890151	0.2

Fuente: elaboración propia a base del *software* Anaconda (2018).

Para DataFrames grandes, el método ***head*** imprime solo sus primeras cinco filas. Esto es útil para cuando realizamos modificaciones cuyo impacto puede ser fácilmente contrastado en pocas filas. De esta manera, se pueden hacer las verificaciones con solo observar unas pocas filas del DataFrame y no todo completo. Si cinco filas no son suficientes, se puede asignar la cantidad de filas que el usuario considere pertinente.

Figura 47: Agregado de filas

```
In [56]: df.head()
```

Out[56]:

	provincia	poblacion	superficie en m/km
0	Buenos Aires	31250168	615.1
1	Catamarca	367828	102.6
2	Cordoba	3308876	165.3
3	Santa Fe	3194537	133.0
4	Mendoza	1738929	148.0

```
In [57]: df.head(3)
```

Out[57]:

	provincia	poblacion	superficie en m/km
0	Buenos Aires	31250168	615.1
1	Catamarca	367828	102.6
2	Cordoba	3308876	165.3

Fuente: elaboración propia a base del *software* Anaconda (2018).

Una columna de un DataFrame se puede recuperar como una Serie, ya sea por notación tipo diccionario o por atributo.

Figura 48: Columna de DataFrame recuperada como Serie

```
In [58]: df.provincia

Out[58]: 0       Buenos Aires
         1          Catamarca
         2            Cordoba
         3          Santa Fe
         4            Mendoza
         5    Capital Federal
         Name: provincia, dtype: object

In [59]: df['poblacion']

Out[59]: 0    31250168
         1      367828
         2     3308876
         3     3194537
         4     1738929
         5     2890151
         Name: poblacion, dtype: int64
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Las filas también se pueden recuperar por posición o nombre con el atributo **loc.**

Figura 49: Atributo loc

```
In [68]: df.loc[1]

Out[68]: provincia            Catamarca
         poblacion               367828
         superficie en m/km       102.6
         Name: 1, dtype: object
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Cuando se asignan listas o arreglos a una columna, la longitud del valor debe coincidir con la longitud del DataFrame. Si se asigna una Serie, sus etiquetas se realinearán exactamente al índice del DataFrame. Si se asigna a una columna existente, los valores se pisan. Si se escribe un nombre distinto (df['censo']), automáticamente se crea una columna nueva bajo ese nombre. En este caso, crearemos la columna *censo* y le asignaremos el año. Nótese que los índices estaban desordenados en la Serie, pero se acomodaron de acuerdo con los índices del DataFrame.

Figura 50: Columna censo

```
In [71]: anio_censo=pd.Series([2010,2010,2010,2001,2001,2001], index=[2,3,4,5,0,1])

In [73]: df['censo']=anio_censo
         df
```

Out[73]:

	provincia	poblacion	superficie en m/km	censo
0	Buenos Aires	31250168	615.1	2001
1	Catamarca	367828	102.6	2001
2	Cordoba	3308876	165.3	2010
3	Santa Fe	3194537	133.0	2010
4	Mendoza	1738929	148.0	2010
5	Capital Federal	2890151	0.2	2001

Fuente: elaboración propia a base del *software* Anaconda (2018).

Si quisiéramos, por ejemplo, modificar un dato en el DataFrame, se sigue una lógica similar. En este caso cambiamos la provincia Catamarca por Jujuy.

Figura 51: Modificación de un dato en el DataFrame

```
In [75]: correccion=pd.Series(['Buenos Aires', 'Jujuy', 'Cordoba', 'Santa Fe', 'Mendoza', 'Capital Federal'], index=[0,1,2,3,4,5])
         df['provincia']=correccion
         df
```

Out[75]:

	provincia	poblacion	superficie en m/km	censo
0	Buenos Aires	31250168	615.1	2001
1	Jujuy	367828	102.6	2001
2	Cordoba	3308876	165.3	2010
3	Santa Fe	3194537	133.0	2010
4	Mendoza	1738929	148.0	2010
5	Capital Federal	2890151	0.2	2001

Fuente: elaboración propia a base del *software* Anaconda (2018).

Hay otra forma mucho más eficiente y simple de hacerlo sin tener que escribir toda la Serie usando el comando **iloc**. Con él determinamos la posición de la fila y de la columna que queremos reemplazar porque trabaja con números enteros.

Por otro lado, tenemos el comando **loc**, que además nos brinda la posibilidad de identificar la etiqueta de la columna y el índice de la fila del elemento que queremos reemplazar. Veamos cómo con **iloc** modificamos la provincia Jujuy por Tierra del Fuego y cómo con **loc** cambiamos su población total.

Figura 52: Comandos loc e iloc

```
In [77]: df.iloc[1,0]='Tierra del Fuego'
         df
```

Out[77]:

	provincia	poblacion	superficie en m/km	censo
0	Buenos Aires	31250168	615.1	2001
1	Tierra del Fuego	367828	102.6	2001
2	Cordoba	3308876	165.3	2010
3	Santa Fe	3194537	133.0	2010
4	Mendoza	1738929	148.0	2010
5	Capital Federal	2890151	0.2	2001

```
In [79]: df.loc[1,'poblacion'] = 152317
         df
```

Out[79]:

	provincia	poblacion	superficie en m/km	censo
0	Buenos Aires	31250168	615.1	2001
1	Tierra del Fuego	152317	102.6	2001
2	Cordoba	3308876	165.3	2010
3	Santa Fe	3194537	133.0	2010
4	Mendoza	1738929	148.0	2010
5	Capital Federal	2890151	0.2	2001

Fuente: elaboración propia a base del *software* Anaconda (2018).

En este caso, en el comando **loc** el índice es un entero, pero, si llegáramos a nombrar los índices con etiquetas en texto, funcionaría del mismo modo.

3.2.3 Métodos aritméticos con DataFrames

Como vimos anteriormente, cuando queremos sumar dos Series, si las etiquetas de los índices no coinciden, se concatenan los índices y los valores correspondientes a esos índices no coincidentes quedan nulos (NaN), aunque alguno de los objetos posea un valor, y solo se suman aquellos valores cuyos índices se encuentren en ambas Series.

Con los DataFrames sucede lo mismo si las etiquetas no son coincidentes para las filas o para las columnas.

Figura 53. DataFrames

```
In [84]: df1 = pd.DataFrame(np.arange(20.).reshape((5, 4)),columns=list('abcd'))
         df1
```

Out[84]:

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0
3	12.0	13.0	14.0	15.0
4	16.0	17.0	18.0	19.0

```
In [85]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),columns=list('abcde'))
         df2
```

Out[85]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

Fuente: elaboración propia a base del *software* Anaconda (2018).

Partiendo de dos DataFrames de distinta forma df1 = (5,4) y df2 = (4,5) con índices coincidentes en filas y columnas, si hacemos la suma de ambos, veamos qué sucede:

Figura 54: Suma de DataFrames de distinta forma con índices coincidentes

```
In [90]: df1+df2
```

Out[90]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	11.0	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	27.0	29.0	31.0	33.0	NaN
4	NaN	NaN	NaN	NaN	NaN

Fuente: elaboración propia a base del *software* Anaconda (2018).

Como se puede ver, al hacer la suma de ambos, la fila 4 del df1 no se propaga al df de resultado, ya que df2 no tiene dicha fila con índice 4. Lo mismo ocurre con la columna "e" del df2, por lo que toda la fila 4 y la columna "e" quedan con valores nulos.

Para que esto no suceda, hay una función muy útil que permite imputar o asignar un valor determinado para que, al hacer la operación, la suma no se realice por un valor nulo, sino por otro que determinemos. En este caso, vamos a asignar cero:

Figura 55: Asignación de valor

```
In [91]: df1.add(df2, fill_value=0)
```

Out[91]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	11.0	13.0	15.0	9.0
2	18.0	20.0	22.0	24.0	14.0
3	27.0	29.0	31.0	33.0	19.0
4	16.0	17.0	18.0	19.0	NaN

Fuente: elaboración propia a base del *software* Anaconda (2018).

En la siguiente tabla, tenemos el resto de los métodos aritméticos para aplicar con DataFrames. Aquellos que tienen el prefijo "r" corresponden a la operación opuesta a la original.

Tabla 1: Métodos aritméticos para aplicar con DataFrames

Método	Descripción
add, radd	Método de agregación (+)
sub, rsub	Método de sustracción (−)
div, rdiv	Método de división (/)
floordiv, rfloordiv	Método de división con redondeo hacia abajo (//)
mul, rmul	Método de multiplicación (*)
pow, rpow	Método de potencia (**)

Fuente: elaboración propia.

Veamos un solo ejemplo:

Figura 56: Método aritmético de ejemplo

```
In [104]: df1 = pd.DataFrame(np.arange(5,46.,5).reshape((3, 3)),columns=list('abc'))
          df2 = pd.DataFrame(np.arange(50,91.,5).reshape((3, 3)),columns=list('abc'))

          df2.div(df1, fill_value=0)
```

Out[104]:

	a	b	c
0	10.000000	5.500	4.0
1	3.250000	2.800	2.5
2	2.285714	2.125	2.0

```
In [105]: df2.rdiv(df1, fill_value=0)
          # que es lo mismo que hacer df1.rdiv(df2, fill_value=0), cambiando de lugar los df
```

Out[105]:

	a	b	c
0	0.100000	0.181818	0.25
1	0.307692	0.357143	0.40
2	0.437500	0.470588	0.50

Fuente: elaboración propia a base del *software* Anaconda (2018).

Por otro lado, los métodos de NumPy que aplican sobre arreglos también aplican sobre los DataFrames. Como por ejemplo, el **método random** para crear arreglos o la función abs (para determinar el valor absoluto de un arreglo) sobre un DataFrame.

Figura 57: Método random y función abs

```
In [109]: dataf = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
                               index=['alfa', 'beta', 'gamma', 'delta'])

          dataf
```

Out[109]:

	b	d	e
alfa	-0.716421	-0.057149	-0.309167
beta	1.371805	-1.010344	0.228642
gamma	0.401594	1.346358	-0.737457
delta	0.316052	0.111750	-0.845972

```
In [110]: np.abs(dataf)
```

Out[110]:

	b	d	e
alfa	0.716421	0.057149	0.309167
beta	1.371805	1.010344	0.228642
gamma	0.401594	1.346358	0.737457
delta	0.316052	0.111750	0.845972

Fuente: elaboración propia a base del *software* Anaconda (2018).

Otra operación frecuente es aplicar una función de arreglos unidimensionales a cada columna o fila. El método **apply** de los DataFrames hace exactamente esto.

Tomemos de ejemplo una función anónima que llamaremos *func*, que calcula la diferencia entre el máximo y el mínimo de una Serie. Se invoca una vez por cada columna del DataFrame. El resultado es una Serie que tiene las etiquetas en las columnas y en su índice:

Figura 58: Aplicación de una función de arreglos unidimensionales a cada columna

```
In [110]: np.abs(dataf)
```

Out[110]:

	b	d	e
alfa	0.716421	0.057149	0.309167
beta	1.371805	1.010344	0.228642
gamma	0.401594	1.346358	0.737457
delta	0.316052	0.111750	0.845972

```
In [111]: func = lambda x: x.max() - x.min()

In [112]: dataf.apply(func)

Out[112]: b    2.088226
          d    2.356702
          e    1.074615
          dtype: float64
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

En cambio, si pasamos el parámetro axis = 'columns' para aplicar la función, se invocará una vez por fila en vez de por columna, quedando por etiqueta de índice de fila la original.

Figura 59: Aplicación de una función de arreglos unidimensionales a cada fila

```
In [113]: dataf.apply(func, axis='columns')

Out[113]: alfa     0.659271
          beta     2.382149
          gamma    2.083815
          delta    1.162024
          dtype: float64
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

3.2.4 Estadística descriptiva

Los objetos en Pandas están equipados con un conjunto de métodos matemáticos y estadísticos comunes. La mayoría de estos pertenece a la categoría de reducciones o estadísticas descriptivas, métodos que extraen un solo valor (como la suma o la media) de una Serie o una Serie de valores de estadísticos de filas o columnas en un DataFrame. En comparación con los métodos similares que se encuentran en NumPy, en Pandas tienen un método incorporado para tratar los datos faltantes.

Consideremos como ejemplo un pequeño DataFrame y hagamos unos estadísticos de agregación simple por fila y por columna:

Figura 60: Estadísticos de agregación simple por fila y por columna

```
In [6]: df = pd.DataFrame([[2.2, np.nan], [-6.1, -4.1], [np.nan, np.nan], [-1.75, 3.3]],
                          index=['a', 'b', 'c', 'd'],
                          columns=['rojo', 'azul'])
        df
```

Out[6]:

	rojo	azul
a	2.20	NaN
b	-6.10	-4.1
c	NaN	NaN
d	-1.75	3.3

```
In [7]: df.sum()
Out[7]: rojo   -5.65
        azul   -0.80
        dtype: float64

In [8]: df.sum(axis='columns')
Out[8]: a     2.20
        b   -10.20
        c     0.00
        d     1.55
        dtype: float64
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Los valores de nulos se excluyen, a menos que todo el segmento (ya sea fila o columna) sea nulo. Esta característica se puede deshabilitar con la opción **skipna**:

Figura 61: Skipna

```
In [9]: df.mean(axis='columns', skipna=False)
Out[9]: a      NaN
        b   -5.100
        c      NaN
        d    0.775
        dtype: float64
```

Fuente: elaboración propia a base del *software* Anaconda (2018).

Algunos métodos, como **idxmin** e **idxmax**, devuelven estadísticas indirectas, como el valor del índice donde se alcanzan los valores mínimos o máximos. Pero quizás el más práctico y usado es el método **describe**, que nos permite ejecutar de manera rápida y con un solo comando los estadísticos principales de un DataFrame por columna.

Figura 62: Método describe

```
In [11]: df.idxmax()
Out[11]: rojo    a
         azul    d
         dtype: object

In [12]: df.describe()
Out[12]:
```

	rojo	azul
count	3.000000	2.00000
mean	-1.883333	-0.40000
std	4.151606	5.23259
min	-6.100000	-4.10000
25%	-3.925000	-2.25000
50%	-1.750000	-0.40000
75%	0.225000	1.45000
max	2.200000	3.30000

Fuente: elaboración propia a base del *software* Anaconda (2018).

Unique values y value_counts

Luego tenemos otras clases de métodos que son sumamente útiles a la hora de analizar los datos. Por un lado, tenemos el método **unique**, que nos permite determinar los valores únicos en una Serie unidimensional.

```
>>> nombre_objeto.unique()
```

Su resultado es una nueva Serie con los valores únicos de la Serie principal.

Por otro lado, tenemos el método **value_counts**, que nos devuelve las frecuencias de repetición de cada valor único dentro de una Serie.

```
>>> pd.value_counts(nombre_objeto.values, sort=False)
```

Como estos métodos aplican sobre objetos unidimensionales, aquí es donde, si queremos utilizarlos en un DataFrame, debemos recurrir al método combinado con **apply** para que se ejecute por cada columna de la tabla.

Figura 63: Métodos combinados 1

```
In [58]: diccionario= {'Q1': [1, 3, 4, 4, 4],
                       'Q2': [2, 3, 4, 4, 3],
                       'Q3': [1, 5, 4, 4, 4]}

In [59]: tabla = pd.DataFrame(diccionario,
                              columns=['Q1','Q2','Q3'],
                              index=['a','b','c','d','e'])
         tabla
```

Out[59]:

	Q1	Q2	Q3
a	1	2	1
b	3	3	5
c	4	4	4
d	4	4	4
e	4	3	4

Fuente: elaboración propia a base del *software* Anaconda (2018).

Figura 64: Métodos combinados 2

```
In [60]: tabla.apply(pd.value_counts).fillna(0)
```

Out[60]:

	Q1	Q2	Q3
1	1.0	0.0	1.0
2	0.0	1.0	0.0
3	1.0	2.0	0.0
4	3.0	2.0	3.0
5	0.0	0.0	1.0

Fuente: elaboración propia a base del *software* Anaconda (2018).

Como se puede ver en el ejemplo, a la izquierda, en lugar de los índices, están los valores que se repiten dentro de la tabla. Luego, en cada posición de columna, está el valor de la frecuencia —o repetición—, es decir, de la cantidad de veces que aparece ese valor en la columna. Este método es sumamente útil para el análisis de los datos, para armado de histogramas de visualización y para depurar errores en caso de que se trate de valores únicos que no pueden repetirse en un *dataset*.

El **método apply**, además, tiene una Serie de parámetros que permite aplicar la función o el método por fila o por columna, dependiendo de la necesidad. Esto se logra definiendo el parámetro **axis** como 'rows'(axis=0) o 'columns'(axis=1).

Con el método **fillna** asignamos el valor cero, en caso de que los resultados arrojen valores nulos, para que en el cuadro no se visualicen los valores NaN. Por ejemplo, el valor 2 dentro de la columna Q1 no aparece nunca; sin embargo, en vez de mostrarse como NaN, se visualiza como cero. Esta es una manera de hacer más prolija la exposición de las estadísticas, pero también sirve para no generar errores ni entorpecer los valores nulos en los resultados si luego necesitamos hacer operaciones con la tabla resultante.

Con este módulo obtuvimos las herramientas principales para comenzar a sacar provecho realmente de los datos. Comenzaremos a implementarlas detalladamente en el siguiente módulo, donde el trabajo se volverá mucho más interesante.

Referencias

Anaconda. (2018). Anaconda (Versión 3.7) [*Software* de computación]. Austin, US.

Ávila, D. (s. f.). [Imagen sin título de matriz de 6 × 6]. Recuperado de https://damianavila.github.io/Python-Cientifico-HCC/3_NumPy.html

Solomon, B. (2017). ¿Qué es la vectorización? Recuperado de https://www.it-swarm.dev/es/python/que-es-la-vectorizacion/834957754/