

# INTRODUCCIÓN A PYTHON

Carlos Fernández Musoles

## MÁSTER EN INTELIGENCIA ARTIFICIAL

Módulo I. Introducción a la inteligencia artificial



**Universidad**  
Internacional  
de Valencia

Este material es de uso exclusivo para los alumnos de la Universidad Internacional de Valencia. No está permitida la reproducción total o parcial de su contenido ni su tratamiento por cualquier método por aquellas personas que no acrediten su relación con la Universidad Internacional de Valencia, sin autorización expresa de la misma.

**Edita**

Universidad Internacional de Valencia

# Máster en **Inteligencia Artificial**

---

## **Introducción a Python**

Módulo I. Introducción a la inteligencia artificial  
6 ECTS

---

**Carlos Fernández Musoles**

# Leyendas

---



Enlace de interés



Ejemplo



Importante

---

**abc** Los términos resaltados a lo largo del contenido en color **naranja** se recogen en el apartado **GLOSARIO**.

---

<b>CAPÍTULO 1. INTRODUCCIÓN</b>	7
1.1. Contexto	7
1.2. Por qué Python	8
1.3. Convenciones	9
1.4. Librerías más populares y útiles	10
1.5. Instalación	10
1.6. Python 2 vs. Python 3	11
1.7. Contenidos del curso	11
<b>CAPÍTULO 2. PYTHON 101</b>	12
2.1. Intérprete básico	12
2.2. Ejecución de scripts	13
2.3. Jupyter Notebook	14
2.4. Sintaxis básica	15
2.4.1. Variables	15
2.4.2. Comentarios	17
2.4.3. Comparaciones condicionales	17
2.4.4. Loops	18
2.4.5. Funciones	20
2.5. Colecciones en Python	23
2.5.1. Tuplas	23
2.5.2. Listas	23
2.5.3. Secuencias	24
2.5.4. Diccionarios	25
2.5.5. Sets (conjuntos)	26
2.5.6. Python comprehension	27
<b>CAPÍTULO 3. COLECCIONES: NUMPY</b>	29
3.1. Objeto básico en NumPy: ndarray	29
3.1.1. Dimensiones	30
3.1.2. Manipulación de ndarrays	31
3.1.3. Índices y slicing	32
3.2. Funciones matemáticas sobre colecciones	32
3.3. Filtrado de datos	34
3.4. Exploración y estadística descriptiva	34
3.5. Álgebra lineal	35
3.6. Valores aleatorios	36

<b>CAPÍTULO 4. ESTRUCTURAS DE DATOS: PANDAS</b>	38
4.1. Series	39
4.2. DataFrame	41
4.3. Trabajo con Series y DataFrame	44
4.3.1. Búsqueda	44
4.3.2. Indexación y slicing	44
4.3.3. Ordenación de Series y DataFrame	46
4.4. Operaciones con Series y DataFrame	47
4.5. Estadística con Series y DataFrame	49
<b>CAPÍTULO 5. VISUALIZACIÓN: MATPLOTLIB</b>	51
5.1. matplotlib 101	52
5.2. Múltiples gráficos	54
5.3. Decoraciones y anotaciones	56
5.4. Tipos de gráficos	57
5.5. Grabación de gráficos a archivo	57
5.6. Representación con pandas	58
<b>CAPÍTULO 6. PYTHON PARA CIENCIA DE DATOS</b>	61
6.1. Lectura y escritura de archivos	61
6.1.1. En Python	62
6.1.2. En NumPy	64
6.1.3. En pandas	64
6.2. Trabajo con strings	65
6.3. Limpieza de datos	67
6.3.1. Aplicación de funciones a una colección	67
6.3.2. pandas para transformar datos	68
<b>GLOSARIO</b>	71
<b>ENLACES DE INTERÉS</b>	75
<b>BIBLIOGRAFÍA</b>	76

01

```

.....
mirror_mod.use_x = False
mirror_mod.use_y = True
mirror_mod.use_z = False
operation == "MIRROR_Z"
mirror_mod.use_x = False
mirror_mod.use_y = False
mirror_mod.use_z = True

[ (1+x+y+2a)-(3a+3g+x)
  5+x+k+2a+21
  E=mc2
  1lim h-->0
  2+....+2a....+a
  selection at the end -add
  _ob.select= 1
  ler_ob.select=1
  text.scene.objects.acti
  ("Selected" + str(modifier
  error_ob.select = 0
  bpy.context.selected_ob
  sta.objects[one.name].se
  Int("please select exactl
  1+x+y+2a
  2+...+2a+3a
  1+x+y+2a

```

# Capítulo 1

## Introducción

### 1.1. Contexto

El aumento del interés en inteligencia artificial (IA) actual está en gran medida propulsado por una mejora cualitativa en los algoritmos de machine learning (aprendizaje automático).

Esta mejora ha sido posible gracias a dos cambios fundamentales acontecidos en los últimos quince años: el incremento de la capacidad computacional de los ordenadores actuales (gracias, en parte, al paralelismo de las GPU y a la computación en la nube) y la cantidad descomunal de datos disponibles para entrenar algoritmos (con todas las grandes empresas ofreciendo servicios a cambio de datos).

Pero de nada sirve disponer de petabytes (millones de gigabytes) de información y tener a nuestra disposición computadoras en la exaescala (capaces de ejecutar billones de billones de operaciones por segundo) si no hablamos el lenguaje de las máquinas. En este sentido, cobra mucha relevancia dominar la programación por ordenador.

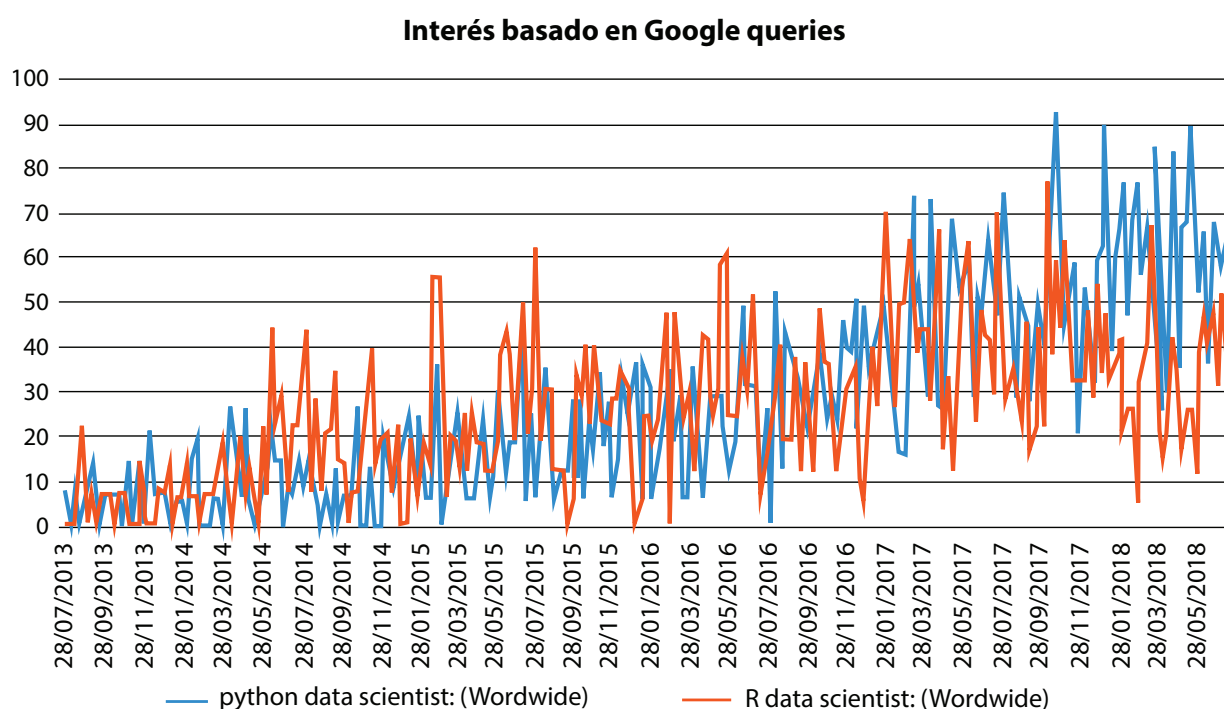
La IA está llamada a ser la cuarta revolución industrial (López, 2018), ya que es capaz de hacer cosas a una escala que antes era impensable. Y para formar parte de esta revolución altamente técnica, es necesario programar.

## 1.2. Por qué Python

Una pregunta frecuente que suele rondar la mente de los que quieren iniciarse en el mundo de la IA es: ¿cuál es el mejor lenguaje de programación para la IA? Responder a esta pregunta normalmente requiere atender a varios niveles de complejidad, ya que puede referirse a qué lenguaje es cualitativamente mejor (algo casi imposible de evaluar) o qué lenguaje es más sencillo o aplicable. Sin embargo, la mayor parte del tiempo, la pregunta tiene un tono más utilitario. Los que se formulan esta pregunta desean saber qué lenguaje tiene más adopción en la industria de IA o, dicho de otro modo, con cuál se le van a abrir más puertas.

Dejando de lado las discusiones cuasifilosóficas, hay muchos lenguajes de programación adecuados para la IA. Desde los generalistas C, C++ y Java a los más aplicados como Lisp o Prolog, podríamos decir que cada lenguaje tiene su aplicación ideal dentro de la IA. Sin embargo, hay dos que destacan por su popularidad: **Python** y **R**.

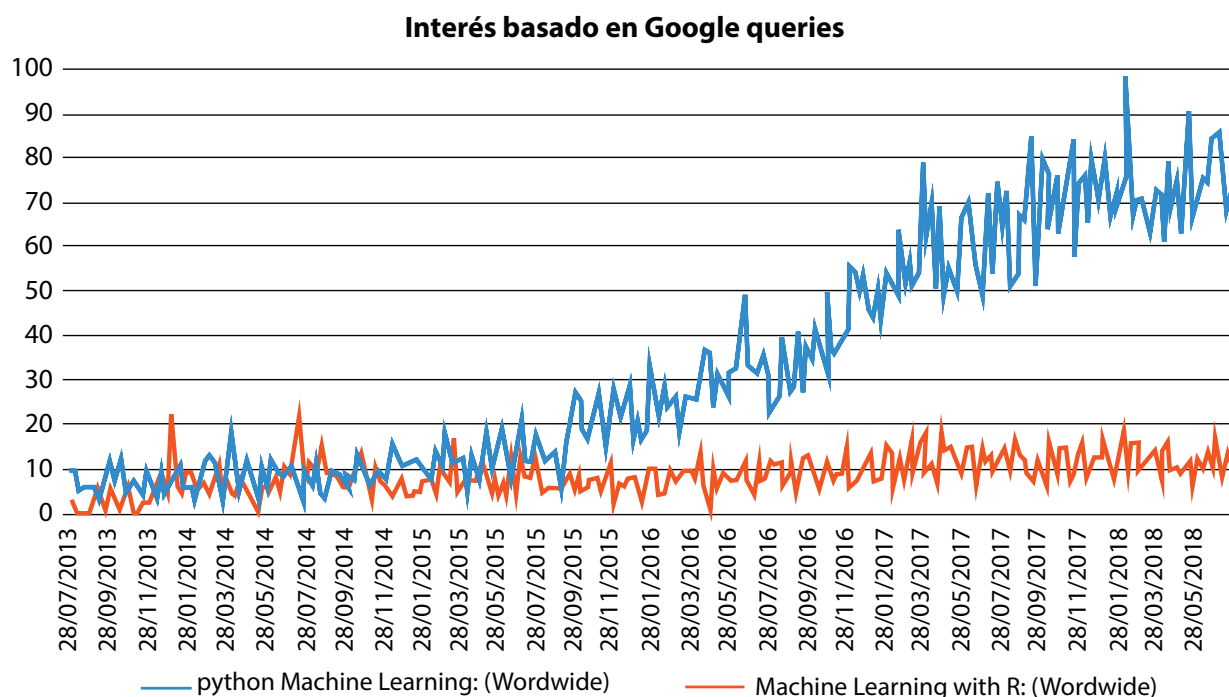
Tanto Python como R son muy demandados en big data. La Figura 1 muestra el nivel de popularidad basado en búsquedas de Google de los términos “Python” y “R” seguidos de “data scientist”. Como vemos, ambos lenguajes se asocian con la ciencia de datos frecuentemente. En este respecto, ambos son opciones perfectamente válidas, con sus ventajas e inconvenientes (BBVA Open4U, 2016).



**Figura 1.** Comparativa de la popularidad relativa de los términos “python data scientist” y “R data scientist” en búsquedas de Google.

Python es mucho más comúnmente utilizado en el área del aprendizaje automático, como indica la Figura 2. En gran medida, esto es el resultado de varios factores, entre los que se encuentran la facilidad de uso (sintaxis similar al inglés, sin necesidad de compilación, etc.) y la increíble asistencia que ofrece su comunidad (tanto en discusión de ideas como en ayuda para solventar problemas y acceso a herramientas en forma de librerías).





**Figura 2.** Comparativa de la popularidad relativa de los términos “Python machine learning” y “Machine learning with R” en búsquedas de Google.

R es un lenguaje formulado y pensado para el tratamiento estadístico y el manejo de datos. Así pues, es una elección popular en las matemáticas estadísticas y en ciertas áreas de la IA (como la ciencia de datos y el aprendizaje no supervisado). Sin embargo, carece del carácter generalista de Python, así como de su rico ecosistema.

En este curso nos centraremos en Python como lenguaje unificador dentro de la inteligencia artificial, sin que esta elección signifique que Python es innatamente superior a cualquier otro lenguaje. Como en otras áreas de la tecnología, el lenguaje de programación debe ajustarse a la tarea que se quiere completar. El caso es que, dado su rico ecosistema, Python es la elección más adecuada como común denominador dentro de la programación en IA. Como ejemplo, la inmensa mayoría de las librerías más utilizadas en la investigación en aprendizaje automático están escritas en Python o tienen interfaces para el mismo: Scikit-learn (Pedregosa, Varoquaux, Gramfort, Michel y Thirion, 2011), TensorFlow (Abadi et al., 2016), Theano (Bergstra et al., 2011), Caffe (Jia et al., 2014), Keras (Chollet, 2015).

### 1.3. Convenciones

En este manual, seguiremos las siguientes convenciones en cuanto a formatos de texto:

- Fuente normal en cursiva para links, ficheros, nombres y extensiones.
- Fuente Consola para código.
- **Fuente Consola en negrita para comandos que el usuario debe escribir literalmente.**
- Fuente Consola en cursiva para texto que debe ser reemplazado por el usuario en ejemplos y output de programas o consola de textos.

## 1.4. Librerías más populares y útiles

Python está en plena madurez como lenguaje, y prueba de ello es la cantidad de librerías desarrolladas que enriquecen su ecosistema. A continuación listamos varias de las más populares, tanto que casi se convierten en una extensión natural del lenguaje (en **negrita** los que cubriremos explícitamente en este curso):

- **NumPy**: Numerical Python, para computación numérica.
- **pandas**: estructuras de datos.
- **matplotlib**: gráficos y diagramas para visualización de datos.
- **IPython y Jupyter**: computación explorativa e interactiva.
- **SciPy**: computación científica (optimización, estadística, álgebra lineal...).
- **Scikit-learn**: aprendizaje automático general.
- **statsmodel**: análisis estadístico.

En lo relativo a importar paquetes externos (que dotan a Python de funcionalidad añadida), para las librerías más populares utilizaremos nomenclaturas que se han adoptado como estándar:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

## 1.5. Instalación

El proceso de instalación de Python depende en gran medida del sistema operativo empleado. Aunque Python se puede instalar independientemente, se recomienda descargar e instalar Anaconda, un paquete de software que incluye Python y otras herramientas útiles en el desarrollo, algunas de las cuales veremos en el resto del curso (**NumPy, pandas, matplotlib**).



### Enlace de interés

Descarga e instrucciones para la instalación del paquete Anaconda.

<http://www.anaconda.com>

Para comprobar que se ha instalado correctamente, se puede abrir la línea de comandos en Windows **cmd** (o la consola en Linux y MacOS) e introducir el comando **python** (o **ipython** en MacOS) para iniciar el intérprete:

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit
(AMD64)] on win32
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>>
```

Para salir del intérprete, se puede introducir el comando **exit()** o bien pulsar **Ctrl + D**.

## 1.6. Python 2 vs. Python 3

Python nació en 1991. En 2008, fue publicada la tercera gran versión del lenguaje, Python 3. Hasta entonces, las versiones más populares eran las 2.x, y por ello todas las librerías estaban escritas para funcionar en esas versiones. La versión de Python 3 trajo consigo muchos cambios, algunos de los cuales impedían la retrocompatibilidad del código escrito en Python 2.x. Por ese motivo, a pesar de la nueva versión, hasta hace bien poco Python 2.x continuó siendo la versión favorita de muchos programadores y desarrolladores de librerías.

Con el paso del tiempo, la mayoría de librerías han sido portadas a Python 3.x y hoy en día es la versión considerada estándar. En este curso nos centraremos en Python 3.x. Concretamente, asumiremos la versión 3.6, aunque el código debería ser compatible con versiones más avanzadas.

## 1.7. Contenidos del curso

Este curso se centrará en proporcionar las bases para un conocimiento general del lenguaje Python, que luego se utilizará para explorar el entorno SciPy, un ecosistema gratuito en Python para las matemáticas, la ingeniería y la ciencia en general.

Tras ver los fundamentos de Python, su sintaxis básica y las colecciones más comunes, se hará un repaso específico de las librerías más importantes del entorno SciPy: Jupyter Notebook (archivos interactivos de IPython), **NumPy** (colecciones multidimensionales y operaciones matemáticas), **matplotlib** (visualización gráfica de datos), **pandas** (estructuras de datos) y H5py (formato HDF5 en Python). Cada una de estas librerías es de uso común en análisis de datos, minería de datos y aprendizaje automático. Todos los ejemplos utilizados se enmarcan en la aplicación para su uso en estos dominios.

Para simplificar, el manual se referirá a **terminal** sin distinción de sistema operativo para indicar la línea de comandos (en Windows) o la terminal de comandos (en Linux y MacOS).

## Capítulo 2

# Python 101

A diferencia de otros lenguajes, como C, C++ o Java, Python es un **lenguaje interpretado**, lo que significa que no hace falta compilarlo (traducirlo a lenguaje de máquina) para ejecutarlo. Hay varias formas de ejecutar código en Python. En esta sección veremos las más comunes: el intérprete básico, scripts y Jupyter Notebook.

## 2.1. Intérprete básico

El método más básico de ejecutar código Python es utilizando el intérprete básico. Para ello, se introduce el comando **python** en la terminal. Esto abre el intérprete de Python, en el que se puede escribir, línea por línea, código Python.



```
c:\Windows>python
```

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914  
64 bit (AMD64)] on win32
```

&gt;&gt;&gt;

```
>>> Type "help", "copyright", "credits" or "license" for more information.

>>> a = 5
>>> b = a * 2
>>> print(b)

10
```

## 2.2. Ejecución de scripts

El comando **python** también se puede utilizar para ejecutar **scripts**, que son archivos de texto que agrupan múltiples líneas de código. La utilización de scripts facilita la creación de bloques de código más complejos. Para ello, se pasa como parámetro el nombre del archivo **python script.py**.



### Ejemplo

Archivo **test.py** continene el siguiente texto:

```
a=5
b=a*2
print(b)
```

El siguiente comando produce el mismo output que en el ejemplo anterior, 10:

```
python test.py
```



Por convención, los scripts deben tener una extensión **.py** para indicar que son archivos ejecutables por el intérprete.

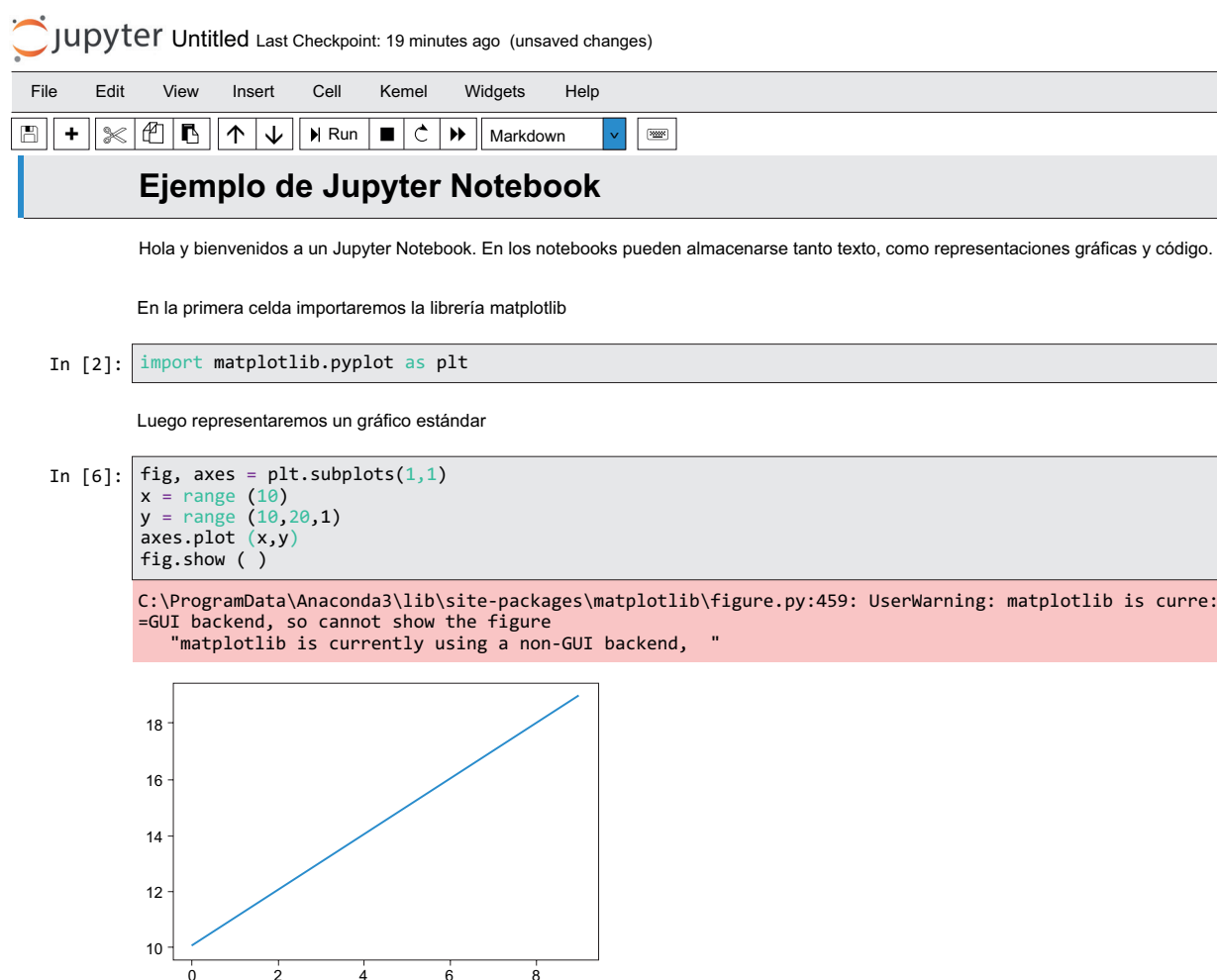
Como los scripts son simples archivos de texto, se puede utilizar cualquier editor para generarlos. Sin embargo, hay editores de textos especializados, además de IDE (entornos de desarrollo integrados), que pueden facilitar la escritura de código Python con características como resaltado de sintaxis, asistencia en el proceso de **debugging** (trazado del código en búsqueda de errores), etc.

Visual Studio Code es una opción recomendable por su disponibilidad (gratis en todas las plataformas) y su funcionalidad, pero existen muchas más (Geany, Sublime, Notepad++, etc.).

Como entornos de desarrollo se encuentran Spyder, PyCharm, entre otros.

## 2.3. Jupyter Notebook

Jupyter es una iniciativa de código abierto para el desarrollo interactivo de programas en múltiples lenguajes. Jupyter Notebook es una aplicación web que permite crear y compartir documentos que contienen imágenes, texto y código ejecutable. Es una aplicación muy utilizada en la comunidad científica como herramienta de facilitación de la colaboración y el desarrollo iterativo en Python. La figura 3 muestra un ejemplo de un Jupyter Notebook en Python.



**Figura 3.** Ejemplo de Jupyter Notebook.

Un Jupyter Notebook consiste en una serie de celdas consecutivas. Cada celda puede tener uno de los siguientes tipos: *Code* (para código ejecutable), *Markdown* (para texto con formato), *Headings* (para encabezados; es una celda markdown con el carácter # al inicio) o *Raw NBConvert* (para texto sin formato). Se pueden crear celdas con el menú *Insert > New Cell* o con las teclas **alt + enter**. Todas las celdas son ejecutables (con el botón *Run* o con las teclas **shift + enter**). Las celdas markdown formatean el texto incluido al ejecutarse, mientras que las celdas code utilizan IPython, un intérprete de Python avanzado, para ejecutar el código.

Las sesiones escritas en el Notebook pueden grabarse en un archivo **.pynb** para acceso futuro, con el menú *File > Save and Checkpoint*. Esto permite compartir resultados fácilmente con otros colaboradores.



### Enlace de interés

Aunque está fuera del temario de este curso, es posible compartir los notebooks manteniéndolos en un servidor remoto. De forma similar, Google ofrece una herramienta gratuita para mantener notebooks en su plataforma y así poder compartirlos con otros.

<https://colab.research.google.com/notebooks/welcome.ipynb>

## 2.4. Sintaxis básica

### 2.4.1. Variables

En programación, el uso de **variables** es fundamental. Una variable es una referencia a un valor; se puede entender como un cajón que contiene un elemento. En Python hay cinco **tipos de elementos** básicos: `float` (números reales), `int` (números enteros), `str` (texto), `bytes` (valores binarios) y `bool` (`True` o `False`). Además, existen dos tipos especiales: `None` (que indica ausencia de valor) y `NaN` (para indicar que una variable numérica no tiene un valor numérico, del inglés "Not a Number"). A diferencia de otros lenguajes, las variables no tienen un tipo estático, sino que el tipo es implícito y puede cambiar a lo largo del programa.

Para crear una variable, se escribe el nombre de la variable (debe ser único y empezar por un carácter alfabético) y se le asigna un valor con el signo `=`. Para asignar un valor textual (`string`), se pasa el valor entrecomillado (tanto el entrecomillado simple `' '` como el doble `" "` son aceptables). Una vez creadas, las variables se pueden utilizar en expresiones, del mismo modo en el que se usan en ecuaciones matemáticas. La Tabla 1 muestra un listado de operaciones binarias (entre dos elementos) en Python.

**Tabla 1**  
*Operaciones binarias en Python*

Operación	Descripción
<code>a + b</code>	suma a y b
<code>a - b</code>	resta a menos b
<code>a / b</code>	a dividido entre b
<code>a // b</code>	a dividido entre b (quitando decimales)
<code>a * b</code>	a multiplicado por b
<code>a ** b</code>	a elevado a b
<code>a % b</code>	devuelve el resto de la división a / b (modulus)
<code>a &amp; b</code>	bool: <code>True</code> si a y b son <code>True</code> int: bitwise AND
<code>a   b</code>	bool: <code>True</code> si a o b es <code>True</code> int: bitwise OR
<code>a ^ b</code>	bool: <code>True</code> si solo uno de los dos es <code>True</code> int: bitwise EXCLUSIVE OR
<code>a == b</code>	<code>True</code> si a y b son iguales
<code>a != b</code>	<code>True</code> si a y b son distintos
<code>a is b</code>	<code>True</code> si a y b son referencias al mismo objeto
<code>a is not b</code>	<code>True</code> si a y b son referencias a distintos objetos
<code>a &lt;= b, a &lt; b</code>	<code>True</code> si a es menor o igual que b (o solo menor)
<code>a &gt;= b, a &gt; b</code>	<code>True</code> si a es mayor o igual que b (o solo mayor)



```
str = 'hola mundo'
number1 = 42
number2 = 12
result = number1 - number2
print(result)
comb = str(number1) + str
print(comb)
number = 'forty two'
print(number1)
```

**Programa 1.** Ejemplo básico de programa Python.

```
30
42hola mundo
forty two
```

**Salida 1.** Salida del ejemplo de programa Python.

Como se puede ver en el ejemplo anterior, para aplicar operaciones (en este caso +, que entre objetos `string` se interpreta como concatenación) entre dos variables de distinto tipo, si la conversión no es obvia, se puede convertir (lo cual se denomina **casting**) una de ellas al tipo de la otra.

En este caso, hemos convertido un valor `int` a `str` utilizando la expresión `str(number)`. Del mismo modo, podemos hacer casting a cualquier tipo utilizando `int()`, `float()`, `bool()`, `str()`.



La mayoría de los tipos básicos son interconvertibles. Sin embargo, hay castings que son inviables, como por ejemplo `int("a")`. En esos casos, el intérprete retornará un error de tipo `ValueError`, lo que indica que no se ha podido convertir el valor.

Aunque esto se verá con más detalle más adelante, se pueden crear objetos lista (`list`), que son una colección de elementos del mismo tipo. La forma más básica de crear una lista es con `[ ]`:

```
lista1 = [1,5,-2]
lista2 = ['cefalopodo', 'mamifero', 'ave']
```

Cuando se quiere comprobar si dos variables son iguales, es preciso recalcar las diferencias entre comparar los elementos dentro de la variable y comparar las variables entre sí. Para comparar el valor de las variables, podemos usar `a == b`; si lo que queremos es comparar si dos variables hacen referencia al mismo objeto, debemos usar `a is b`.





### Ejemplo

```
lista1 = [1,2,3]
lista2 = [1,2,3]
print(lista1 == lista2)
print(lista1 is lista2)
```

Output:

```
True
False
```

## 2.4.2. Comentarios

Los comentarios son partes del código que son ignoradas por el intérprete y, por lo tanto, ni se evalúan ni se ejecutan. Sirven para aportar información al lector sobre el funcionamiento del código, aunque también se utilizan durante el desarrollo para testear partes del código. En cualquier caso, son una pieza fundamental de cada programa, ya que aumentan la legibilidad del código si se usan adecuadamente. En Python, todos los caracteres escritos tras el símbolo `#` son considerados comentarios. Pueden aparecer al principio de la línea o tras una expresión.

## 2.4.3. Comparaciones condicionales

Si las variables son el primer elemento fundamental en los programas, las **comparaciones condicionales** son el segundo. A base de comparaciones condicionales se puede dirigir el flujo de ejecución para ejecutar una parte del programa en vez de otra. La sintaxis es la siguiente:

```
if <comparación>:
    <expresión1>
else:
    <expresión2>
```

En `<comparación>` se puede poner cualquier expresión que evalúe a `bool` (`True` o `False`). Si el valor es `True`, se ejecutará la `<expresión1>`; si es `False`, se ejecutará la `<expresión2>`. Ambas expresiones pueden contener múltiples frases.

```
a = 10
b = 5
if a < b:
    print("a is smaller")
else:
    printf("a is bigger or equal")
```

**Programa 2.** Uso de condicionales.



```
a is bigger or equal
```

**Salida 2.** Salida condicional, dado que a es > que b.



En Python no se utilizan paréntesis para estructurar el código, sino que se usa la indentación para crear bloques. En el caso de las comparaciones, ambas expresiones han de ir indentadas (al menos un espacio, pero todas las frases dentro de un bloque deben usar la misma indentación). Es comúnmente aceptado utilizar cuatro espacios o una tabulación como indentación de bloque. Si se necesita crear un bloque dentro de un bloque (por ejemplo, un `if` dentro de un `while` loop), las indentaciones son acumulables).

Se pueden concatenar comparaciones sustituyendo `else` por `elif <comparación2>:` para condicionales más complejos.

Aunque menos legible, hay una forma compacta de expresar comparaciones, el **operador ternario**. Se suele utilizar cuando las expresiones a ejecutar son muy sucintas, aunque se debe evitar si complica demasiado la legibilidad del código. La estructura del operador ternario equivalente al condicional anterior es:

```
<expresión1> if <comparación> else <expresión2>
```

## 2.4.4. Loops

Los **loops** permiten a los programas ejecutar código de forma iterativa o repetitiva. Hay dos tipos de expresiones que resultan en loops: `for` y `while`.

Los `for` loops se utilizan para iterar sobre una colección o secuencia, elemento a elemento. Tienen la siguiente forma:

```
for i in <secuencia>:  
    <expresión>
```

`i` toma el valor de un elemento dentro de la `<secuencia>` en cada iteración.

Los `while` loops se utilizan para ejecutar una parte del código mientras una condición sea cierta. Tienen la siguiente forma:

```
while <condición>:  
    <expresión>
```

Mientras la <condición> sea True, se ejecutará la <expresión>. Para evitar loops infinitos, la <expresión> debe modificar algún elemento en algún momento para que la <comparación> evalúe False.

Una función muy útil en loops (especialmente for) es range:

`range(min,max,step)`

`range` retorna una lista de elementos, desde `min` (inclusivo) hasta `max` (exclusivo), en incrementos definidos por `step`. El único argumento requerido es `max`. Si `min` no se incluye, se asume que es 0; `step` se asume 1. La llamada `range(1,10,2)` retorna la secuencia [1,3,5,7,9]. Esta secuencia se puede usar en el `for` loop para determinar el número de iteraciones.



### Ejemplo

```
# for loop para imprimir todos los elementos de una lista
for n in range(10):
    print(n)
# while loop para imprimir los números pares menores de 10
a = 1
while a < 10:
    if a % 2 == 0:
        print(a)
    a = a + 1
```

En ocasiones es deseable saltar una iteración, es decir, parar la ejecución de la iteración actual y pasar a la siguiente. Esto se puede realizar con la palabra clave `continue`. También es posible salir de un loop forzosamente con la palabra clave `break`.



### Ejemplo

Otra forma de escribir el `while` loop anterior que imprime los números pares menores de 10 es:

```
a = 1
while True:
    if a % 2 != 0:
        continue
    if a >= 10:
        break
    print(a)
    a++
```



Es importante apreciar que la diferencia entre ambos es simplemente semántica y que, por lo tanto, cualquier loop se puede expresar tanto con un `for` como con un `while` (aunque, dependiendo de la intención, uno será más legible y adecuado que el otro).

## 2.4.5. Funciones

Las **funciones** o **métodos** son formas de estructurar el código. Se utilizan para reutilizar código (por ejemplo, una función que se encargue de comprobar si un número es par) y también para separar lógicamente segmentos de código cuya funcionalidad está muy acotada (por ejemplo, un método que escanee una lista y reemplace ciertos elementos por 0).

Para definir una función antes de su utilización, se debe declarar:

```
def nombre(<lista de parámetros>):  
    <expresión>  
    return <lista de valores> # opcional
```

El nombre de la función puede ser cualquier nombre (siguiendo las mismas normas que para las variables). Cada función puede tener ninguno o más parámetros, indicados en `<lista de parámetros>` y separados por comas. Las funciones pueden acabar sin retornar un valor, pero pueden retornar uno o más valores, separados por comas.

```
def divisible_by(mylist,divisor):  
    divisible= []  
    for n in mylist:  
        divisible.append(n % divisor == 0)  
    return divisible, len(divisible)  
lista1 = list(range(9))  
divs, count = evens(lista1,2)  
print(count)  
print(divs)
```

**Programa 3.** Definición y uso de una función.

```
9  
[True, False, True, False, True, False, True, False, True]
```

**Salida 3.** Ejemplo de uso de una función definida.

El Python, se puede asignar una palabra clave a cada argumento, para que los usuarios no tengan que recordar el orden o significado de cada argumento. Para ello, en el Programa 3 se modifica la llamada a la función `divisible_by` para indicar el nombre del parámetro:

`divisible_by(mylist=[1,2,3],divisor=2)`. Esa llamada es equivalente a `divisible_by(divisor=2,mylist=[1,2,3])`. También se pueden proponer valores por defecto: si el usuario no proporciona un argumento, la función asigna un valor fijo. Para ello, se cambia la definición de la función con el valor por defecto que se requiera:

```
def divisible_by(mylist,divisor=2):  
...
```

Ahora ejecutar `divisible_by([1,2,3])` es equivalente a `divisible_by([1,2,3], 2)`.



### Ejemplo

Se pueden utilizar parámetros por defecto, por ejemplo, para saber si el usuario ha proporcionado ciertos argumentos o no.

```
def f(a=None, b=None, c=None):  
    if a is not None:  
        print('Se ha proporcionado a')  
    if b is not None:  
        print('Se ha proporcionado b')  
    if c is not None:  
        print('Se ha proporcionado c')  
f(a=1,c="hola")
```

**Programa 4.** Parámetros por defecto en funciones Python.

```
Se ha proporcionado a  
Se ha proporcionado c
```

**Salida 4.** Uso de parámetros por defecto.

En Python es perfectamente posible guardar una referencia a una función en una variable y luego ejecutar la función a través de dicha variable. Del ejemplo anterior, si asignamos la función `f` a la variable `b` (`b = f`), obtenemos el mismo resultado con `f(a=1,c="hola")` que con `b(a=1,c="hola")`.

Si queremos aplicar una serie de funciones a cada objeto en una colección, es muy útil almacenar las funciones en una lista y aplicarlas sucesivamente. El Programa 5 muestra un ejemplo de esto, en el que se limpia una lista de `strings` para que cada elemento no tenga espacios en blanco a ambos lados (`str.strip`) y que empiece por mayúscula (`str.title`).



```
def clean_strings(strings, ops):
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result
strings = ['Valencia ', ' barcelona', ' bilbao ']
operations = [str.strip, str.title]
print(clean_strings(strings, operations))
```

**Programa 5.** Almacenamiento de referencias a funciones en una lista.

```
['Valencia', 'Barcelona', 'Bilbao']
```

**Salida 5.** Las strings originales han sido formateadas a través de las referencias a las funciones almacenadas en una lista.

También es posible definir funciones de forma anónima, sin nombre, a través de **funciones lambda**.

Estas son útiles cuando la definición es sucinta y no se utiliza múltiples veces, por ejemplo, como argumento que se pasa a otra función. Suelen utilizarse para manipular un argumento de alguna forma. Las funciones lambda se escriben de la siguiente forma:

lambda <lista de argumentos> : <valor a retornar>



### Ejemplo

Ordenación de las strings de una lista en función de su longitud:

```
strings = ["C", "Carlos", "Car"]
strings.sort(key=lambda x : len(x))
print(strings)
```

**Programa 6.** Ejemplo de utilización de función lambda.

```
['C', 'Car', 'Carlos']
```

**Salida 6.** Uso de una función lambda para ordenar strings.

## 2.5. Colecciones en Python

Todos los tipos básicos contienen un solo elemento. Es posible combinar varios elementos en un mismo objeto para formar una colección. En Python, hay varias clases de colecciones: tuplas, listas, secuencias, diccionarios y conjuntos.

### 2.5.1. Tuplas

Una **tupla** es una agrupación de elementos inmutable, con los elementos separados por comas y entre paréntesis: `(1,2,6,10)` o `("Antonio", "Zorrilla", 29)`. Una tupla puede contener elementos de distinto tipo, pero, una vez creado, no es posible modificar, añadir o eliminar elementos, ni modificar los existentes. Para acceder a un elemento, se puede utilizar el operador `[]`. Por ejemplo, si `tuple1 = (1, 2, 3, 5)`, entonces `tuple1[1]` retorna el segundo elemento, en este caso, 2.

Las tuplas tienen dos métodos propios: `<tupla>.count(<elemento>)` retorna el número de ocurrencias del `<elemento>` en la tupla; `<tupla>.index(<elemento>)` retorna la posición del `<elemento>` en la tupla (si no existe, retorna un `ValueError`).

La sintaxis de Python permite desempaquetar una tupla asignando varias variables al mismo tiempo. Por ejemplo, `a,b,c = (1,2,3)` resulta en `a = 1`, `b = 2` y `c = 3`. También se puede desempaquetar parte de la tupla y poner el resto en una variable: `a, *otros = ("Carlos", "Gualberto", "Gherardo")` resulta en `a = "Carlos"` y `otros = ["Gualberto", "Gherardo"]`. Por cierto, internamente, este es el mecanismo que utiliza Python cuando una función retorna más de un valor.

### 2.5.2. Listas

Las **listas** son similares a las tuplas en cuanto a que son enumeraciones de elementos de cualquier tipo. A diferencia de las tuplas, las listas sí son mutables, es decir, es posible añadir, eliminar y modificar elementos. Para crear una lista, se enumeran los elementos entre corchetes, separados por comas: por ejemplo, `lista1 = ["No", "Si", 17, 0.121]`. También se utiliza el operador `[]` para acceder a los elementos de una lista y, como es mutable, se puede utilizar para modificar un elemento: `lista1[0] = None` resultaría en `[None, "Si", 17, 0.121]`.

Las listas también tienen los métodos `count()` e `index()` descritos para las tuplas. Hay tres funciones que permiten añadir elementos a las listas:

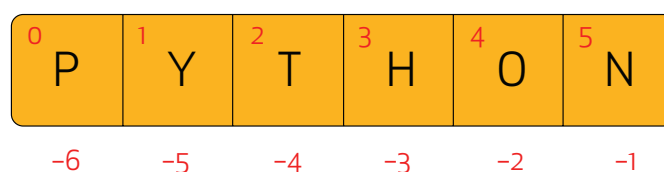
- `<lista>.append(<elemento>)` añade `<elemento>` final de la lista.
- `<lista>.insert(<índice>, <elemento>)` añade `<elemento>` en la posición `<índice>`.
- `<lista>.extend(<secuencia>)` añade los elementos en la `<secuencia>` al final de la lista. Es funcionalmente equivalente a concatenar dos listas (`[1,2,3] + [4,5,6]`) pero evita la creación de una nueva lista.

Para eliminar elementos se puede utilizar `<lista>.remove(<elemento>)`, que elimina la primera ocurrencia de `<elemento>`, o `<lista>.pop([índices])`, que elimina y retorna los elementos en las posiciones indicadas en `[índices]`.

Otra función interesante es `<lista>.sort(reverse=False)`, que reordena la lista comparando los elementos entre sí. Sin embargo, como los elementos de una lista pueden tener tipos distintos, es importante asegurarse de que los elementos son comparables cuando se utiliza `sort`; de lo contrario, Python retornará un `TypeError`.

Una forma rápida de crear listas de secuencias es utilizando la función ya presentada `range()`. Por ejemplo, `list(range(-5,6))` genera una lista del -5 al 5 en incrementos de 1 valor.

Python permite trabajar con elementos en listas de forma muy flexible gracias a lo que se conoce como **slicings** o cortes. Un slicing consiste en obtener una parte de una lista definiendo los límites (primer elemento y último) y el incremento, de forma muy similar a como se define `range()`. Los tres indicadores son opcionales, y si no se especifican el intérprete usa el primer y último como límites y 1 como incremento. La forma general del slicing es `<lista>[<primero>:<último>:<incremento>]`. Se pueden indicar límites negativos, en cuyo caso cuenta desde el final de la lista (el límite -2 indica el tercer elemento desde el final de la lista). La Figura 4 muestra la correspondencia entre índices positivos y negativos con una lista de letras.



**Figura 4.** Correspondencia de índices positivos y negativos en una lista de letras ["P", "Y", "T", "H", "O", "N"].



### Ejemplo

```
lista_completa = list(range(0,6,1)) # lista [0,1,2,3,4,5]
slice1 = lista_completa [:] # lista [0,1,2,3,4,5]
slice2 = lista_completa [1:3] # lista [1,2]
slice3 = lista_completa[:-1] # lista [0,1,2,3,4]
slice4 = lista_completa[::2] # lista [0,2,4]
```

## 2.5.3. Secuencias

Las **secuencias** son objetos iterables no materializados (no son listas, sino objetos). Aunque se pueden utilizar directamente en loops, no retornan una lista en sí (para ello se han de materializar con `list(<secuencia>)`). Ya hemos visto una forma de crear secuencias en Python: `range()`. Otra secuencia muy utilizada para iterar sobre una colección es `enumerate`, que retorna dos objetos iterables: el índice del elemento actual y su valor. Así, podemos iterar cada elemento de una lista de la siguiente forma:

```
for i, value in enumerate([10,20,30])
```




En cada loop, `i` tomará el valor del índice en cuestión (0, 1 y 2), mientras que `value` tomará el valor del elemento (10, 20 y 30).

`reversed(<secuencia>)` crea una secuencia invertida a partir de `<secuencia>` o colección.

Por último, la función `zip` crea una secuencia de tuplas a partir de dos listas, emparejando cada elemento. Toma la forma siguiente:

`list(zip([e1,e2,e3],[o1,o2,o3]))` retorna `[(e1,o1),(e2,o2),(e3,o3)]`

`zip` es muy útil cuando se desea iterar en un loop dos listas al mismo tiempo, como se muestra en el Programa 7.



```
nombres = ["Messi","Iniesta","Modric"]
edades = [31,34,34]
for (nombre,edad) in zip(nombres,edades):
    print("Nombre: " + nombre + ", edad: " + str(edad))
```

**Programa 7.** Uso de `zip` para iterar sobre dos listas simultáneamente.

```
Nombre: Messi, edad 31
Nombre: Iniesta, edad 34
Nombre: Modric, edad 34
```

**Salida 7.** En cada iteración, se accede a un elemento de cada lista.

Python permite hacer unzip mediante la sintaxis `*` (que internamente crea, como hemos visto, una tupla con la información restante):

```
lista1 = list(range(5))
lista2 = list(reversed(range(5)))
zipped = zip(lista1,lista2)
original1, original2 = zip(*zipped)
lista1 == original1 # True
lista2 == original2 # True
```

## 2.5.4. Diccionarios

Los **diccionarios** son colecciones de elementos que incluyen parejas de claves (referencias únicas) y valores. Cualquier objeto inmutable puede ser clave; cualquier objeto puede ser utilizado como valor, tanto elementos básicos como diccionarios u otras colecciones (diccionarios anidados). La forma de crear un diccionario básico es:

```
dict1 = {<clave1>:<valor1>, <clave2>:<valor2>,...}
```

Con el operador `[]` se accede a los valores del diccionario, pasando el índice deseado (clave). Por ejemplo, si `dict1 = {"Nombre": "Carlos", "Apellido": "Fernandez"}`, entonces `dict1["Nombre"]` retorna "Carlos". También se puede acceder para modificar valores: `dict1["Nombre"] = "Leo"` cambia el nombre de la primera entrada.

Si al acceder o modificar un diccionario la clave no está presente, Python la genera implícitamente: `dict1["Edad"] = 34` añade la pareja "Edad": 34 al diccionario `dict1`. Para comprobar si una clave está presente en el diccionario, se puede usar el operador `in`. Por ejemplo, `"Nombre" in dict1` retorna `True`, pero `"Profesion" in dict1` retorna `False`.

Se pueden eliminar claves de un diccionario de dos formas: `del dict1[key]` elimina la clave `key` del diccionario; `dict1.pop(key)` hace lo mismo.

Para iterar un diccionario, se puede hacer enumerando sus claves (`for key in dict1.keys()`) o directamente con `for key in dict1`, enumerando sus valores (`for value in dict1.values()`) o incluso ambos.

```
for key,value in zip(dict1.keys(),dict1.values())
```

Aquí hemos creado dos secuencias separadas, con las claves y los valores, y las hemos unido con `zip`. Luego, en cada iteración, tenemos la clave (`key`) y el valor (`value`) por separado. Nota: también se pueden iterar claves y valores utilizando `<diccionario>.items()`.

Además de poder crear diccionarios explícitamente con el par de caracteres `{}`, Python permite transformar secuencias en diccionarios fácilmente, utilizando `zip`:

```
nombres = ["Messi","Iniesta","Modric"]
edades = [31,34,34]
diccionario = dict( zip(nombres,valores) ) # { "Messi": 31, "Iniesta": 34,
"Modric": 34 }
```

### 2.5.5. Sets (conjuntos)

La última colección que cubriremos en este curso son los **sets** o conjuntos, que son una forma específica de diccionarios sin valores, solo claves. De esta forma, cada elemento en un set está representado un máximo de una vez (como el concepto de conjunto en matemáticas). Cualquier objeto inmutable puede ser un elemento en un set (tipos básicos o tuplas, por ejemplo).

Para crear un set, se utiliza la función `set(<[elementos]>)`, pasando un listado de elementos. Por ejemplo, `set([1,1,2,3,5])` retorna el set `{1,2,3,5}`.

Como con los conjuntos matemáticos, hay varias operaciones básicas que se pueden hacer entre sets:

- Unión (signo `|`): retorna todos los elementos de ambos sets.
- Intersección (signo `&`): retorna solo los elementos presentes en ambos sets
- Diferencia (signo `-`): retorna los elementos del primer set que no se encuentran en el segundo.
- Diferencia simétrica (signo `^`): retorna solo los elementos contenidos en un solo set y no en el otro.

También hay ciertas funciones que facilitan la comparación entre sets:

- `<set1>.issubset(<set2>)` retorna True si `<set1>` es un subconjunto de `<set2>`.
- `<set1>.issuperset(<set2>)` retorna True si `<set1>` es un superconjunto de `<set2>`.
- `<set1>.isdisjoint(<set2>)` retorna True si `<set1>` y `<set2>` son sets disjuntos (es decir, si su intersección es nula).

## 2.5.6. Python comprehension

Uno de los casos más citados como ejemplo del poder de la sintaxis básica de Python son las **list comprehensions** (aunque también se pueden usar en diccionarios). Se utilizan para construir listas, diccionarios o conjuntos a través de expresiones más sucintas que el equivalente `for` loop. La forma general de list comprehension es la siguiente:

```
[<expresión1> if <condition> else <expresión2> for value in <collection>]
```

Esta expresión genera una lista en la que cada elemento está formado evaluando la `<expresión1>` (que normalmente incluye algún tipo de transformación de `value`). Como vemos, puede también incluirse un condicional para usar `<expresión1>` o `<expresión2>` en cada iteración.

*Dictionary / set comprehension* es similar, pero en vez de usar el par de caracteres `[ ]` se utiliza el par `{ }` para generar la colección.



### Ejemplo

```
# Usando for loop básico
numbers = list(range(-10,10))
values = []
for x in numbers:
    if(x > 0):
        values .append(x*2)
    else:
        values .append(x/2)
# Usando list comprehension
values = [x*2 if x>0 else x/2 for x in range(-10,10)]

# usando dict comprehension para intercambiar claves y valores

compra= {"Bananas":1.5, "Peras":2.5, "Uvas":1.2}

rev_dict = {value:key for key,value in dict1.items()} # {1.5:"Bananas",
2.5:"Peras", 1.2:"Uvas"}
```

Aunque en ocasiones dificulta la legibilidad, es posible anidar comprehensions concatenándolas. La expresión `[elemento for item in compra.items() for elemento,precio in item]` es equivalente al `for` loop anidado siguiente:

```
precios = []
for item in compra.items():
    for elemento,precio in item:
        precios.append(precio)
```

01

```

.....
mirror_mod.use_x = False
mirror_mod.use_y = True
mirror_mod.use_z = False
operation == "MIRROR_Z"
mirror_mod.use_x = False
mirror_mod.use_y = False
mirror_mod.use_z = True

[ (1+x+y+2a)-(3a+3g+x)
  5+x+k+2a+21
  E=mc2
  1lim h-->0
  2+....+2a....+a
  selection at the end -add
  _ob.select= 1
  ler_ob.select=1
  text.scene.objects.acti
  ("Selected" + str(modifier
  error_ob.select = 0
  bpy.context.selected_ob
  sta.objects[one.name].se
  Int("please select exactl
  1+x+y+2a
  2+...+2a+3a
  1+x+y+2a

```

## Capítulo 3

### Colecciones: NumPy

**NumPy** es una librería externa para Python. Su nombre deriva de “**N**umerical **P**ython” y facilita la computación numérica en Python. Para utilizarla, se ha de importar el módulo `numpy` con la expresión `import numpy`. La siguiente expresión importa el módulo `numpy` y lo hace accesible a través del alias `np`:

```
import numpy as np
```

**NumPy** está escrito en lenguaje C, por lo que es muy eficiente y rápido de usar. Es el estándar de facto en cuanto a computación numérica, y muchas otras librerías lo tratan como tal.

**NumPy** se utiliza frecuentemente para crear colecciones multidimensionales y aplicar funciones matemáticas estándares a los elementos en una colección.

### 3.1. Objeto básico en NumPy: ndarray

En el corazón de **NumPy** está el objeto `ndarray`. Una `ndarray` es una colección multidimensional de datos del mismo tipo, aunque se pueden usar `ndarrays` para colecciones de `bool` o `string`; son útiles y muy eficientes cuando se usan para grabar y manipular valores numéricos (`float` e `int`).



Cada `ndarray` solo puede contener elementos de un solo tipo (`int`, `float`, `string`, `bool`, etc.). Si hay que crear una `ndarray` que contenga tipos diferentes, Python trata de convertir los tipos a uno solo. Por ejemplo:

- `np.array([1, "2"])` se convierte en `array(["1", "2"])`
- `np.array([True, 0])` se convierte en `array([1, 0])`

Para crear `ndarrays`, **NumPy** ofrece varias formas:

- `np.array(<secuencia>)` crea una `ndarray` a partir de una lista o secuencia de Python. Si la lista o secuencia es anidada, se creará una dimensión por nivel. Por ejemplo, `a = np.array([1, 2], [3, 4, 5])` crea la `ndarray` bidimensional `array([1, 2], [3, 4, 5])`
- `np.zeros(<dimensiones>)`, `np.ones(<dimensiones>)` y `np.empty(<dimensiones>)` generan una `ndarray` con valores 0, 1 o sin determinar, respectivamente. El valor `dimensiones` se pasa como una tupla. Por ejemplo, `np.zeros((2, 2))` genera `array([0, 0], [0, 0])`. Con `np.empty`, los elementos asignados inicialmente son indeterminados, es decir, pueden tomar cualquier valor.
- `np.arange(min, max, step)` crea una `ndarray` de forma similar a la secuencia `range`.
- `np.full(<tamaño>, <valor>)` crea una `ndarray` de tamaño definido por la tupla `<tamaño>` con todos los valores inicializados a `<valor>`.

Algunos de los métodos más interesantes en cualquier objeto `ndarray` son los siguientes:

- `sort()` ordena los elementos de la array.
- `argmax()` retorna el índice del elemento más alto.
- `argmin()` retorna el índice del elemento más bajo.



#### Enlace de interés

Existen muchos métodos aplicables a cada `ndarray`. Aunque aquí solo cubrimos unos pocos, se puede consultar el resto en el siguiente enlace:

<https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html>

### 3.1.1. Dimensiones

Cada `ndarray` contiene metainformación sobre sus dimensiones y el tipo de sus elementos. Así, `array1.ndim` retorna el número de dimensiones, `array1.shape` retorna una tupla con las dimensiones y `array1.dtype` retorna el tipo de sus elementos.

**NumPy** permite forzar una `ndarray` a unas dimensiones específicas. A través del método `reshape(x,y)` de cada `ndarray`, `-x` determina el número de elementos en la primera dimensión, `y` determina el número de elementos en la segunda. Por ejemplo, la `arr1 = np.array([1,2,3,4])` puede cambiar a una 2D con `arr1.reshape(2,2)`, resultando en `array([[1,2], [3,4]])`.

Un ejemplo en el que `reshape` es muy conveniente es cuando se desea generar una matriz con valores consecutivos. `np.arange(9).reshape(3,3)` retorna la matriz:

```
[ [0, 1, 2],
  [3, 4, 5],
  [6, 7, 8] ]
```

Como opuesto a `reshape`, `flatten()` se utiliza para allanar una `ndarray` multidimensional a una de una sola dimensión.

### 3.1.2. Manipulación de ndarrays

**NumPy** permite juntar dos arrays, bien de lado a lado o bien una encima de la otra, con las funciones `np.hstack(a1,a2)` o `np.vstack(a1,a2)` respectivamente (del inglés, *horizontal stack* y *vertical stack*).

También es posible dividir una array en múltiples. `np.array_split(array1, divisiones)` divide `array1` en tantas partes como se define en `divisiones`. Se puede pasar el argumento `axis` para determinar si se debe dividir por filas (`axis=0`) o columnas (`axis=1`). `divisiones` puede ser una tupla o lista (de tamaño máximo igual a las dimensiones de `array1`), en cuyo caso se puede especificar dónde cortar en cada dimensión.

```
a = np.arange(16).reshape(4,4)
dividida = np.array_split(a, (3,3) )
for i,d in enumerate(dividida):
    print(i)
    print(d)
```

**Programa 8.** Manipulación de las dimensiones de una `ndarray`.

```
0
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
1
[]
2
[[12 13 14 15]]
```

**Salida 8.** `ndarray` dividida con el método `split`.

### 3.1.3. Índices y slicing

Tanto los índices como slicing en `ndarrays` funcionan de manera homóloga a como lo hacen en listas Python, con la diferencia de que, cuando se solicita una slice de una `ndarray`, se retorna una referencia a la parte afectada de la `ndarray`, y no una copia. Esto significa que, si se modifica algún elemento de la slice, el cambio se verá reflejado en la `ndarray` original.

**NumPy** permite mucha más flexibilidad para realizar y trabajar con slices. Por ejemplo, se puede asignar un único valor a los elementos de la slice con `array1[:3] = 0` (asigna 0 a los tres primeros elementos). También se pueden hacer slices condicionales, es decir, retornar solo los elementos que cumplen una condición; para ello, se especifica una condición (o condiciones encadenadas con operadores lógicos `&`, `|` o `~`) entre corchetes.



#### Ejemplo

```
array1 = np.arange(-3,4) # [-3,-2,-1,0,1,2,3]
array1[array1 < 0] # [-3,-2,-1]
array1[array1 % 2 == 0] = 10 # [-3, 10, -1, 10, 1, 10, 3]
array1[array1 == 10] = 0 # [-3, 0, -1, 0, 1, 0, 3]
```

Para más flexibilidad, se puede utilizar una lista como índice a una `ndarray`, en cuyo caso se retornan los elementos con los índices especificados en la lista (si la lista es 2D, el primer elemento determina la fila y el segundo la columna). Por ejemplo, `array[ [1,2] ]` retorna el segundo y tercer elementos.

Cuando se trabaja con `ndarrays` multidimensionales, se puede acceder a elementos a través de índices separados por comas. En la `ndarray` bidimensional `array1`, `array1[0][1] == array[0,1]`.

## 3.2. Funciones matemáticas sobre colecciones

Uno de los aspectos que más valor tiene de **NumPy** es la facilidad de aplicar operaciones o funciones matemáticas a un conjunto de valores de una vez, sin necesidad de iterar sobre cada uno de los valores con un loop. Así, se puede aplicar cualquier operador binario (+, -, /, \*, %, etc.) a toda una `ndarray` de la misma forma que con cualquier variable:

```
array = np.arange(5) # array(0,1,2,3,4)
array = array * 10 # array(0,10,20,30,40)
```

También se pueden utilizar operadores entre dos `ndarrays`, lo que resulta en una `ndarray` producto de la operación entre cada uno de los elementos:

```
array1 = np.array([1,1,1,1])
array2 = np.array([5,5,5,5])
result = array1 + array2 # array(6,6,6,6)
```





### Enlace de interés

Si se aplican operadores binarios a `ndarrays` con dimensiones distintas, Python aplica la técnica de **broadcasting**, que, en su forma más simple, resulta en virtualmente incrementar el tamaño de la `ndarray` más pequeña hasta igualar el de la otra, para así poder aplicar la operación correctamente. Esto es lo que sucede en el caso básico de aplicar una operación entre un escalar y una `ndarray` (por ejemplo, `np.arange(10) * 2`).

El valor de la porción incrementada es normalmente una copia del elemento ya existente en la `ndarray`, aunque en situaciones más complejas se determina siguiendo una serie de reglas. Para más información acerca de las reglas de broadcasting, se puede visitar el siguiente enlace:

<https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html>

Además de poder aplicar operaciones Python a `ndarrays`, **NumPy** ofrece una serie de **operadores unarios** (que se aplican a un solo operando) y **operadores binarios**. La Tabla 2 lista algunos de los más útiles. Nota: para utilizar los operadores, se debe hacer a través del alias `np` (o el que se haya decidido al importar **NumPy**). Por ejemplo, `np.abs(np.array([1, -2, 3]))` retorna `array(1, 2, 3)`.

**Tabla 2**

Listado de los operadores binarios y unarios más comunes dentro de **NumPy**

Tipo	Operación	Descripción
Unario	<code>abs</code>	Valor absoluto de cada elemento
	<code>sqrt</code>	Raíz cuadrada de cada elemento
	<code>exp</code>	$e^x$ , siendo $x$ cada elemento
	<code>log</code> , <code>log10</code> , <code>log2</code>	Logaritmos en distintas bases de cada elemento
	<code>sign</code>	Retorna el signo de cada elemento (-1 para negativo, 0 o 1 para positivo)
	<code>ceil</code>	Redondea cada elemento por arriba
	<code>floor</code>	Redondea cada elemento por abajo
	<code>isnan</code>	Retorna si cada elemento es <code>NaN</code>
	<code>cos</code> , <code>sin</code> , <code>tan</code>	Operaciones trigonométricas en cada elemento
	<code>arccos</code> , <code>arcsin</code> , <code>arctan</code>	Inversas de operaciones trigonométricas en cada elemento
Binario	<code>add</code>	Suma de dos arrays
	<code>subtract</code>	Resta de dos arrays
	<code>multiply</code>	Multipliación de dos arrays
	<code>divide</code>	División de dos arrays
	<code>maximum</code> , <code>minimum</code>	Retorna el valor máximo / mínimo de cada pareja de elementos
	<code>equal</code> , <code>not_equal</code>	Retorna la comparación (igual o no igual) de cada pareja de elementos
	<code>greater</code> , <code>greater_equal</code> , <code>less</code> , <code>less_equal</code>	Retorna la comparación (>, >=, <, <= respectivamente) de cada pareja de elementos

### 3.3. Filtrado de datos

Una función muy útil en el análisis de datos, por la versatilidad que da para filtrar y modificar información, es `np.where`. En su forma más básica, retorna una `ndarray` resultado de aplicar una condición elemento a elemento sobre una `ndarray`, escogiendo un valor `a` o el valor `b` en función del resultado de la comparación. Toma la siguiente forma:

```
np.where(<condición>, <elemento_condición_afirmativa>, <elemento_condición_negativa>)
```

Por ejemplo, se puede utilizar `np.where` para crear una lista en la que los elementos positivos aparecen con un 1 y los demás con 0 de la siguiente forma:

```
resultado = np.where(array1 < 0, 0, 1)
```

`np.where` permite también utilizar `ndarrays` como elementos sustitutivos, de forma que, en vez de sustituir los valores con condicional afirmativo por un único valor, se escoge un elemento de una `ndarray`. Por ejemplo, siguiendo el ejemplo anterior, se pueden sustituir todos los elementos negativos por 0, pero dejar los positivos como están:

```
resultado = np.where(array1 < 0, 0, array1)
```

Si se proporciona solo el valor condicional, **NumPy** retorna los índices para los que la condición es afirmativa.

Por ejemplo, para conocer la posición de los elementos negativos de `array = np.array([-1, 1, -2, 2])`, se puede hacer a través de `np.where(array < 0)`, que retorna `array([0, 2])`.

También se puede emplear `np.where` para limpieza básica de datos, por ejemplo, para eliminar ocurrencias de NaN en los datos. Si tenemos, `array = np.array([1, 2, np.nan])`, aplicando `np.where(np.isnan(array), 0, array)` obtenemos `array([1, 2, 0])`.

### 3.4. Exploración y estadística descriptiva

Una parte fundamental del analista de datos es entender las características de la información que maneja. Así, la fase preliminar de exploración ayuda al profesional a comprender la naturaleza de los datos que se tienen entre manos.

En este sentido, la estadística descriptiva ofrece una visión general de los datos en conjunto, como el valor medio, cómo de diferentes son los valores, mínimos y máximos, etc.

En la Tabla 3 vemos algunas de las funciones de estadística descriptiva dentro de **NumPy**. Todas son accesibles a través del alias **NumPy**, pasando como argumento la `ndarray` que se desea explorar. Por ejemplo, `np.mean(arr1)` retorna el valor medio de la array `arr1`.

**Tabla 3***Funciones en NumPy de estadística descriptiva*

Función	Descripción
<code>sum(arr1)</code>	Suma de todos los elementos de <code>arr1</code>
<code>mean(arr1)</code>	Media aritmética de los elementos en <code>arr1</code>
<code>std(arr1)</code>	Desviación estándar de los elementos en <code>arr1</code>
<code>cumsum(arr1)</code>	Retorna array con la suma acumulada de cada elemento con todos los anteriores
<code>cumprod(arr1)</code>	Retorna array con el producto acumulado de cada elemento con todos los anteriores
<code>min(arr1), max(arr1)</code>	Mínimo y máximo de <code>arr1</code>
<code>any(arr1)</code>	En array de tipo <code>bool</code> , retorna <code>True</code> si algún elemento es <code>True</code>
<code>all(arr1)</code>	En array de tipo <code>bool</code> , retorna <code>True</code> si todos los elementos son <code>True</code> (o $>0$ en valores numéricos)
<code>unique(arr1)</code>	Retorna una array con valores únicos (similar a <code>set</code> )
<code>in1d(arr1, arr2)</code>	Retorna array con <code>bool</code> indicando si cada elemento de <code>arr1</code> está en <code>arr2</code>
<code>union1d(arr1, arr2)</code>	Retorna la unión de ambas arrays
<code>intersect1d(arr1, arr2)</code>	Retorna la intersección de ambas arrays

Si se trabaja con `ndarrays` multidimensionales, la mayoría de funciones permiten pasar un argumento para indicar qué eje de la `ndarray` considerar. De esta forma, se puede obtener la suma de todos los elementos, columna por columna, de una array de la siguiente forma:

```
array1 = np.array([ [1,2,3], [10,20,30] ])
np.sum(array1,axis=0) # array([11,22,33])
```

En este caso, `axis` acepta el eje (o dimensión): 0 para columnas, 1 para filas, y así sucesivamente si hay más dimensiones.

### 3.5. Álgebra lineal

**NumPy**, como buena librería de arrays, dispone de funciones y métodos para facilitar el trabajo en álgebra lineal, desde operaciones con matrices, hasta resolución de sistemas de ecuaciones.

Para multiplicar dos matrices expresadas como `ndarrays`, se puede utilizar `np.dot(matriz1, matriz2)`, o bien directamente `matriz1 @ matriz2` (en Python 3.5+). También se puede obtener el mismo resultado con el método propio de una array, `matriz1.dot(matriz2)`. Nota: `matriz1 * matriz2` resulta en una multiplicación elemento a elemento y no en el producto de dos matrices. Para que la multiplicación tenga sentido, las dimensiones de las matrices deben ser compatibles (si el tamaño de `matriz1` es  $A \times B$ , el tamaño de `matriz2` debe ser  $B \times C$ , es decir, el número de columnas de la primera matriz debe coincidir con el número de filas de la segunda).

Otras operaciones convenientes para `ndarrays` que representan matrices son:

- Calcular la traspuesta con `matriz1.T` o `matriz1.transpose()`.
- Obtener los elementos en la diagonal con `matriz1.diagonal()`.

Construir una matriz con elementos específicos en su diagonal (pero 0 en el resto): `np.diag(array)`, con `array` siendo unidimensional.

**NumPy** incluye un submódulo llamado `numpy.linalg` dedicado al álgebra lineal, con funcionalidad dedicada a la factorización de matrices, cálculo de determinantes e inversas, etc. La Tabla 4 describe algunas de las funciones más importantes.

**Tabla 4**

*Lista de las funciones más importantes en `numpy.linalg`*

Función	Descripción
<code>dot(mat1,mat2)</code>	Retorna el producto escalar entre dos arrays. Si son matrices 2D, es equivalente a la multiplicación de ambas
<code>matmul(mat1,mat2)</code>	Retorna el producto entre dos matrices
<code>trace(mat1,mat2)</code>	Suma de las diagonales de ambas matrices
<code>det(mat1)</code>	Retorna el determinante de la matriz <code>mat1</code>
<code>eig(mat1)</code>	Computa los autovalores y autovectores de la matriz cuadrada <code>mat1</code>
<code>inv(mat1)</code>	Retorna la inversa de la matriz <code>mat1</code>
<code>qr(mat1)</code>	Computa la factorización QR de <code>mat1</code>
<code>solve(A,b)</code>	Resuelve el sistema lineal de ecuaciones $Ax = b$ para <code>x</code> , cuando <code>A</code> es una matriz cuadrada



#### Enlace de interés

Para más información sobre las funciones dentro de `numpy.linalg`, se puede acceder a la documentación oficial en el siguiente enlace:

<https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.linalg.html>

## 3.6. Valores aleatorios

Python contiene un módulo denominado `random` que se puede utilizar para generar números aleatorios. Una vez importado con `import random`, se pueden generar números aleatorios con `random.random()` para fracciones de 0 a 1, `random.randint(min,max)` para números enteros dentro de un rango o `random.uniform(low,high)` para fracciones dentro de un rango. También ofrece la posibilidad de elegir un valor de entre una colección al azar con `random.choice(<colección>)`, donde `colección` puede ser también una `string`.

Aunque el módulo `random` puede ser útil en casos simples, tiende a ser relativamente lento. **NumPy** ofrece una serie de funciones dentro del **namespace** `np.random` con mejor rendimiento. Entre otras se encuentran las siguientes:

- `np.random.rand()` retorna un número aleatorio entre 0 y 1
- `np.random.randint(min, max)` retorna un número entero entre `min` (inclusive) y `max` (exclusive)
- `np.random.randn(d0, d1...)` retorna una `ndarray` con elementos aleatorios de una distribución normal (media 0 y varianza 1) de dimensiones especificadas por `d0`, `d1...`

Para obtener números a partir de distintas distribuciones estadísticas, **NumPy** ofrece los siguientes métodos:

- `np.random.binomial(n, p)` genera números de una distribución binomial.
- `np.random.uniform(low, high)` genera números a partir de una distribución uniforme (con igual probabilidad para todos).
- `np.random.poisson(lambda, size)` genera números de una distribución poisson.



#### Enlace de interés

Para más información sobre números aleatorios en **NumPy** y otras distribuciones, se puede visitar el siguiente enlace:

<https://docs.scipy.org/doc/numpy/reference/routines.random.html>



Por supuesto, como máquinas deterministas que son, los ordenadores no pueden generar números realmente aleatorios. Los algoritmos utilizan ecuaciones para generar números pseudoaleatorios, pero siempre a partir de un valor dado, que se denomina semilla. Esto quiere decir que una secuencia de números pseudoaleatorios a partir de una semilla es siempre la misma, aunque para el observador tenga un patrón aleatorio. Si se escoge otra semilla, la secuencia será distinta. Esto sirve muchas veces en simulaciones científicas para asegurarse de que las condiciones de un experimento son idénticas.

Para iniciar una secuencia pseudoaleatoria con una semilla determinada, en **NumPy** se utiliza el método `np.random.seed(semilla)`.



Para utilizar **pandas**, la forma más habitual es hacerlo a través del alias **pd**:

```
import pandas as pd
```

Si se desea ser más específico, se puede importar exclusivamente una parte de **pandas**, por ejemplo:

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

Importar directamente partes de una librería evita tener que repetir el alias múltiples veces. Así, se puede utilizar **Series**, en vez de **pd.Series**. **Series** y **DataFrame** son las dos estructuras de datos básicas en **pandas**. Las veremos con detalle en las próximas secciones.

## 4.1. Series

Las **Series** son estructuras unidimensionales similares a las **ndarray** de **NumPy**, en las que cada elemento posee también un índice único. La forma más sencilla de crear una serie es con el constructor **pd.Series(<lista>, index=<lista de índices>)**. El argumento **index** es opcional y, si no se pasa, **pandas** asume índices de 0 a **n-1** (donde **n** es el tamaño de la <lista> de elementos).

```
serie = pd.Series(['Barcelona', 'Madrid', 'Valencia', 'Sevilla'],  
index=['a', 'b', 'c', 'd'])  
serie.name = 'Al menos dos equipos en primera'  
serie.index.name = 'Id'  
print(serie)
```



**Programa 9.** Ejemplo de utilización del objeto **Series** de **pandas**.

```
Id  
a    Barcelona  
b     Madrid  
c    Valencia  
d     Sevilla  
Name: Al menos dos equipos en primera, dtype: object
```

**Salida 9.** Imprimir un objeto **Series** muestra el nombre del índice, el nombre de la serie, el tipo de objeto de cada elemento y los elementos incluidos (junto a sus índices).

Como se puede ver en el ejemplo anterior, tanto la serie (**serie.name**) como la columna de índices (**serie.index.name**) pueden nombrarse para facilitar su introspección.

El acceso a los elementos de una serie se puede hacer a través del índice explícito o en listas, por orden. En el ejemplo, **serie['d'] == serie[3]**. En general se puede trabajar con **Series** de la misma forma que con **ndarrays**:

- Listado de índices para acceso a múltiples elementos: tanto a través del índice explícito (`serie[ ['a','c'] ]`) como el ordinal (`serie[ [0,2] ]`).
- Índice condicional para filtrar elementos: `serie[serie > 10]` retorna una serie solo con los elementos mayores de 10.
- Slicing: `serie[: -1]` retorna una serie con todos los elementos menos el último.
- Funciones aplicadas a todos los elementos: `np.sqrt(serie)` retorna una serie en la que cada elemento es la raíz cuadrada del original.



Como cualquier otra colección, se puede iterar sobre los valores de una serie con un simple for loop.

```
for index, value in enumerate(serie):
```

En cada iteración, `index` toma el valor del índice del elemento, y `value` es el elemento en sí. Si solo interesa iterar sobre los valores, se puede hacer directamente sin `enumerate` en la forma simplificada: `for value in serie`.

Como las **Series** son estructuras con índice explícito, se pueden entender como simples diccionarios Python en que la clave es el índice y el valor es el elemento. Como tales, se pueden aplicar operaciones de diccionarios:

Averiguar si un índice está presente: `'b' in serie`

Convertir un diccionario en una serie: `serie = pd.Series( { 'Carlos' : 100, 'Marcos': 98} )`. Dado que un set es un diccionario de claves sin valores, también se puede convertir un set en una serie de la misma forma.

**pandas** hace posible la unión de dos **Series** con el operador `+`. Cuando se unen dos **Series**, el resultado es una serie cuyos elementos se determinan en función de los índices:

- Los elementos cuyo índice está en ambas series se suman.
- Los elementos cuyo índice solo aparece en una serie se añaden con el valor `NaN`.



### Ejemplo

```
serie1 = pd.Series([10,20,30,40],index=range(4) )  
serie2 = pd.Series([1,2,3],index=range(3) )  
serieComb = (serie1 + serie2)
```

>>>



```
>>> # serieComb contiene lo siguiente

0          11.0
1          22.0
2          33.0
3           NaN
dtype: float64
```

El operador `-` funciona de forma similar, pero restando los valores de los elementos cuyo índice aparece en ambas series.

Con ambos operadores, es posible acabar con valores perdidos (*missing values*) si hay elementos únicos en cualquiera de las series. Para estos casos, y para el caso general de filtrar los datos, **pandas** ofrece funciones para detectar dichos valores. `pd.isnull(<serie>)` retorna una serie cuyos valores corresponden a `True` si el valor de la `<serie>` es nulo, y `False` si no lo es. Un caso práctico es sustituir los valores perdidos de una serie con un valor fijo. Para ello, se utiliza el resultado de `pd.isnull()` para filtrar elementos de una serie y asignar un valor, como se muestra en el Programa 10.

```
serie1 = pd.Series([1,2,3,np.NaN])
serie1[ pd.isnull(serie1) ] = 0
print(serie1)
```

**Programa 10.** Utilización de `pd.isnull` para filtrar y reemplazar valores nulos en una serie.

```
0      1.0
1      2.0
2      3.0
3      0.0
dtype: float64
```

**Salida 10.**

La función `pd.notnull(<serie>)` es equivalente pero contraria (retorna `True` cuando el elemento **no** es nulo).

## 4.2. DataFrame

Para trabajar con datos tabulares (filas y columnas), **pandas** incluye la versátil estructura `DataFrame`. Un `DataFrame` o frame se puede entender como una colección de `Series` (columnas), todas compartiendo un listado de índices únicos. La forma más común de crear un frame es con un diccionario en el que cada clave se asocia a un listado de elementos de igual longitud.

```
diccionario = { "nombre" : ["Marisa","Laura","Manuel"], "edad" : [34,29,11] }
frame = pd.DataFrame(diccionario)
```

	nombre	edad
0	Marisa	34
1	Laura	29
2	Manuel	11

Como podemos observar, la clave se utiliza como nombre de cada columna (serie), y cada elemento se asocia a una fila en función del índice. Cada columna puede tener un tipo de elemento (en este caso, 'nombre' tiene tipo `string` y 'edad' el tipo `int`).

Como ocurre con Series, en las que se puede especificar el listado a utilizar como índice, en DataFrames se puede pasar un listado de nombres para columnas con el argumento `columns` en el constructor. Esto sirve también para determinar el orden de las columnas en el frame.

```
pd.DataFrame(<diccionario>, columns=<listado de columnas>)
```

Implícitamente, **pandas** automáticamente detecta si una clave en el diccionario aparece en el listado de columnas; cuando el nombre de la columna no aparece como clave, **pandas** crea una nueva columna en el frame con valores `NaN` para todas las filas.



### Ejemplo

```
diccionario = {"nombre" : ["Marisa","Laura","Manuel"], "edad" : [34,29,11] }
frame = pd.DataFrame(diccionario, columns = ['Nacionalidad', 'nombre',
'edad', 'profesion'] )
```

# frame contiene lo siguiente

	Nacionalidad	nombre	edad	profesion
0	NaN	Marisa	34	NaN
1	NaN	Laura	29	NaN
2	NaN	Manuel	11	NaN


Una vez creado el frame, se puede acceder a cada una de sus columnas por separado utilizando como índice el nombre de la columna, bien con corchetes (`<frame>['columna']`) o directamente (`<frame>.columna`). En el ejemplo anterior, se puede acceder a la columna edad con `frame['edad']` o `frame.edad`.

Con frecuencia, los DataFrames pueden tener miles o millones de entradas, por lo que a veces es conveniente, para explorar los datos, observar solo los primeros casos, con `<frame>.head()`.

Añadir datos a un frame es sencillo y análogo a como se hace en diccionarios o ndarrays.

Para cambiar los elementos de una columna, se asigna directamente con `<frame>['columna'] = <valor>`. Si la columna no existe, se creará una nueva. Si ya existe, los elementos se sustituyen con `<valor>` (si la columna ya existe, también se puede asignar directamente con `<frame>.columna`). Nótese que el valor puede ser un elemento simple (numérico, texto, etc.), pero también puede ser una *Series*, de modo que se puede inicializar el valor de cada fila en la tabla.

Para eliminar columnas, se puede utilizar el comando `del <frame>['columna']` o utilizar el método `<objeto>.drop(<índices>)`, válido tanto para *Series* como para *DataFrame*.



```
d_and_d_characters = {'Name' : ['bundenth','theorin','barlok'],
'Strength' : [10,12,19], 'Wisdom' : [20,13,6]}
character_data = pd.DataFrame(d_and_d_characters)
print('Original data')
print(character_data)
vitality_data = pd.Series([11,10,14])
character_data['vitality'] = vitality_data
character_data['alive'] = True
print('Modified data')
print(character_data)
del character_data['alive']
print('Final data')
print(character_data)
```

**Programa 11.** Ejemplo de creación y manipulación de un *DataFrame*.

Original data

	Name	Strength	Wisdom
0	bundenth	10	20
1	theorin	12	13
2	barlok	19	6

Modified data

	Name	Strength	Wisdom	vitality	alive
0	bundenth	10	20	11	True
1	theorin	12	13	10	True
2	barlok	19	6	14	True

Final data

	Name	Strength	Wisdom	vitality
0	bundenth	10	20	11
1	theorin	12	13	10
2	barlok	19	6	14

**Salida 11.** Creación y manipulación de un *DataFrame*.

`<frame>.values` retorna los datos del frame en formato lista; cada elemento es una fila de la tabla original, cada elemento de la cual toma el valor de la fila en la columna correspondiente. En el ejemplo de los personajes de *Dungeons and Dragons*, `character_data.values[0]` retorna `['bundenth',10,20,11]`. Acceder a los `values` puede ser útil para iterar sobre todos los elementos de una fila.

Como si de una matriz se tratara, se puede acceder al `DataFrame` inverso (cambiar filas por columnas y viceversa) con `<frame>.T`.

## 4.3. Trabajo con Series y DataFrame

En esta sección vamos a ver funcionalidades en **pandas** que se pueden aplicar tanto a objetos `Series` como `DataFrame`, como búsqueda de elementos, eliminación de datos, indexación, slicing y reindexación.

### 4.3.1. Búsqueda

El método `<objeto>.isin(<lista>)` retorna una serie o frame (dependiendo del tipo de `<objeto>`) máscara, de las mismas dimensiones que `<objeto>` pero con el valor de cada elemento siendo `True` si el valor original se encuentra dentro de `<lista>` y `False` si no está presente. De forma más específica, se puede obtener una máscara con los elementos que son nulos (`<objeto>.isnull()`) o `NaN` (`<objeto>.isna()`).

También se pueden formular queries para determinar si una fila (búsqueda por índice) o columna (búsqueda por columna) existe, del mismo modo que se hace con los diccionarios o las listas. Por ejemplo, `"Carlos" in obj.index` retorna `True` si hay una entrada con índice `Carlos` en `obj`. Para buscar columnas, se puede hacer a través del listado `<frame>.columns`, por ejemplo, `"age" in frame.columns`.

### 4.3.2. Indexación y slicing

Como mencionamos en la sección de `Series`, tanto indexación como slicing en **pandas** funcionan de forma análoga a como lo hacen en **NumPy**. En el caso de `Series`, es posible indexar por posición (como en **NumPy** o colecciones Python) o por referencia (utilizando el **índice semántico** de la serie). Por ejemplo, en la serie `serie = pd.Series([3,43,12],index=['first','second','third'])`, se puede acceder al primer valor con `serie[0]` y con `serie['first']`.

En el caso de `DataFrame`, solo es posible indexar por referencia (por nombre de la columna), retornando una serie con los valores de la columna especificada.

Tanto en `DataFrame` como en `Series`, es posible que haya lugar a ambigüedades cuando se indexa con números enteros; `objeto[0]` puede ser interpretado como índice posicional (acceder al primer valor) o índice semántico (acceder al valor con índice 0). Por ello, **pandas** siempre interpreta un número entero como índice posicional. Para desambiguar estas situaciones, **pandas** ofrece dos métodos: `<objeto>.loc[X]` accede al elemento con índice semántico `X`, mientras que `<objeto>.iloc[X]` accede al elemento con índice posicional `X`.

El Programa 12 muestra la diferencia de resultados con ambos métodos. Nótese que a través del método `iloc` es posible indexar posicionalmente un `DataFrame`. En el caso de `DataFrame`, es posible definir a qué eje `iloc` o `loc` hacen referencia mediante el argumento `axis` (0 para filas, 1 para columnas). Así, `frame.iloc(axis=1)[0]` retorna los datos de la primera columna de `frame`.

```
frame = pd.DataFrame({"Name" : ['Carlos', 'Pedro'],
    "Age" : [34,22]}, index=[1,0])
print('Frame')
print(frame)
print('Primera fila')
print(frame.iloc[0])
print('Elemento con index 0')
print(frame.loc[0])
```

**Programa 12.** Indexación posicional y semántica a través de `loc` e `iloc`.

```
Frame
   Name  Age
1  Carlos  34
0  Pedro  22
Primera fila
Name Carlos
Age  34
Name: 1, dtype: object
Elemento con index 0
Name  Pedro
Age  22
Name: 0, dtype: object
```

**Salida 12.** El output es diferente para `loc` e `iloc` cuando el índice semántico y el posicional no coinciden.

De la misma forma que se puede indexar por posición o referencia, **pandas** permite hacer slicing con índices ordinales y semánticos. El slicing con índices posicionales funciona de manera idéntica al resto de colecciones en Python. Slicing con índices semánticos es similar, pero con la peculiaridad de que se incluyen ambos extremos del rango (al contrario que con los posicionales, en los que el elemento de la derecha no se incluye). La forma general de slicing con índices semánticos es la siguiente:

```
<objeto>[<index1>:<index2>:<step>]
```

Slicing con índices semánticos funciona directamente con **Series** pero no con **DataFrame**. Para **DataFrame**, se puede hacer slicing posicional o semántico empleando los métodos **loc** e **iloc**, pero, en vez de utilizar un solo índice como en el Programa 12, se define una slice. Por ejemplo, en el frame definido en el Programa 12, los siguientes comandos obtienen los siguientes resultados definidos:

```
frame.iloc(axis=1)[0]      # primera columna (nombres)

frame.loc(axis=1)["Name"] # primera columna (nombres)

frame.iloc(axis=0)[1:]     # segunda fila ("pedro", 22)

frame.loc(axis=0)[1:]     # filas a partir del índice semántico 1 ("carlos" y "pedro")
```

**loc** e **iloc** son útiles para filtrar columnas si se desea trabajar solo con un subconjunto de ellas. En general, se utiliza `<frame>.iloc[X,Y]` para obtener los elementos con filas **X** y columnas **Y** (ambos pueden ser listas). `<frame>.loc[X,Y]` funciona de manera análoga pero con índices semánticos. Por ejemplo, en el frame de D&D del Programa 11, se pueden obtener solo las columnas **Strength** y **Vitality** con `character_data.loc[:,["Strength","Vitality"]]`, o todas las columnas desde **Name** hasta **Wisdom** con `character_data.loc[:, "Name":"Wisdom"]`.

Como en **NumPy**, es posible la indexación condicional, muy útil para la filtración o detección de datos. Por ejemplo, se pueden obtener las filas de los elementos mayores de edad con el comando `frame[frame["Age"] > 18]`. En general, al utilizar índices condicionales, Python retorna los elementos para los que la condición es **True**.

Por último, es posible utilizar índices posicionales y semánticos para eliminar datos de una serie o frame a través del método ya mencionado `<objeto>.drop(<índices>)`.

### 4.3.3. Ordenación de Series y DataFrame

Hay diversas formas de ordenar series y frames:

- `<objeto>.sort_index()` ordena el objeto alfabéticamente según el índice de cada elemento. Argumentos opcionales `axis=0` o `1` y `ascending=True` o `False`.
- `<serie>.sort_value()` es similar a `sort_index()`, pero se utiliza el valor de cada elemento para ordenar.
- `<frame>.sort_values(by=<columna>)` ordena las filas con respecto al valor en la `<columna>` de cada elemento. Acepta `ascending`.

Además de reordenar las estructuras, **pandas** puede ordenar categóricamente objetos asignando un valor ordinal a cada elemento. Esto es de utilidad cuando se desea hacer un ranking de valores. El método `<objeto>.rank()` retorna un objeto (frame o serie) en el que cada elemento es la posición que este ocupa en relación con el resto de su columna.



### Ejemplo

Con la matriz aleatoria `rand_matrix` siguiente:

```
rand_matrix = np.random.randint(6, size=(2,3))
frame = pd.DataFrame(rand_matrix , columns=list('ABC'))
```

	A	B	C
0	4	4	1
1	1	3	5

Un ranking queda de la siguiente forma:

```
frame.rank()
```

	A	B	C
0	2.0	2.0	1.0
1	1.0	1.0	2.0

Como en todas las funciones de ordenación, `rank()` acepta el argumento opcional `ascending`.

## 4.4. Operaciones con Series y DataFrame

Los objetos en **pandas** (series y frames) incluyen métodos aritméticos para sumar, restar, dividir, multiplicar, elevar a potencia, etc. entre objetos del mismo tipo. Este es un listado de algunos de los métodos aritméticos más comunes y su símbolo equivalente:

```
<objeto1>.add(<objeto2>) == <objeto1> + <objeto2>
<objeto1>.sub(<objeto2>) == <objeto1> - <objeto2>
<objeto1>.mul(<objeto2>) == <objeto1> * <objeto2>
<objeto1>.div(<objeto2>) == <objeto1> / <objeto2>
```

Internamente, **pandas** compara los elementos de los objetos de la operación y, si está presente en ambos, el resultado es la aplicación del operando. Si el elemento solo aparece en uno, se añade un nuevo elemento al resultado con valor `NaN`. Si se prefiere, se puede cambiar el comportamiento por defecto de rellenar con `NaN`. Para ello, se pasa el argumento `fill_value=<valor>` a cualquiera de los métodos. Por ejemplo, `serie1.add(serie2, fill_value=0)` rellenará las casillas donde no se puede aplicar la suma con 0.



Nótese que, al aplicar una operación a un frame o una serie, la operación debe estar definida y, por lo tanto, ser válida para el tipo de elemento contenido en el objeto. Es decir, para sumar dos **Series**, el operador `+` debe estar definido para el tipo de elemento en la **Series**. Si el tipo de objeto es **string**, nótese que el signo `+` está definido como concatenación y es válido, pero el signo `-` no lo es, de modo que Python retornará un error `unsupported operand type(s) for -: 'str' and 'str'`.

A los operadores aritméticos se les suman otros operadores, como los operadores lógicos:

<objeto1> or <objeto2>

<objeto1> not <objeto2>

Es técnicamente posible aplicar operaciones aritméticas entre **Series** y **DataFrame**, para lo que **pandas** alinea basándose en el índice para ejercer la operación. Un uso común es, por ejemplo, ver la diferencia de una fila en un **DataFrame** respecto al resto. Se puede ver un ejemplo de ello en el Programa 13. Como vimos en el capítulo sobre **NumPy**, este tipo de operaciones entre elementos bidimensionales y unidimensionales es posible gracias al broadcasting.

```
rand_matrix = np.random.randint(10, size=(3, 4))
print(rand_matrix)

df = pd.DataFrame(rand_matrix , columns=list('ABCD'))
print(df - df.iloc[0])
```

**Programa 13.** Ejemplo de operación aritmética entre un frame y una serie.

```
[[5 4 2 8]
 [5 1 1 3]
 [6 3 3 7]]

   A  B  C  D
0  0  0  0  0
1  0 -3 -1 -5
2  1 -1  1 -1
```

**Salida 13.** El resultado final muestra la diferencia de todas las filas con la primera.



Como **pandas** está construido sobre **NumPy**, todas las operaciones unarias y binarias descritas anteriormente y recogidas en la Tabla 2 son aplicables a series y frames.

Otra herramienta muy versátil es la aplicación de funciones lambda a series y frames mediante el método <objeto>.apply(<función\_lambda>). Por defecto, la función lambda será aplicada columna por columna, aunque se puede pasar el argumento **axis** para definir el objetivo (0 para filas, 1 para columnas). El Programa 14 muestra un ejemplo de la utilización de funciones lambda para calcular la diferencia entre el valor máximo y mínimo en cada columna.





```
rand_matrix = np.arange(12).reshape(3, 4)
frame = pd.DataFrame(rand_matrix , columns=list('ABCD'))
print(frame)
frame.apply(lambda x : x.max() - x.min())
print(frame)
```

**Programa 14.** Uso de funciones lambda para averiguar la diferencia entre valores en cada columna de un frame.

```
   A  B  C  D
0  0  1  2  3
1  4  5  6  7
2  8  9 10 11
```

```
A  8
B  8
C  8
D  8
dtype: int64
```

**Salida 14.** La función lambda es aplicada a cada columna del frame.

## 4.5. Estadística con Series y DataFrame

Para entender mejor los datos que se poseen, es muy útil la exploración. **pandas** ofrece varios métodos de estadística sumaria para **Series** y **DataFrame** que ayudan a ver los datos en su conjunto.

En **Series** y **DataFrames**:

- `<objeto>.sum()` retorna el valor acumulado de todos los valores (en series) o por columna (en frames). Acepta el argumento `axis` para determinar si la suma se hace por fila o columna.
- `<objeto>.mean()` retorna lo mismo que `sum` pero con la media de valores.
- `<objeto>.cumsum()` retorna una suma acumulativa por elementos. Cada elemento es el valor acumulado de los elementos precedentes.
- `<objeto>.idxmin()` y `<objeto>.idxmax()` retorna el índice semántico (fila) del elemento menor o mayor de cada columna.

La Tabla 5 contiene algunos de los métodos de estadística sumaria más importantes para series y frames.

**Tabla 5***Métodos más comunes de estadística descriptiva para Series y DataFrame*

Método	Descripción
<b>count</b>	Número de valores no NaN
<b>describe</b>	Conjunto de estadísticas sumarias
<b>min, max</b>	Valores mínimo y máximo
<b>argmin, argmax</b>	Índices posicionales del valor mínimo y máximo
<b>idxmin, idxmax</b>	Índices semánticos del valor mínimo y máximo
<b>sum</b>	Suma de los elementos
<b>mean</b>	Media de los elementos
<b>median</b>	Mediana de los elementos
<b>mad</b>	Desviación absoluta media del valor medio
<b>var</b>	Varianza de los elementos
<b>std</b>	Desviación estándar de los elementos
<b>cumsum</b>	Suma acumulada de los elementos
<b>diff</b>	Diferencia aritmética de los elementos

Además de la estadística descriptiva, **pandas** reúne métodos para entender la relación matemática entre dos conjuntos de valores.

- `<serie1>.corr(<serie2>)`: correlación entre las dos series `[-1,1]`.
- `<frame>.corr()`: matriz de correlación en el frame. Mide la variación entre columnas.
- `<frame1>.corrwith(<frame2>)`: grado de correlación entre frames. Acepta el argumento `axis` para determinar la base de la comparación (0 filas, 1 columnas).

# Capítulo 5

Gran parte del tiempo del analista de datos se pasa en esta fase exploratoria, y sin duda una de las técnicas más utilizadas es la visualización de datos. Representar visualmente los datos puede complementar la exploración de los datos, pero también ayudar a generar ideas para modelos en fases avanzadas. Al otro lado del espectro, representar los resultados de los modelos implementados de forma gráfica ayuda a cuantificar las mejoras alcanzadas y presentar nuestras ideas de forma concisa.

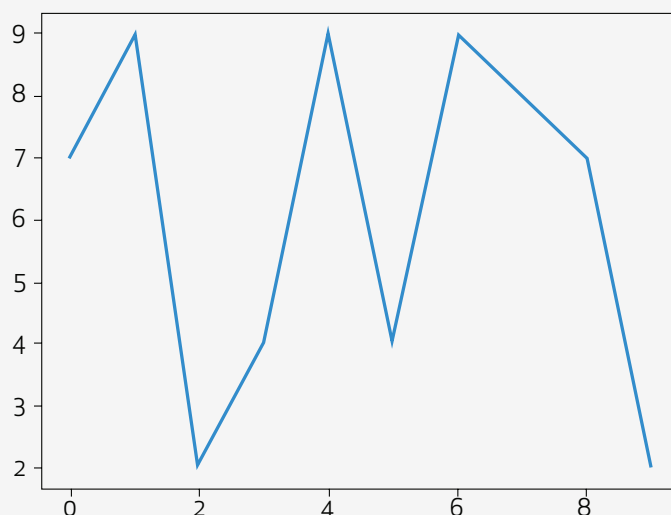
```
import matplotlib.pyplot as plt
```

## 5.1. matplotlib 101

La forma más directa de crear un gráfico es a través del método `plot(<data>)`, donde `<data>` es el conjunto de datos que se quiere representar. El Programa 15 muestra un ejemplo.

```
import numpy as np
import matplotlib.pyplot as plt
data = np.random.randint(10,size=(10))
data # array([7, 9, 2, 4, 9, 4, 9, 8, 7, 2])
plt.plot(data) # creación del gráfico
plt.show() # comando que muestra los gráficos creados hasta el momento
```

**Programa 15.** Representación de datos de forma gráfica con `plt.plot()`.



**Salida 15.** Gráfico que representa los datos generados por NumPy.

Como se puede observar en el Programa 15, `plt.plot()` recibe los datos que se desean representar. En su forma general, la función acepta dos listas, una para cada eje (X e Y), en el que cada elemento es un punto en el gráfico. En este ejemplo, como solo se proporciona una lista de valores, **matplotlib** asume que son los que representan la coordenada Y, y asume que la X son valores de 0 a 9 (o hasta el tamaño de la lista). En general, la siguiente expresión genera un gráfico utilizando los datos de `x_data` e `y_data`:

```
plt.plot(x_data,y_data)
```

Como se indica en el Programa 15, Python no presenta el gráfico directamente al usuario, sino que se mantiene en la memoria hasta que se ejecuta el comando `plt.show()`. Esto se puede utilizar para trabajar con varios gráficos al mismo tiempo y presentarlos todos a la vez con un solo comando.

Los gráficos en **matplotlib** residen dentro del objeto **Figure**. Este objeto se utiliza para representar un gráfico, pero también permite la subdivisión para múltiples gráficos. Para obtener una referencia al objeto

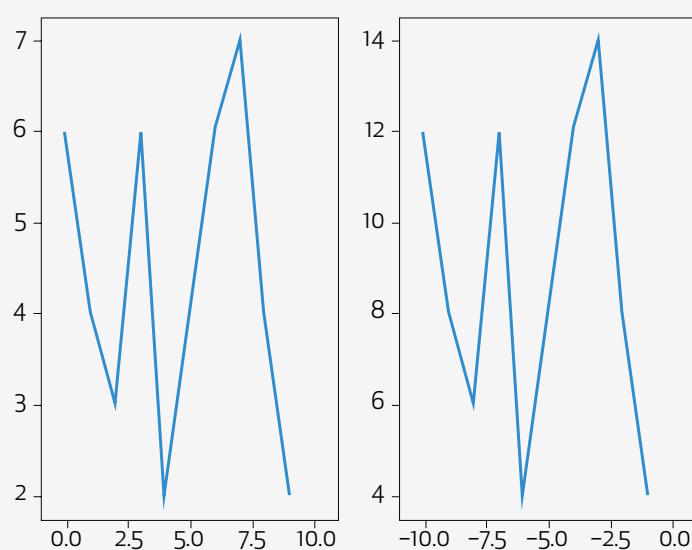
general, se puede ejecutar `fig = plt.figure()`, que retorna dicha referencia. Una vez obtenida la referencia, se puede subdividir la figura con `fig.add_subplot(X,Y,1)`, que divide la figura en `X` filas e `Y` columnas. No obstante, **matplotlib** presenta un método directo para generar figuras y subfiguras:

```
fig, axes = plt.subplots(X,Y)
```

`subplots` retorna una referencia a la figura en general (`fig`) y una `ndarray` bidimensional (`X` filas e `Y` columnas) con referencias a cada subfigura (`axes`). Una vez obtenida una referencia para cada subfigura, se puede proceder a dibujar en cada una con el método `plot()`. Nótese que, al ser una `ndarray`, `axes` puede indexarse fácilmente por coordenadas; así, se accede a la subfigura en la primera fila y segunda columna a través de `axes[0,1]`. El Programa 16 muestra un ejemplo de cómo trabajar con varias subfiguras.

```
x_data1 = np.arange(0,10)
x_data2 = np.arange(-10,0)
y_data1 = np.random.randint(10,size=(10))
y_data2 = y_data1 * 2
fig, axes = plt.subplots(1,2)
#preparar subfigura1
axes[0].set_xlim([-1,11])
axes[0].plot(x_data1,y_data1)
#preparar subfigura2
axes[1].set_xlim([-11,1])
axes[1].plot(x_data2,y_data2)
#mostrar ambas figuras
fig.show()
```

**Programa 16.** Ejemplo de creación de gráficos en distintas subfiguras.



**Salida 16.** Cada gráfico se muestra en una de las dos subfiguras creadas.



### Enlace de interés

**matplotlib** tiene una colección extensa de métodos para configurar cada gráfico como el rango de valores de cada eje, el espacio entre figuras, los colores a utilizar, etc. A continuación se ofrece un enlace a la API para más información:

[https://matplotlib.org/api/pyplot\\_api.html](https://matplotlib.org/api/pyplot_api.html)

Sin embargo, es una documentación muy extensa, por lo que se recomienda visitar antes la página de ejemplos de figuras para descubrir cómo realizar gráficos de distintos tipos:

[https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.figure.html](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.figure.html)

## 5.2. Múltiples gráficos

Tanto el método general `plt.plot()` como el que reside en figuras y subfiguras aceptan varios argumentos que afectan a cómo se representa la información. Entre los más utilizados se encuentran `plt.plot(x, y, style, label, linewidth=1, alpha=1)`:

- **style:** `string` que define el estilo con el que dibujar los datos. Sigue la forma '`[color][marcador][línea]`', en la que cada opción se define como en la Tabla 6. Un ejemplo es '`ro-`' para marcadores circulares, rojos con líneas continuas.
- **label:** utilizado en la leyenda como referencia al gráfico actual.
- **linewidth:** ancho de la línea al dibujar.
- **alpha:** nivel de opacidad. Se puede utilizar para dar transparencia a la figura.

**Tabla 6**

Información sobre cómo formar la `string style` para determinar el estilo con `plot()`.

Color	Descripción	Marcador	Descripción
<b>b</b>	blue	.	point marker
<b>g</b>	green	,	pixel marker
<b>r</b>	red	o	circle marker
<b>c</b>	cyan	v	triangle_down marker
<b>m</b>	magenta	^	triangle_up marker
<b>y</b>	yellow	<	triangle_left marker
<b>k</b>	black	>	triangle_right marker
<b>w</b>	white	1	tri_down marker
		2	tri_up marker
		3	tri_left marker
		4	tri_right marker
		s	square marker
		p	pentagon marker
		*	star marker
		h	hexagon1 marker
		H	hexagon2 marker
		+	plus marker
		x	x marker
		D	diamond marker
		d	thin_diamond marker
			vline marker
		-	hline marker
Línea	Descripción		
-	solid line style		
--	dashed line style		
-.	dash-dot line style		
:	dotted line style		



### Enlace de interés

Para más información sobre el método `plot()` y sus argumentos:

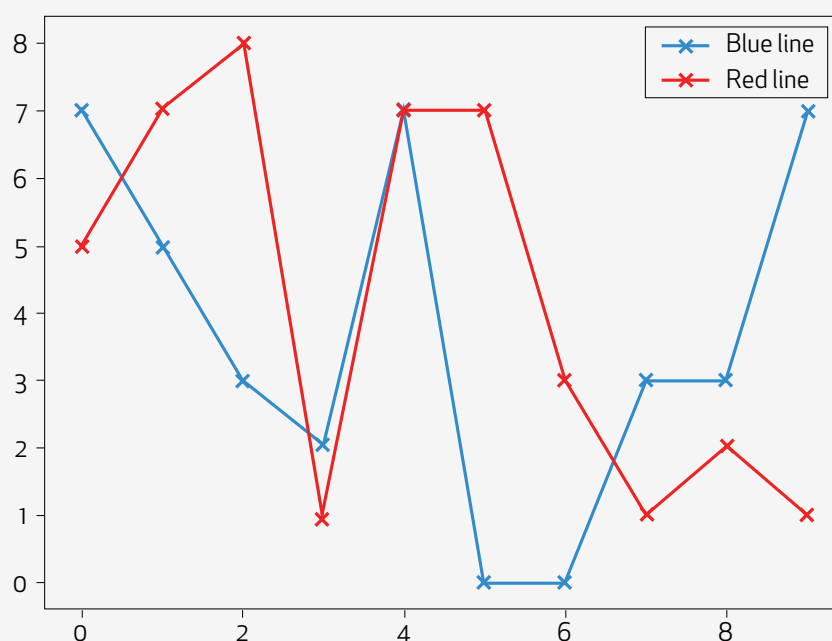
[https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.plot.html](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html)

En ocasiones es importante representar más de un gráfico dentro de una misma figura, por ejemplo, para comparar dos tendencias o múltiples alternativas. Para ello, podemos hacer más de una llamada a la función `plot()` **en la misma figura o subfigura**. Para ayudar a la visualización de elementos distintos, cada llamada puede tener diferentes parámetros de `style` y `label`. Cuando se tiene más de un gráfico en una figura, tradicionalmente se añade una leyenda que nombre cada uno de ellos. Para mostrar la leyenda, se utiliza el método `legend(loc='best')`, bien el general a través de `plt.legend()` o en la figura o subfigura.

El Programa 17 muestra un ejemplo de cómo representar dos gráficos en una figura.

```
x_data = np.arange(10)
y_data1 = np.random.randint(10,size=(10))
y_data2 = np.random.randint(10,size=(10))
fig, axes = plt.subplots(1) #solo solicitamos una figura
axes.plot(x_data,y_data1,'bx-',label='Blue line')
axes.plot(x_data,y_data2,'rx-',label='Red line')
axes.legend(loc='best')
fig.show()
```

**Programa 17.** Dos gráficos dentro de la misma figura.



**Salida 17.** Los dos gráficos residen dentro de la misma figura y se incluye una leyenda.

## 5.3. Decoraciones y anotaciones

Además de poder definir el estilo del gráfico en la función `plot()`, **matplotlib** ofrece un conjunto de métodos **decoradores** que ayudan a modificar el aspecto de la figura. Desde cambiar los límites representados en el gráfico, hasta añadir títulos, hay una gran colección de métodos tanto a través de `plt` como en la clase `Axes`. Siguiendo el estilo de programación orientada a objetos, puede decorarse cada subfigura a través del objeto `axes` (como hasta ahora). Algunos de los decoradores más habituales son:

- `axes.set_xticks(<lista>)` y `set_yticks(<lista>)` definen explícitamente qué números aparecen escritos en los ejes X e Y.
- `axes.set_title('titulo')` establece el título general de la subfigura.
- `axes.set_xlabel('label')` y `set_ylabel('label')` establecen el nombre para cada eje.



### Enlace de interés

Más información sobre decoradores en el siguiente enlace:

[https://matplotlib.org/api/axes\\_api.html#plotting](https://matplotlib.org/api/axes_api.html#plotting)

A través de anotaciones se puede añadir texto y otros dibujos a los gráficos. Se puede añadir texto directamente en las coordenadas  $(x, y)$  con el método `axes.text(x, y, 'texto')`. Para facilitar las anotaciones, **matplotlib** ofrece el método `axes.annotate()`, que permite no solo añadir texto, sino incluir flechas. La forma general del método es la siguiente:

```
annotate(texto, xy, xytext, xycoords, arrowprops)
```

`texto` es la `string` que queremos representar; `xy` son las coordenadas (tupla) en las que deseamos poner la anotación; `xytext` son las coordenadas (tupla) donde se escribe el texto; `xycoords` define el sistema de coordenadas en el cual `xy` está expresado; `arrowprops` es un diccionario que define el tipo y estilo de flecha utilizado.

La siguiente anotación produce el gráfico de más abajo:

```
ax.annotate('resaltar', xy=(2, 1), xytext=(3, 1.5),  
arrowprops = dict(facecolor='black', shrink=0.05))
```

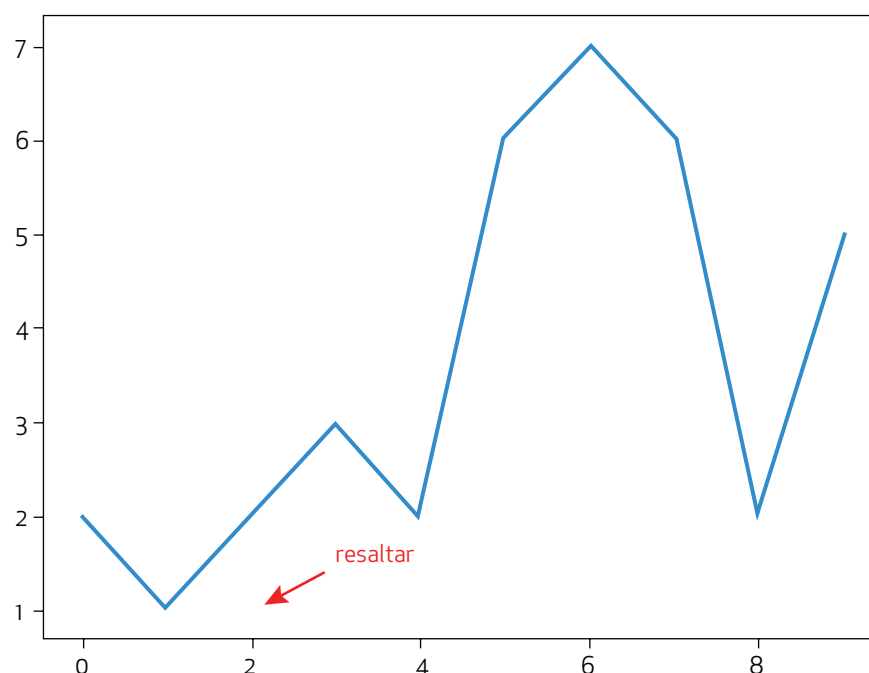


### Enlace de interés

Más información y ejemplos de anotaciones en el siguiente enlace:

<https://matplotlib.org/users/annotations.html>





**Figura 5.** Ejemplo de anotación en un gráfico.

## 5.4. Tipos de gráficos

**matplotlib** incluye funciones específicas para representar la información de distintas formas, como scatter plot, histograma, diagrama de pastel, gráfico con errores, etc. En este curso utilizaremos la interfaz de **pandas** (véase epígrafe 5.6) para elegir el tipo de gráfico para abstraernos de los detalles de bajo nivel.



### Enlace de interés

No obstante, el siguiente enlace ofrece información de cómo utilizar **matplotlib** para este fin:  
[https://matplotlib.org/api/axes\\_api.html#plotting](https://matplotlib.org/api/axes_api.html#plotting)

## 5.5. Grabación de gráficos a archivo

Una vez producida la figura (individual o múltiple), `plt.savefig(<filename>)` la guarda en el disco duro. `<filename>` indica al mismo tiempo el directorio de destino, el nombre del archivo a utilizar y la extensión.

**matplotlib** puede deducir el formato de la imagen con la extensión de `<filename>`, pero se puede forzar con el parámetro `format`, que acepta una `string` con el nombre de la extensión (**matplotlib** da soporte a una multitud de formatos, entre los que destacan `jpg`, `png` para imágenes de píxeles y `svg`, `eps` y `pdf` para imágenes vectoriales). La forma general de la función es, con parámetros por defecto:

```
plt.savefig(<filename>, dpi=None, facecolor='w', edgecolor='w', format=None,
bbox_inches=None)
```

- `<filename>` indica el directorio, archivo y extensión para guardar la imagen.
- `dpi`, del inglés *dots per inch*, determina la resolución de la imagen (siendo 300 buena calidad).
- `facecolor` y `edgecolor` son los colores a utilizar de fondo y en los bordes (blanco, white 'w', por defecto)
- `format` es el `string` que indica el formato de la imagen ('pdf', 'png', 'svg', 'eps' ...).
- `bbox_inches`, del inglés *bounding box*, indica el tamaño del marco a considerar alrededor del gráfico.

## 5.6. Representación con pandas

**matplotlib** es una librería muy versátil y potente para la producción de gráficos de muchos tipos, No solo para la representación de datos, sino también para la creación de diagramas y esquemas. Sin embargo, requiere conocimiento de una gran cantidad de funciones, con multitud de parámetros que pueden resultar complejos para el usuario medio.

Para disminuir la dificultad en la representación gráfica de datos, **pandas** ofrece una interfaz para la creación de gráficos a partir de objetos **Series** y **DataFrames**. Para ello, ambos objetos tienen un método `plot()` que utiliza **matplotlib** para visualizar los datos.

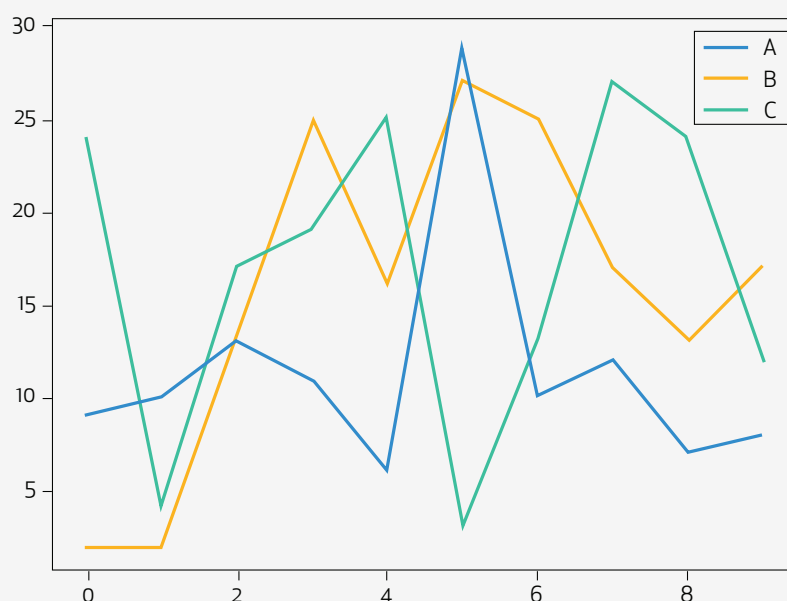
Por defecto, en una **Serie** o **DataFrame**:

- El índice se utiliza como coordenada en el eje X.
- La lista de valores en una serie, o cada columna en un frame, describen una línea del gráfico (cada valor se utiliza como coordenada en el eje Y).
- En frames, el nombre de la columna determina la referencia en la leyenda.

El Programa 18 muestra cómo se representa un **DataFrame** visualmente en **pandas**.

```
rand_matrix = np.random.randint(30,size=(10,3))
frame = pd.DataFrame(rand_matrix , columns=list('ABC'))
frame.plot()
plt.show() # aún es necesario llamar a show() para mostrar los gráficos
```

**Programa 18.** Uso de la interfaz gráfica de **pandas**.



**Salida 18.** Resultado de representar un DataFrame con `plot()`.

Como en **matplotlib**, se pueden pasar argumentos a la función `plot()` para refinar el resultado de la figura.

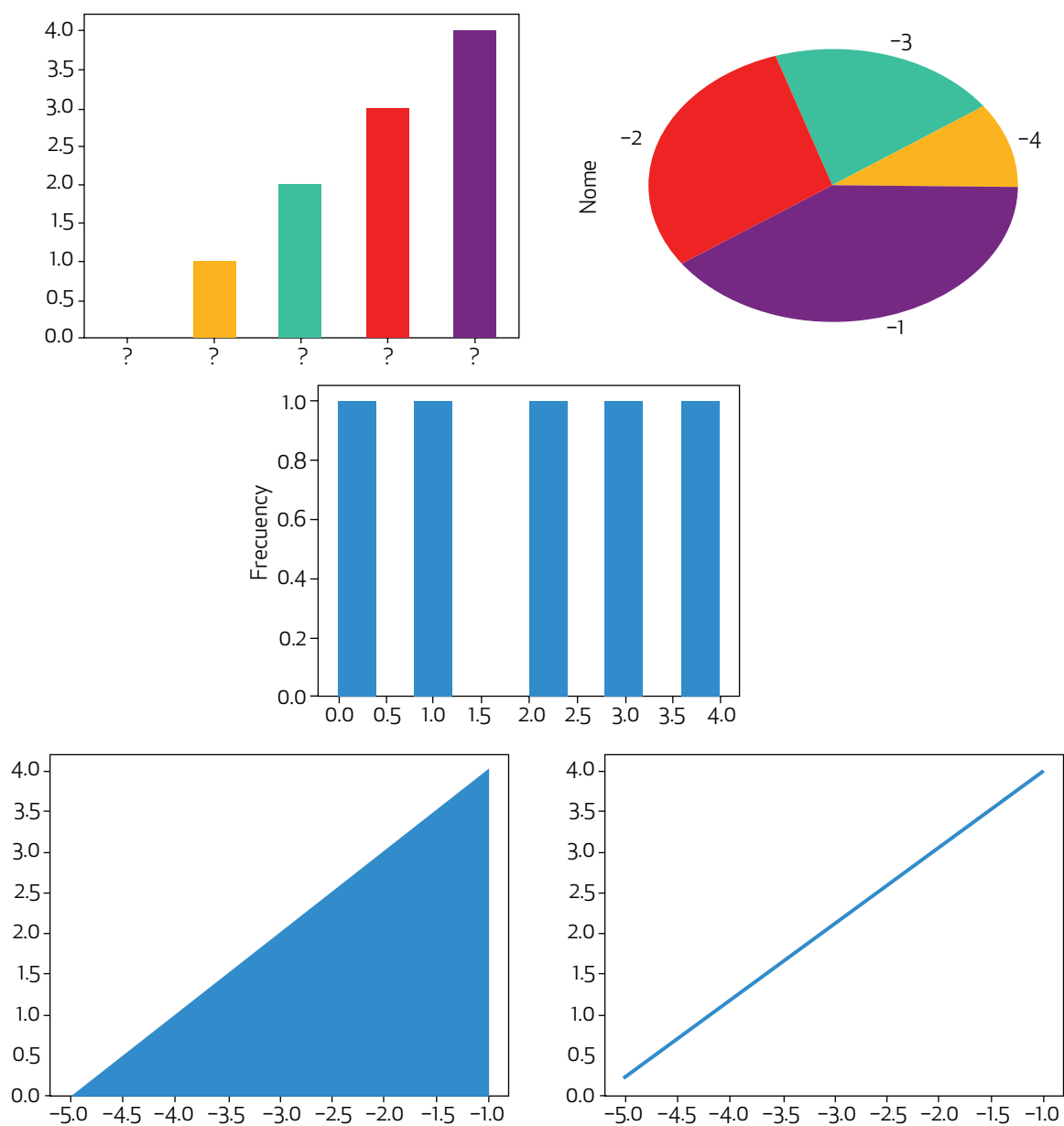
Los parámetros más utilizados son:

- **(Series) label**: referencia a utilizar en la leyenda
- **ax**: subfigura en **matplotlib** en la que dibujar los datos (útil cuando se trabaja con múltiples subfiguras).
- **style**: **string** que define el estilo de la línea (más información en la Tabla 6).
- **kind**: tipo de gráfico a representar ('bar', 'pie', 'hist', 'area', 'line', 'barh', 'density', 'kde').
- **use\_index**: **bool** que determina si utilizar el índice en los ticks del eje X.
- **xticks** e **yticks**: valores explícitos en los ejes X e Y.
- **title**: **string** a utilizar como título de la figura.
- **(DataFrame) subplots**: **bool** para indicar si se desean subfiguras separadas para cada columna.

La Figura 6 muestra cinco tipos de gráficos obtenidos variando el parámetro **kind** en la representación de la siguiente serie en **pandas**:

```
serie = pd.Series(np.arange(5), index=np.arange(-5, 0))
```

Los tipos utilizados son 'bar', 'pie', 'hist', 'area' y 'line'.



**Figura 6.** Diversas representaciones gráficas de una serie variando el parámetro `kind` del método `plot()`. De izquierda a derecha, y de arriba abajo: 'bar', 'pie', 'hist', 'area' y 'line'.

01

```

...
mirror_mod.use_x = False
mirror_mod.use_y = True
mirror_mod.use_z = False
operation = "MIRROR_Z"
mirror_mod.use_x = False
mirror_mod.use_y = False
mirror_mod.use_z = True

[ (1+x+y+2a)-(3a+3g+x)
  5+x+k+2a+21
  E=mc2
  1lim h-->0
  2+....+2a....+a
  selection at the end -add
  _ob.select= 1
  ler_ob.select=1
  text.scene.objects.acti
  ("Selected" + str(modifier.y+2a+21
  error_ob.select = 0
  bpy.context.selected_ob
  sta.objects[one.name].se
  Int("please select exactl
  2+ +2a +a

```

## Capítulo 6

### Python para ciencia de datos

En este último capítulo vamos a describir tres aspectos fundamentales para el científico de datos en Python. En primer lugar, descubriremos cómo leer y escribir datos en el disco duro, importante para cargar los datos a nuestro modelo y guardar el progreso. También describiremos desde una perspectiva pragmática cómo manipular `strings`. Y finalmente introduciremos el tema de la **limpieza de datos** para su utilización tanto en el análisis de datos como en aprendizaje automático.

#### 6.1. Lectura y escritura de archivos

Cargar y guardar datos es fundamental para evitar la transitoriedad del intérprete de Python.

De poco serviría construir modelos predictivos con alta precisión si no fuésemos capaces de almacenarlos para su uso en producción. O sería poco productivo (o frecuentemente imposible) tener que generar los datos para entrenar nuestros modelos cada vez que quisiéramos utilizar dicho modelo.

Hay varias formas de interactuar con el disco duro en Python, desde la escritura y la lectura directa con streams, hasta el uso de las interfaces de **NumPy** y **pandas**.

### 6.1.1. En Python

Como en cualquier lenguaje de programación, Python ofrece clases y métodos para trabajar con **streams** o flujos de datos, que pueden utilizarse para leer y escribir datos en el disco duro. La forma básica de abrir un stream es a través de la función `f = open(<path>)`, que retorna una referencia al stream que puede usarse para leer y escribir información sobre el archivo indicado en `<path>`. Una vez se ha terminado de trabajar con el archivo, se debe cerrar el stream con `f.close()`.

Para evitar tener que recordar cerrar el stream, se puede utilizar la construcción `with` en Python, que permite definir un alias temporal de la siguiente forma:

```
with open(<path>) as f:
    # trabajar con f
```

De esta forma, se puede trabajar con el stream definido como `f` sin necesidad de cerrarlo al final.

Para leer datos del stream, puede hacerse por líneas, iterando de forma natural:

```
for line in f:
    # line contiene una línea en cada iteración
```



Dependiendo del formato del archivo y del sistema operativo, al leer una línea en un archivo de texto es posible que exista un *EOF character* (carácter indicador de final de línea). Python posee un método para eliminar dicho carácter de una `string`: `st.rstrip()`. Así, se pueden obtener todas las líneas de un stream, sin incluir EOF, filtrando de la siguiente forma:

```
lines = [x.rstrip() for x in f]
```

También es posible leer todas las líneas de una vez con `f.readlines()`, que retorna una secuencia con todas las líneas. Alternativamente, se puede solicitar la lectura de cierto número de caracteres con `f.read(5)`. En el caso de que sea adecuado hacerlo, se puede navegar a través del stream con las funciones `f.seek(<posición>)`, que mueve el puntero actual (posición actual de lectura) a la posición indicada, y `f.tell()`, que retorna la posición actual del stream.

Al abrir un stream, Python otorga un modo de acceso. Si no se define, por defecto el stream tiene solo modo de lectura, por lo que se puede leer información del archivo, pero no escribir. Para definir otro modo de acceso, se utiliza el segundo parámetro de la función `open(<path>, <mode>)`, donde `<mode>` es una de las `strings` indicadas en la Tabla 7.

Para escribir datos a un stream (abierto con el modo adecuado, `'w'`, `'x'`, `'a'`, o `'r+'`), se pueden utilizar los métodos `f.write(<string>)` o `f.writelines(<secuencia>)` para escribir una `string` literal o una secuencia de `strings` (cada una en una línea) respectivamente.


**Tabla 7***Descripción de los modos de acceso en streams Python*

Modo de acceso	Descripción
<b>r</b>	Solo lectura
<b>w</b>	Solo escritura (borra el archivo si ya existe)
<b>x</b>	Solo escritura (falla si el archivo ya existe)
<b>a</b>	Crea un archivo (si existe lo abre y añade al final)
<b>r+</b>	Lectura y escritura
<b>b</b>	Se puede añadir a otros modos para acceso binario
<b>t</b>	Modo texto para archivos de texto

Leer y escribir datos a un archivo de texto es suficiente en muchos casos. Sin embargo, dada la ubicuidad de los datos tabulares en big data y aprendizaje automático, es conveniente tener métodos para tratar con esta información directamente. Así, Python ofrece soporte para leer y procesar archivos CSV, estándar en la descripción de tablas.

Tanto para importar como para escribir datos tabulares en formato CSV, el módulo `csv` contiene funciones idóneas. `data = csv.read(<stream>,<delimiter>)` construye un lector de CSV iterable.

Como los archivos CSV pueden tener los elementos separados por varios caracteres, el parámetro `<delimiter>` se utiliza para indicarlo. Cada iteración retorna una colección con los elementos de una línea de la tabla, en la que el elemento con índice 0 es la primera columna, etc. El Programa 19 contiene un ejemplo de cómo leer e iterar sobre los valores de una tabla CSV.



```
# Tabla1.csv contiene la Tabla 1 del manual,
# con valores separados por comas

import csv
f = open('Tabla1.csv')
data_reader = csv.reader(f,delimiter=',')
with open('Nueva_tabla.csv','w') as f2:
    csv_writer = csv.writer(f2,delimiter=';')
    for line in data_reader:
        print(line[0])
        csv_writer.writerow(line)
f.close()
```

**Programa 19.** Lectura, procesado y guardado de elementos en archivos CSV.

**Operación**

```

a + b
a - b
a / b
a // b
a * b
a ** b
a % b
a & b
a | b
a ^ b
a == b
a != b
a is b
a is not b
a <= b, a < b
a >= b, a > b

```

**Salida 19.** Resultado de la lectura del archivo CSV, en el que solo se imprimen los elementos de la primera columna. Como resultado, el archivo CSV es copiado a 'Nueva\_tabla.csv' pero utilizando el carácter ; como separador.

### 6.1.2. En NumPy

**NumPy** ofrece su propia interfaz para leer y escribir `ndarrays` en disco. Por convención, estos archivos tienen el formato `'.npy'`. Para guardar cualquier la `ndarray` `array1` en el disco, se utiliza el método `np.save(<filename>, array1)`.

También se pueden guardar múltiples `ndarrays` al mismo tiempo con `np.savez(<filename>, a=arr1, b=arr2)` sin comprimir, o `np.savez_compressed(<filename>, a=arr1, b=arr2)`.

En cualquier caso, el método `loaded = np.load(<filename>)` carga la `ndarray` del archivo `<filename>`. Si el archivo contiene varias `ndarrays`, `loaded` es un diccionario con todas.

### 6.1.3. En pandas

De la misma forma que **NumPy** puede grabar y leer `ndarrays` directamente del disco, **pandas** ofrece formas de manipular datos tabulares. El método más común es `pd.read_table`, que genera un `DataFrame` o `Series` a partir de la información en el archivo CSV:

```
object = pd.read_table(<filename>, names=<nombres_de_columnas>, index_col=<columna_como_index>, sep=<caracter_separador>, na_values=<string_na>)
```



Descripción de los parámetros:

- `names=<nombres_de_columnas>`: lista de `strings` a utilizar como nombres de columnas.
- `index_col=<columna_como_index>`: columna a utilizar como `label` o índice de las filas en el `DataFrame` o `Series`.
- `sep=<caracter_separador>`: carácter a utilizar como separador en el archivo CSV.
- `na_values=<string_na>`: lista de `strings` en el CSV que serán interpretados como valores nulos `NaN`. Puede también indicarse un diccionario en el que cada clave es el nombre de una columna, y el valor es una lista de elementos que se interpretarán como `NaN` en esa columna.

Para grabar en un archivo, tanto `DataFrame` como `Serie` tienen un método:

```
to_csv(<filename>, na_rep=<string_NaN>, columns=<columnas_a_grabar>,  
sep=<separador>)
```

El parámetro `na_rep` se utiliza para indicar la `string` que **pandas** utilizará en el CSV para valores `NaN` y nulos. `columns` sirve para seleccionar un subconjunto de columnas a grabar (si no se desea grabar todo el objeto). `sep` es para indicar el carácter separador utilizado en el CSV.



#### Enlace de interés

**pandas** también ofrece interfaces para leer archivos en formatos estándares como JSON, Excel o HDF5. Para más información, el enlace a continuación contiene la API de **pandas** relacionada con input y output, incluyendo lectura y escritura de varios formatos:

<https://pandas.pydata.org/pandas-docs/stable/api.html#input-output>

## 6.2. Trabajo con strings

En el tratado y procesamiento de datos, dada la naturaleza de muchos datos, es frecuente tener que manipular `strings`. Algunas de las tareas habituales son: limpiar `strings` de espacios blancos innecesarios, dividir `strings` en unidades, combinar `strings` o forzar un formato particular.

En Python, para utilizar caracteres de escape en `strings`, como tabulación (`'\t'`) o fin de línea (`'\n'`), se marcan con el carácter `'\'` primero. Esto quiere decir que, internamente, Python interpreta cualquier `'\'` en una `string` como un carácter de escape. No obstante, este no es siempre cierto, y el caso más común es cuando se desea expresar un directorio completo en Windows. Si se desea incluir el carácter `'\'` como tal en una `string`, se debe indicar como doble `'\\'`. Otra alternativa es marcar la `string` con el prefijo `r`, lo que indica a Python que no hay caracteres de escape. Las dos `strings` siguientes son equivalentes:

```
s1 = 'c:\\windows\\user\\myfiles'  
s2 = r'c:\windows\user\myfiles'
```

Formar una `string` con una plantilla determinada es conveniente, por ejemplo, cuando se desea imprimir cierta información de forma homogénea. Para ello, Python incluye la función `<string>.format()`, que asocia marcadores dentro de la `string` con valores absolutos en los parámetros.



### Ejemplo

```
# Uso de una plantilla para formar strings
template = "{0:.2f} {1:s} equivalen a {2:d} euros"
euros= template.format(9,'libras esterlinas',10)
dolares = template.format(11,'dolares',10)
# euros tiene el valor "9 libras esterlinas equivalen a 10 euros"
# dolares tiene el valor "11 dolares equivalen a 10 euros"
```

Manipular `strings` es sencillo:

- Para dividir una `string` en un listado de elementos mediante un separador, se puede utilizar el método `<string>.split(<separador>)`, que retorna dicho listado.
- Para unir elementos de una lista en una `string` con un carácter de por medio (operación inversa a 1), `<string_intermedia>.join(<listado>)` forma una `string` concatenando los elementos del `<listado>` y añadiendo `<string_intermedia>` entre medio.
- `<original>.replace(<string_a>,<string_b>)` reemplaza las ocurrencias de `<string_a>` en la `string` `<original>` con `<string_b>`.
- `<string>.strip()` elimina espacios en blanco a ambos lados de `<string>`.
- `<string>.upper()` y `<string>.lower()` convierten la `<string>` en mayúsculas y minúsculas respectivamente.

Otra tarea frecuentemente realizada es la de buscar un fragmento dentro de una `string`. Python permite comprobar la presencia con un simple `<substring> in <string>`, que retorna `True` cuando `<substring>` forma parte de `<string>`. Por ejemplo:

```
genre = 'comedy|drama|romantic'
'comedy' in genre # retorna True
'thriller' in genre # retorna False
```

También se puede averiguar el índice de la primera ocurrencia de un carácter con `<string>.index('@')`, que retorna un número con la posición de la primera ocurrencia, por ejemplo, para separar el dominio del nombre de usuario en una dirección de correo. `<string>.find(<caracter>)` funciona de manera similar pero retorna `-1` si no encuentra el carácter. Se puede averiguar el número de ocurrencias de un carácter con `<string>.count(<caracter>)`.

## 6.3. Limpieza de datos

Los modelos de predicción son tan buenos como la información que se usa para construirlos y entrenarlos ("garbage in, garbage out"). Por lo tanto, no es de extrañar que gran parte del tiempo que emplean los analistas de datos lo dedican a explorar y transformar la información que poseen para llevarla a un estado en el que pueda ser utilizada.

Esta sección intenta ser una breve introducción a cómo realizar algunas de las tareas de limpieza de datos, y no una descripción exhaustiva de las técnicas o filosofías que hay detrás de ellas.

### 6.3.1. Aplicación de funciones a una colección

Cuando se poseen datos en cualquier colección de Python, en ocasiones se desea aplicar una medida global (por ejemplo, una función que escale los valores) a todos los elementos. Python posee tres tipos de funciones globales:

**map** aplica una función a todos los elementos de una colección y retorna una secuencia sin materializar con los resultados. La forma general es `list(map(<funcion>, <coleccion>))`.

**filter** retorna una secuencia sin materializar con solo los elementos de una colección que cumplen una condición establecida. La forma general es `list(filter(<funcion_booleana>, <coleccion>))`, donde `<funcion_booleana>` es una función que retorne `True` o `False`.

**Reduce** ejecuta una función con los elementos de una lista y retorna un escalar con el resultado. La forma general es `value = reduce(<funcion>, items)`.



La función `reduce` se encuentra en el módulo `functools`, por lo que requiere importación para ser utilizada.

En `map`, `filter` y `reduce` es un caso habitual en el que se suelen utilizar las funciones lambda para pasar funciones, ya que normalmente son expresiones cortas que no se reutilizan. El Programa 20 muestra ejemplos de las tres funciones globales.

```
from functools import reduce # necesario para reduce
# ejemplo de map para elevar todos los elementos al cuadrado
items = np.arange(10)
print(items)
squared = list(map(lambda x : x**2, items))
print(squared)
```

>>>

```
# ejemplo de filter para eliminar los elementos menores de 10
bigger_items = list(filter(lambda x : x > 10, squared))
print(bigger_items)
# ejemplo de reduce para obtener la suma de los elementos restantes
final_sum = reduce(lambda x,y : x + y, bigger_items)
print(final_sum)
```

**Programa 20.** Ejemplos de la aplicación de las tres funciones globales.

```
[0 1 2 3 4 5 6 7 8 9]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[16, 25, 36, 49, 64, 81]
271
```

**Salida 20.** map, filter y reduce usados como ejemplo de funciones globales sobre una colección.

### 6.3.2. pandas para transformar datos

Cuando se trabaja en proyectos de big data, es raro encontrarse con datasets listos para ser utilizados directamente. Frecuentemente los datos requieren de procesos de transformación para tratar valores no definidos, columnas no estandarizadas, elementos duplicados, etc. **pandas** incluye herramientas para facilitar el tratamiento y la limpieza de datos.

#### Valores no definidos

Recolectar datos es una tarea compleja y muchas veces inacabada, por lo que habitualmente se trabaja con datasets con valores incompletos. Para complicar las cosas, algunos algoritmos de aprendizaje automático y análisis de datos no toleran valores no definidos, por lo que estos huecos plantean un reto para el analista.

Hay varias formas de resolver valores no definidos

1. **Eliminar datos con valores nulos.** Pueden eliminarse elementos enteros si uno de sus valores es NaN. `<objeto>.dropna()` filtra dichos elementos. En el caso de **DataFrames**, si uno de los valores en una columna es NaN, la fila entera queda eliminada. Si se desea ser menos restrictivo, por ejemplo, para solo eliminar las filas que no contengan al menos 3 valores completos, puede indicarse con el argumento `thres=3`. Para eliminar solo los elementos con **todos** los valores NaN, se puede pasar el argumento `how='all'`.
2. **Sustituir los valores NaN (fijo).** En vez de eliminar elementos del dataset (particularmente difícil si la información es escasa), una alternativa es sustituir los valores NaN por valores fijos —el valor en concreto depende del problema en cuestión—. Tanto series como frames tienen

el método `<objeto>.fillna(<valor>)`, con el cual se sustituyen todos los NaN por `<valor>`. El valor puede ser uno específico por columna, pasando como argumento un diccionario (en el que la clave es la columna y el valor es el elemento sustitutivo).

3. **Sustituir los valores NaN (dinámico).** Utilizando el argumento `method` de `<objeto>.fillna(method=<metodo>)`, es posible rellenar los valores NaN de forma dinámica. Por ejemplo, `'backfill'` utiliza el valor inmediatamente siguiente al no definido para rellenar el hueco, y `'ffill'` utiliza el valor anterior. Otra forma de rellenar valores es mediante interpolación con `<objeto>.interpolate()`, que toma los valores anterior y siguiente, y los interpola para rellenar el no definido.

## Valores duplicados y reemplazos

Para encontrar valores duplicados, los objetos **pandas** tienen el método `<objeto>.duplicated()`, que retorna un objeto de las mismas dimensiones con `True` si el valor está duplicado o `False` si no lo está.

Para filtrar los elementos duplicados, se puede hacer a través del método `<objeto>.drop_duplicates()`, que es equivalente a `<objeto>[<objeto>.duplicated() == False]`.

También es conveniente tener la capacidad de reemplazar ciertos valores, como errores de medición. **Series** y **DataFrames** contienen el método `<objeto>.replace(<original>, <nuevo>)`, con el que pueden realizar dicho cambio. El valor `<nuevo>` puede ser un elemento único o una lista. Es posible pasar más de un recambio a la vez, pasando como argumento un diccionario, `<objeto>.replace(<diccionario>)`. En este caso, se reemplazan los elementos definidos como clave por los elementos valor.

## Valores categóricos

Otra transformación fundamental es la conversión de valores continuos en categóricos. Para ello, **pandas** ofrece un nuevo objeto: **Categorical**. Para construirlo, se definen las categorías (contenedores o bins) y los valores, y **pandas** retorna un objeto **Categorical** indicando a qué categoría pertenece cada valor. La forma general es:

```
categorias = pd.cut(<valores>, <bins>)
```

Una vez construido, se puede hacer un recuento de todas las categorías con `pd.value_counts(<categoria>)`.

El argumento `<bins>` puede indicarse como un escalar, y **pandas** automáticamente divide los `<valores>` en segmentos de igual longitud. De forma similar, se puede utilizar la variante `pd.qcut(<valores>, <nbins>)` para que **pandas** divida los `<valores>` en tantas categorías como se determina en `<nbins>`, pero el rango se elige específicamente para repartir los valores de forma equitativa entre las categorías.



### Ejemplo

Queremos categorizar la edad de los clientes en menor de edad (0-17), joven (18-34), adulto (35-64) y maduro (65-99).

```
bins = [0,18,35,65,99]
edades = [2,16,25,18,33,71,44,54]
categorias = pd.cut(edades,bins)
```

El objeto `Categorical` contiene la siguiente información, en la que cada elemento indica el segmento al que el valor en `edades` pertenece.

```
[(0, 18], (0, 18], (18, 35], (0, 18], (18, 35], (65, 99], (35, 65], (35, 65]]
Categories (4, interval[int64]): [(0, 18] < (18, 35] < (35, 65] < (65, 99]]
```

En este caso, el recuento de categorías con `pd.value_counts(categorias)` da:

```
(18, 35]          3
(0, 18]           3
(35, 65]          2
(65, 99]          1
dtype: int64
```



## Broadcasting

Es una operación interna de Python por la cual es posible realizar operaciones entre elementos de dimensiones distintas. En su forma más simple, resulta en virtualmente incrementar el tamaño de la `ndarray` menor hasta igualar el de la otra, para así poder aplicar la operación correctamente. Esto es lo que sucede en el caso básico de aplicar una operación entre un escalar y una `ndarray` (por ejemplo, `np.arange(10) * 2`).

## Casting

Del inglés *cast*, casting es la operación por la cual se fuerza la conversión de un tipo de elemento a otro, como `int(1.1)` o `string(10)`.

## Comparación condicional

Una comparación condicional es una expresión en código que se evalúa hasta dar con un elemento booleano (`True` o `False`). Las comparaciones entre dos objetos emplean comparadores como `==`, `is`, `!=`, `not`, `<`, `<=`, `>`, `>=` para determinar el valor final de la expresión. Se pueden combinar comparaciones con el uso de operadores lógicos como `&` (and), `|` (or) y `~` (not). Las comparaciones condicionales, combinadas con los comandos `if` y `else`, se utilizan para ramificar o bifurcar el código dependiendo del valor de una expresión.

## Datos tabulares

Los datos formulados en dos dimensiones (filas y columnas) son referidos como datos tabulares.

## Debugging

Del inglés *debug*, se utiliza tradicionalmente en informática para hacer referencia al proceso por el cual el programador detecta, investiga y resuelve problemas (bugs) en su código.

## Decorador

En **matplotlib**, los decoradores son funciones que sirven para modificar el aspecto general de una figura y sus ejes.

## Diccionario

Un diccionario es una colección de claves (índices) asociados a valores. Toma la forma `{ clave1 : valor1, clave2 : valor2, ... , claveN : valorN }`. Cada índice puede aparecer un máximo de una vez en el diccionario.

## Función

Una función es un fragmento de código con input y output (ambos pueden ser nulos) que puede invocarse en otro punto del código. Se suelen usar para aglutinar funcionalidad de repetida utilidad (y evitar la repetición de código) y para separar código por funcionalidad.

## Función lambda

Es un tipo especial de función con la particularidad de que es anónima y, por lo tanto, solo se utiliza en el momento de declararla. En Python es muy frecuente expresar con una función lambda argumentos de métodos que aceptan otra función.

## Índice semántico

En **pandas**, el índice semántico hace referencia al índice nominal o etiqueta (`label`) que posee cada uno de los elementos de una serie o frame. Se diferencia del índice posicional en que este último es siempre un número entero que describe la posición en la colección, mientras que el índice semántico puede ser un número o `string`.

## Lenguaje interpretado

Un lenguaje se denomina interpretado cuando su ejecución se hace a partir de la lectura e interpretación concurrente del código escrito por el usuario. La lectura del código y su ejecución por parte de la computadora suceden de forma simultánea. Se distinguen de los lenguajes compilados en que estos requieren un paso previo (compilación), en el que el código se traduce a lenguaje máquina.

## Limpieza de datos

Todo método que sirve para transformar, manipular o modificar los datos para su utilización posterior en técnicas de análisis de datos o aprendizaje automático.

## List comprehension

Se trata de un concepto intrínseco a Python por el cual se pueden formar colecciones a partir de expresiones con loops y condicionales.

## Lista

Una lista es una colección de elementos. A diferencia de las tuplas, una lista es mutable, es decir, su contenido puede ser alterado una vez creada. Toma la forma `[a, b, c]`.

## Loop

Un loop (o ciclo) es un bloque de código que puede evaluarse múltiples veces por repetición. En Python, existen dos tipos de loops: `for` y `while`.



## Método

Véase *función*.

## Namespace

En lenguajes de programación, un namespace (espacio de la nomenclatura) es el ámbito en el cual se reconoce un determinado objeto, función o variable. En Python, hace referencia al lugar en el que se encuentra definida e implementada una clase o función, por lo que para su utilización se requiere importación.

## Operador binario

Un operador binario es un operador que requiere dos operandos para formar un resultado. Ejemplos de operadores binarios son la adición, la resta y la multiplicación.

## Operador ternario

Un operador ternario es una forma corta de una expresión `if else`, combinando la comparación condicional y las expresiones dependientes en una sola expresión. En Python toma la siguiente forma: `<expresión1> if <comparación> else <expresión2>`

## Operador unario

Un operador unario es un operador que requiere un solo operando para formar un resultado. Ejemplos de operadores unarios son el coseno, el seno y el logaritmo.

## Script

En su sentido más amplio, un script es una serie de comandos que la máquina ejecuta de forma secuencial. En el ámbito de los lenguajes de programación de alto nivel (como Python), con frecuencia hace referencia a los archivos que contienen código como scripts.

## Secuencia

Una secuencia es un objeto iterable en Python. Se diferencia de una colección en que una secuencia no tiene por qué estar materializada y, por lo tanto, no es accesible directamente. Por ejemplo, la función `range()` retorna una secuencia que puede iterarse, pero sus elementos son inaccesibles por indexación.

## Set

Un set (o conjunto) es una colección de elementos únicos. Informalmente se puede entender como un diccionario formado solo por claves o índices.

## Slicing

El slicing (o corte) hace referencia a una forma de acceso a colecciones (listas, `Series`, `DataFrame`, `ndarrays`) a través de la cual se obtiene una porción específica de sus elementos.

## Stream

Un stream hace referencia a un flujo de datos. Se suele emplear el término cuando se trata de lectura y escritura de datos, haciendo referencia al objeto que lleva a cabo dichas operaciones.

## Terminal

Un terminal es un programa de ejecución de línea de comandos en el sistema operativo donde el usuario puede interactuar con el sistema a través del teclado. En Windows, es `cmd` (línea de comandos); en MacOS y Linux, es la terminal de comandos.

## Tipo de elemento

El tipo de una variable determina el tipo de elemento al que hace referencia. En lenguajes de tipo dinámico como Python, una variable puede cambiar de tipo a lo largo de la ejecución del programa. Ejemplos de tipos son `int`, `float`, `string`, `bool`.

## Tupla

Una tupla es una colección de elementos inmutables en Python. Toma la forma `(a, b, c)`, donde `a`, `b` y `c` son elementos (pueden ser listas u objetos).

## Variable

En programación, una variable es una referencia a un objeto. El contenido de las variables o el elemento al que hacen referencia pueden cambiar. Se puede entender como un cajón en el cual se puede guardar un elemento.

## Enlaces de interés



### Documentación oficial de Python 3.7

Página oficial en la que se incluye la documentación de todas las versiones de Python. Dispone de material en varios idiomas, entre los que se encuentran el inglés y el francés. Además de documentación sobre la API de Python, se pueden encontrar tutoriales, preguntas frecuentes y novedades.

<https://docs.python.org/3/>

### El libro de los antipatrones en Python

Libro de acceso gratuito con una colección de recomendaciones para escribir código en Python. Incluye aspectos estilísticos, de seguridad, rendimiento, etc. El material está en inglés.

<https://docs.quantifiedcode.com/python-anti-patterns/>

### Librería Scikit-learn

Módulo en Python para el análisis y la minería de datos, con una gran colección de herramientas útiles para el aprendizaje automático. El enlace incluye documentación y ejemplos de cómo trabajar con la librería en cada uno de los dominios. Hace uso de NumPy y matplotlib.

<http://scikit-learn.org/stable/>

### Documentación oficial de NumPy y SciPy

Página oficial mantenida por la comunidad en la que se encuentra documentación sobre dos de las librerías más utilizadas en computación científica, NumPy y SciPy. El material está en inglés y contiene guías de referencia, manuales y guías para desarrolladores.

<https://docs.scipy.org/doc/>

### Documentación oficial de pandas

Página oficial de la librería de análisis de datos por excelencia: pandas. El material está en inglés y proporciona documentación, así como formas de interactuar con la comunidad de desarrolladores y usuarios de pandas, tanto para resolver dudas como para involucrarse en la mejora del módulo.

<https://pandas.pydata.org/index.html>

### Documentación oficial de matplotlib

Página oficial de la librería matplotlib, la más popular para la representación visual de datos. Está en inglés e incluye ejemplos, tutoriales y guías oficiales para utilizar el módulo de forma adecuada.

<https://matplotlib.org/>



- Abadi, M. et al. (noviembre, 2016). TensorFlow: A System for Large-Scale Machine Learning. En K. Keeton, T. Roscoe (Presidencia), *12th USENIX Symposium on Operating Systems Design and Implementation* (pp. 268-283). Simposio llevado a cabo en el congreso de USENIX, Savannah, GA.
- BBVA Open4U (agosto, 2016). *Ventajas e inconvenientes de Python y R para la ciencia de datos*. Recuperado de <https://bbvaopen4u.com/es/actualidad/ventajas-e-inconvenientes-de-python-y-r-para-la-ciencia-de-datos>
- Bergstra, J., Bastien, F., Breuleux, O., Lamblin, P., Pascanu, R., Delalleau, O., Desjardins, G., Warde-Farley, D., Goodfellow, I., Bergeron, A., Bengio, Y. (2011). Theano: Deep Learning on GPUs with Python. *Journal for Machine Learning Research*, 1, 1-48.
- Chollet, F. (2015). Keras. Recuperado de <https://keras.io>
- Jia, Y.; Shelhamer, E.; Donahue, J.; Karayev, S.; Long, J.; Girshick, R.; Guadarrama, S.; Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *Proceedings of the 22nd ACM international conference on Multimedia*. 675-678.
- López, R. (2018). *Libro online de IAAR*. Recuperado de <https://iaarbook.github.io/>
- Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, 2825-2830.

01

```

    _operation ==
    _error_mod.use_x = False
    _error_mod.use_y = True
    _error_mod.use_z = False
    _operation == "MIRROR_Z"
    _error_mod.use_x = False
    _error_mod.use_y = False
    _error_mod.use_z = True

```

selection at the end -add

ob.select= 1

for ob.select=1

E=mc<sup>2</sup>

context.scene.objects.active.y+2a+21

"Selected" + str(modifier)

1+x+y+2a

error\_ob.select = 0

lim h-->0

bpy.context.selected ob

x=0

OPERATOR CLASSES

45 4a 3

1+x+y+2a+21

lim h-->0

45 4a 3

{x-12-y+n...}

Autor  
Carlos Fernández Musoles