



# Aprende Python

**Sergio Delgado Quintero**

**23 de septiembre de 2021**



---

## Core

---

<b>1</b>	<b>Introducción</b>	<b>3</b>
1.1	Hablando con la máquina . . . . .	3
1.2	Algo de historia . . . . .	7
1.3	Python . . . . .	13
<b>2</b>	<b>Entornos de desarrollo</b>	<b>21</b>
2.1	Thonny . . . . .	22
2.2	Contexto real . . . . .	27
<b>3</b>	<b>Tipos de datos</b>	<b>35</b>
3.1	Datos . . . . .	36
3.2	Números . . . . .	47
3.3	Cadenas de texto . . . . .	61
<b>4</b>	<b>Control de flujo</b>	<b>85</b>
4.1	Condicionales . . . . .	86
4.2	Bucles . . . . .	98
<b>5</b>	<b>Estructuras de datos</b>	<b>111</b>
5.1	Listas . . . . .	112
5.2	Tuplas . . . . .	139
5.3	Diccionarios . . . . .	146
5.4	Conjuntos . . . . .	164
5.5	Ficheros . . . . .	171
<b>6</b>	<b>Modularidad</b>	<b>179</b>
6.1	Funciones . . . . .	180
6.2	Objetos y Clases . . . . .	220
6.3	Excepciones . . . . .	248
6.4	Módulos . . . . .	254

---

<b>7 Procesamiento de texto</b>	<b>263</b>
7.1 string . . . . .	264
<b>8 Ciencia de datos</b>	<b>269</b>
8.1 jupyter . . . . .	270
8.2 numpy . . . . .	290
8.3 pandas . . . . .	332
8.4 matplotlib . . . . .	378



Curso gratuito para aprender el lenguaje de programación **Python** con un enfoque **práctico**, incluyendo **ejercicios** y cobertura para distintos **niveles de conocimiento**.<sup>1</sup>

---

**Consejo:** «Programming is not about typing, it's about thinking.» – Rich Hickey

---

---

<sup>1</sup> En la foto de portada aparecen los Monty Python. Fuente: [noticiascyl](#)



# CAPÍTULO 1

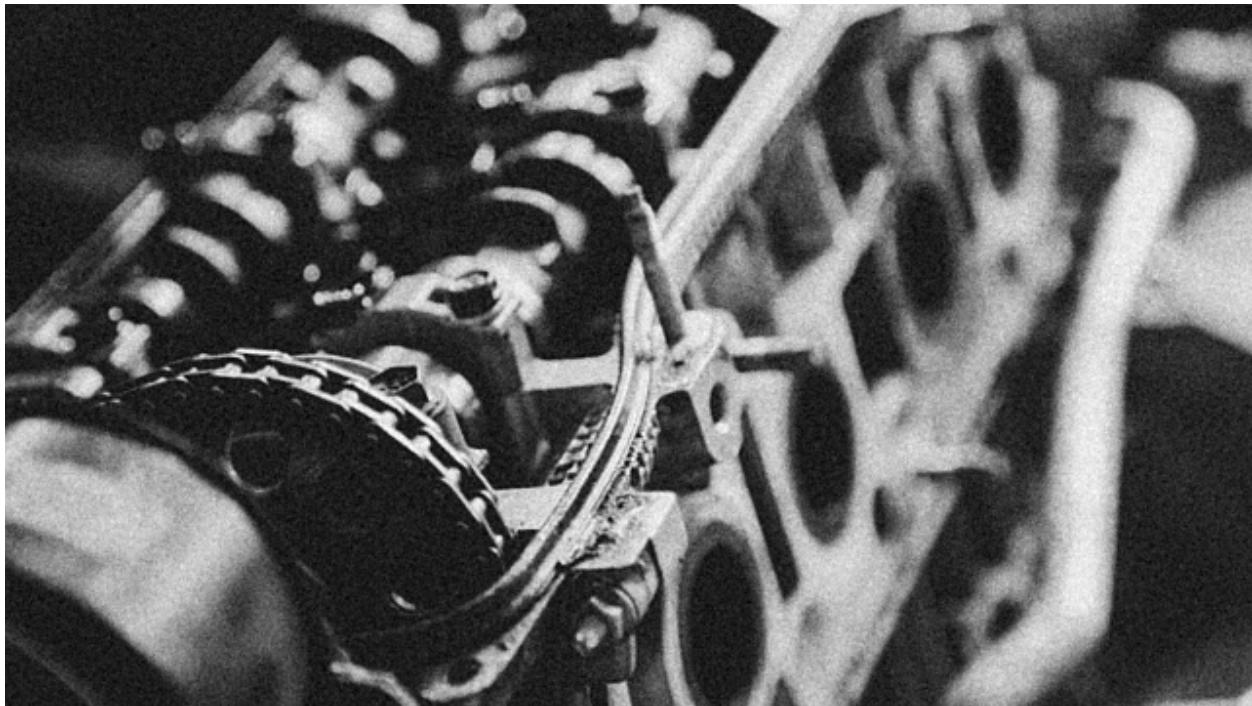
---

## Introducción

---

Este capítulo es una introducción a la programación para conocer, desde un enfoque sencillo pero aclaratorio, los mecanismos que hay detrás de ello.

### 1.1 Hablando con la máquina



Los ordenadores son dispositivos complejos pero están diseñados para hacer una cosa bien: **ejecutar aquello que se les indica**. La cuestión es cómo indicar a un ordenador lo que queremos que execute. Esas indicaciones se llaman técnicamente **instrucciones** y se expresan en un lenguaje. Podríamos decir que **programar** consiste en escribir instrucciones para que sean ejecutadas por un ordenador. El lenguaje que utilizamos para ello se denomina **lenguaje de programación**.<sup>1</sup>

### 1.1.1 Código máquina

Pero aún seguimos con el problema de cómo hacer que un ordenador (o máquina) entienda el lenguaje de programación. A priori podríamos decir que un ordenador sólo entiende un lenguaje muy «simple» denominado **código máquina**. En este lenguaje se utilizan únicamente los símbolos **0** y **1** en representación de los *niveles de tensión* alto y bajo, que al fin y al cabo, son los estados que puede manejar un circuito digital. Hablamos de **sistema binario**. Si tuviéramos que escribir programas de ordenador en este formato sería una tarea ardua, pero afortunadamente se han ido creando con el tiempo lenguajes de programación intermedios que, posteriormente, son convertidos a código máquina.

Si intentamos visualizar un programa en código máquina, únicamente obtendríamos una secuencia de ceros y unos:

```
00001000 00000010 01111011 10101100 10010111 11011001 01000000 01100010  
00110100 00010111 01101111 10111001 01010110 00110001 00101010 00011111  
10000011 11001101 11110101 01001110 01010010 10100001 01101010 00001111  
11101010 00100111 11000100 01110101 11011011 00010110 10011111 01010110
```

### 1.1.2 Ensamblador

El primer lenguaje de programación que encontramos en esta «escalada» es **ensamblador**. Veamos un ejemplo de código en ensamblador del típico programa que se escribe por primera vez, el «Hello, World»:

```
SYS_SALIDA equ 1

section .data
    msg db "Hello, World",0x0a
    len equ $ - msg ;longitud de msg

section .text
global _start ;para el linker
_start: ;marca la entrada
    mov eax, 4 ;llamada al sistema (sys_write)
```

(continué en la próxima página)

<sup>1</sup> Foto original por Garett Mizunaka en Unsplash.

(proviene de la página anterior)

```

mov ebx, 1 ;descripción de archivo (stdout)
mov ecx, msg ;msg a escribir
mov edx, len ;longitud del mensaje
int 0x80 ;llama al sistema de interrupciones

fin: mov eax, SYS_SALIDA ;llamada al sistema (sys_exit)
    int 0x80

```

Aunque resulte difícil de creer, lo «único» que hace este programa es mostrar en la pantalla de nuestro ordenador la frase «Hello, World», pero además teniendo en cuenta que sólo funcionará para una arquitectura x86.

### 1.1.3 C

Aunque el lenguaje ensamblador nos facilita un poco la tarea de desarrollar programas, sigue siendo bastante complicado ya que las instrucciones son muy específicas y no proporcionan una semántica entendible. Uno de los lenguajes que vino a suplir – en parte – estos obstáculos fue **C**. Considerado para muchas personas como un referente en cuanto a los lenguajes de programación, permite hacer uso de instrucciones más claras y potentes. El mismo ejemplo anterior del programa «Hello, World» se escribiría así en lenguaje *C*:

```

#include <stdio.h>

int main() {
    printf("Hello, World");
    return 0;
}

```

### 1.1.4 Python

Si seguimos «subiendo» en esta lista de lenguajes de programación, podemos llegar hasta **Python**. Se dice que es un lenguaje de *más alto nivel* en el sentido de que sus instrucciones son más entendibles por un humano. Veamos cómo se escribiría el programa «Hello, World» en el lenguaje de programación Python:

```
print('Hello, World')
```

¡Pues así de fácil! Hemos pasado de *código máquina* (ceros y unos) a *código Python* en el que se puede entender perfectamente lo que estamos indicando al ordenador. La pregunta que surge es: ¿cómo entiende una máquina lo que tiene que hacer si le pasamos un programa hecho en Python (o cualquier otro lenguaje de alto nivel)? La respuesta es un **compilador**.

### 1.1.5 Compiladores

Los compiladores son programas que convierten un lenguaje «cualquiera» en *código máquina*. Se pueden ver como traductores, permitiendo a la máquina interpretar lo que queremos hacer.

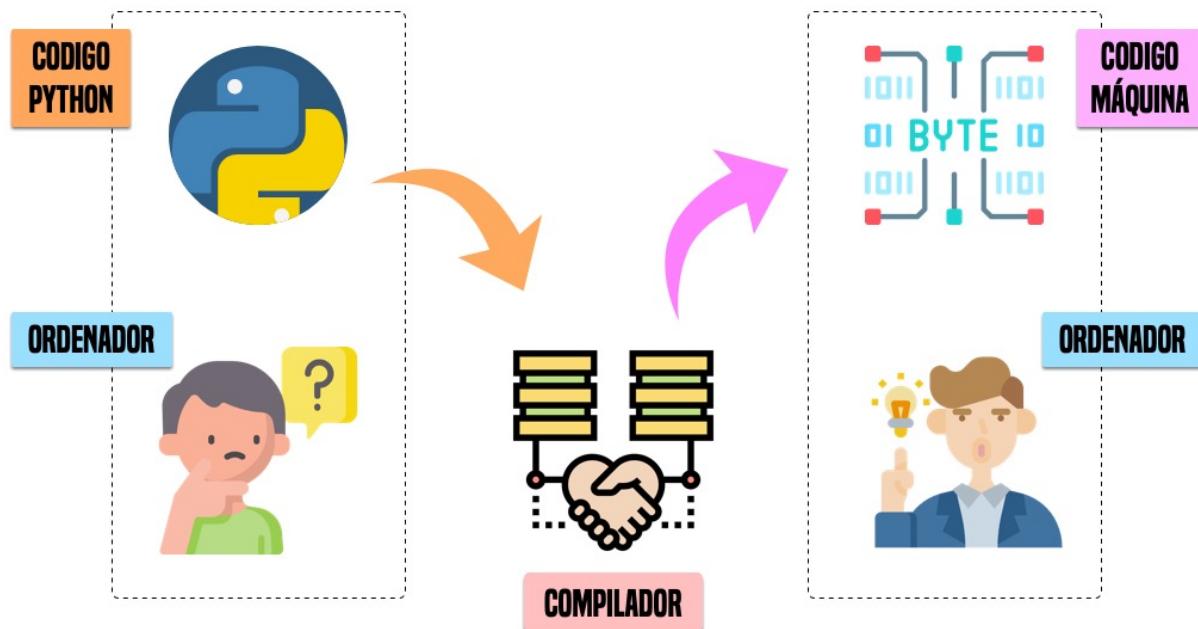


Figura 1: Esquema de funcionamiento de un compilador<sup>2</sup>

---

**Nota:** Para ser más exactos, en Python hablamos de un **intérprete** en vez de un compilador, pero a los efectos es prácticamente lo mismo. La diferencia está en que el intérprete realiza la «compilación» (*interpretación*) y la «ejecución» de una vez, mientras que el compilador genera un formato «ejecutable» (*código objeto*) que se ejecuta en otra fase posterior.

---

<sup>2</sup> Iconos originales por Favicon.

## 1.2 Algo de historia



La historia de la programación está relacionada directamente con la aparición de los computadores, que ya desde el siglo XV tuvo sus inicios con la construcción de una máquina que realizaba operaciones básicas y raíces cuadradas ([Gottfried Wilhem von Leibniz](#)); aunque en realidad la primera gran influencia hacia la creación de los computadores fue la máquina diferencial para el cálculo de polinomios, proyecto no concluido de [Charles Babbage](#) (1793-1871) con el apoyo de [Lady Ada Countess of Lovelace](#) (1815-1852), primera persona que incursionó en la programación y de quien proviene el nombre del lenguaje de programación [ADA](#) creado por el DoD (Departamento de defensa de Estados Unidos) en la década de 1970.<sup>1</sup>

### 1.2.1 Hitos de la computación

La siguiente tabla es un resumen de los principales hitos en la historia de la computación:

Tabla 1: Hitos en la computación

Personaje	Aporte	Año
Gottfried Leibniz	Máquinas de <b>operaciones básicas</b>	XV
Charles Babbage	<b>Máquina diferencial</b> para el cálculo de polinomios	XVII

continué en la próxima página

<sup>1</sup> Foto original por Dario Veronesi en Unsplash.

Tabla 1 – proviene de la página anterior

Personaje	Aporte	Año
Ada Lovelace	Matemática, informática y escritora británica. <b>Primera programadora</b> de la historia por el desarrollo de algoritmos para la máquina analítica de Babbage	XVII
George Boole	Contribuyó al <b>algebra binaria</b> y a los sistemas de circuitos de computadora (álgebra booleana)	1854
Herman Hollerit	Creador de un <b>sistema para automatizar</b> la pesada tarea del censo	1890
Alan Turing	<b>Máquina de Turing</b> - una máquina capaz de resolver problemas - Aportes de Lógica Matemática - Computadora con tubos de vacío	1936
John Atanasoff	Primera <b>computadora digital</b> electrónica patentada: Atanasoff Berry Computer (ABC)	1942
Howard Aiken	En colaboración con IBM desarrolló el <b>Mark I</b> , una computadora electromecánica de 16 metros de largo y más de dos de alto que podía realizar las cuatro operaciones básicas y trabajar con información almacenada en forma de tablas	1944
Grace Hopper	Primera <b>programadora</b> que utilizó el Mark I	1945
John W. Mauchly	Junto a John Presper Eckert desarrolló una <b>computadora electrónica</b> completamente operacional a gran escala llamada Electronic Numerical Integrator And Computer (ENIAC)	1946
John Von Neumann	Propuso guardar en <b>memoria</b> no solo la <b>información</b> , sino también los <b>programas</b> , acelerando los procesos	1946

Luego los avances en las ciencias informáticas han sido muy acelerados, se reemplazaron los **tubos de vacío** por **transistores** en 1958 y en el mismo año, se sustituyeron por **circuitos integrados**, y en 1961 se miniaturizaron en **chips de silicio**. En 1971 apareció el primer microprocesador de Intel; y en 1973 el primer sistema operativo CP/M. El primer computador personal es comercializado por IBM en el año 1980.

---

<sup>2</sup> Fuente: Meatze.



Figura 2: Ada Lovelace: primera programadora de la historia<sup>2</sup>

### 1.2.2 De los computadores a la programación

De acuerdo a este breve viaje por la historia, la programación está vinculada a la aparición de los computadores, y los lenguajes tuvieron también su evolución. Inicialmente, como ya hemos visto, se programaba en **código binario**, es decir en cadenas de 0s y 1s, que es el lenguaje que entiende directamente el computador, tarea extremadamente difícil; luego se creó el **lenguaje ensamblador**, que aunque era lo mismo que programar en binario, al estar en letras era más fácil de recordar. Posteriormente aparecieron **lenguajes de alto nivel**, que en general, utilizan palabras en inglés, para dar las órdenes a seguir, para lo cual utilizan un proceso intermedio entre el lenguaje máquina y el nuevo código llamado código fuente, este proceso puede ser un compilador o un intérprete.

Un **compilador** lee todas las instrucciones y genera un resultado; un **intérprete** ejecuta y genera resultados línea a línea. En cualquier caso han aparecido nuevos lenguajes de programación, unos denominados estructurados y en la actualidad en cambio los lenguajes orientados a objetos y los lenguajes orientados a eventos.<sup>3</sup>

### 1.2.3 Cronología de lenguajes de programación

Desde la década de 1950 se han sucedido multitud de lenguajes de programación que cada vez incorporan más funcionalidades destinadas a cubrir las necesidades del desarrollo de aplicaciones. A continuación se muestra una tabla con la historia de los lenguajes de programación más destacados:

El **número** actual de lenguajes de programación depende de lo que se considere un *lenguaje de programación* y a *quién se pregunte*. Según [TIOBE](#) más de 250; según [Wikipedia](#) más de 700, según [Language List](#) más de 2500; y para una cifra muy alta podemos considerar a [Online Historical Encyclopaedia of Programming Languages](#) que se acerca a los **9000**.

### 1.2.4 Creadores de lenguajes de programación

El avance de la computación está íntimamente relacionado con el desarrollo de los lenguajes de programación. Sus creadores y creadoras juegan un *rol fundamental* en la historia tecnológica. Veamos algunas de estas personas:<sup>4</sup>

Tabla 2: Creadores de lenguajes de programación

Personaje	Aporte
Alan Cooper	Desarrollador de <b>Visual Basic</b>
Alan Kay	Pionero en programación orientada a objetos. Creador de <b>Smalltalk</b>

continué en la próxima página

<sup>3</sup> Fuente: Universidad Técnica del Norte.

<sup>4</sup> Fuente: Wikipedia.

Tabla 2 – proviene de la página anterior

Personaje	Aporte
Anders Hejlsberg	Desarrollador de <b>Turbo Pascal, Delphi y C#</b>
Bertrand Meyer	Inventor de <b>Eiffel</b>
Bill Joy	Inventor de <b>vi</b> . Autor de <b>BSD Unix</b> . Creador de <b>SunOS</b> , el cual se convirtió en <b>Solaris</b>
Bjarne Stroustrup	Desarrollador de <b>C++</b>
Brian Kernighan	Coautor del primer libro de programación en lenguaje <b>C</b> con Dennis Ritchie y coautor de los lenguajes de programación <b>AWK</b> y <b>AMPL</b>
Dennis Ritchie	Inventor de C. Sistema Operativo <b>Unix</b>
Edsger W. Dijkstra	Desarrolló las bases para la programación estructurada
Grace Hopper	Desarrolladora de <b>Flow-Matic</b> , influenciando el lenguaje <b>COBOL</b>
Guido van Rossum	Creador de <b>Python</b>
James Gosling	Desarrollador de <b>Oak</b> . Precursor de <b>Java</b>
Joe Armstrong	Creador de <b>Erlang</b>
John Backus	Inventor de <b>Fortran</b>
John McCarthy	Inventor de <b>LISP</b>
John von Neumann	Creador del concepto de sistema operativo
Ken Thompson	Inventor de B. Desarrollador de <b>Go</b> . Coautor del sistema operativo Unix
Kenneth E. Iverson	Desarrollador de <b>APL</b> . Co-desarrollador de <b>J</b> junto a Roger Hui
Larry Wall	Creador de <b>Perl</b> y Perl 6
Martin Odersky	Creador de <b>Scala</b> . Previamente contribuyó en el diseño de Java
Mitchel Resnick	Creador del lenguaje visual <b>Scratch</b>
Nathaniel Rochester	Inventor del primer lenguaje en <b>ensamblador</b> simbólico (IBM 701)
Niklaus Wirth	Inventor de <b>Pascal, Modula y Oberon</b>
Robin Milner	Inventor de <b>ML</b> . Compartió crédito en el método Hindley–Milner de inferencia de tipo polimórfica
Seymour Papert	Pionero de la inteligencia artificial. Inventor del lenguaje de programación <b>Logo</b> en 1968
Stephen Wolfram	Creador de <b>Mathematica</b>
Yukihiro Matsumoto	Creador de <b>Ruby</b>

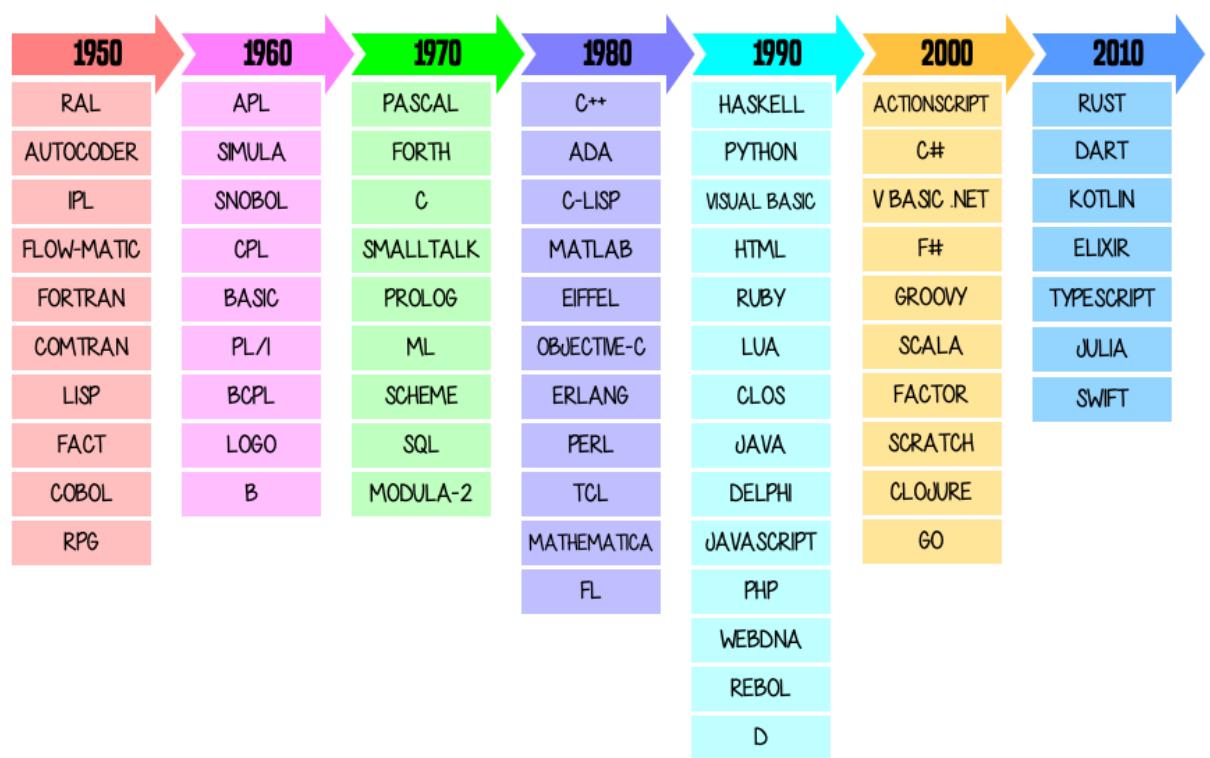


Figura 3: Cronología de los lenguajes de programación más destacados

## 1.3 Python



Python es un lenguaje de programación de *alto nivel* creado a finales de los 80/principios de los 90 por Guido van Rossum, holandés que trabajaba por aquella época en el *Centro para las Matemáticas y la Informática* de los Países Bajos. Sus instrucciones están muy cercanas al **lenguaje natural** en inglés y se hace hincapié en la **legibilidad** del código. Toma su nombre de los **Monty Python**, grupo humorista de los 60 que gustaban mucho a Guido. Python fue creado como sucesor del lenguaje ABC.<sup>1</sup>

### 1.3.1 Características del lenguaje

A partir de su definición de la Wikipedia:

- Python es un lenguaje de programación **interpretado** cuya filosofía hace hincapié en una sintaxis que favorezca un **código legible**.
- Se trata de un lenguaje de programación **multiparadigma**, ya que soporta **orientación a objetos**, **programación imperativa** y, en menor medida, programación funcional. Usa **tipado dinámico** y es **multiplataforma**.
- Añadiría, como característica destacada, que se trata de un lenguaje de **propósito general**.

---

<sup>1</sup> Foto original por Markéta Marcellová en Unsplash.

## Ventajas

- Libre y gratuito (OpenSource).
- Fácil de leer, parecido a pseudocódigo.
- Aprendizaje relativamente fácil y rápido: claro, intuitivo....
- Alto nivel.
- Alta Productividad: simple y rápido.
- Tiende a producir un buen código: orden, limpieza, elegancia, flexibilidad, ...
- Multiplataforma. Portable.
- Multiparadigma: programación imperativa, orientada a objetos, funcional, ...
- Interactivo, modular, dinámico.
- Librerías extensivas («pilas incluídas»).
- Gran cantidad de librerías de terceros.
- Extensible (C++, C, ...) y «embebible».
- Gran comunidad, amplio soporte.
- Interpretado.
- Fuertemente tipado, tipado dinámico.
- Hay diferentes implementaciones: CPython, PyPy, Jython, IronPython, MicroPython,  
...

## Desventajas

- Interpretado (velocidad de ejecución, multithread vs GIL, etc.).
- Consumo de memoria.
- Errores durante la ejecución.
- Dos versiones mayores no del todo compatibles (v2 vs v3).
- Desarrollo móvil.
- Documentación a veces dispersa e incompleta.
- Varios módulos para la misma funcionalidad.
- Librerías de terceros no siempre del todo maduras.

### 1.3.2 Uso de Python

Al ser un lenguaje de propósito general, podemos encontrar aplicaciones prácticamente en todos los campos científico-tecnológicos:

- Análisis de datos.
- Aplicaciones de escritorio.
- Bases de datos relacionales / NoSQL
- Buenas prácticas de programación / Patrones de diseño.
- Concurrencia.
- Criptomonedas / Blockchain.
- Desarrollo de aplicaciones multimedia.
- Desarrollo de juegos.
- Desarrollo en dispositivos embebidos.
- Desarrollo móvil.
- Desarrollo web.
- DevOps / Administración de sistemas / Scripts de automatización.
- Gráficos por ordenador.
- Inteligencia artificial.
- Internet de las cosas.
- Machine Learning.
- Programación de parsers / scrapers / crawlers.
- Programación de redes.
- Propósitos educativos.
- Prototipado de software.
- Seguridad.
- Tests automatizados.

De igual modo son muchas las empresas, instituciones y organismos que utilizan Python en su día a día para mejorar sus sistemas de información. Veamos algunas de las más relevantes:

Existen rankings y estudios de mercado que sitúan a Python como uno de los lenguajes más *usados* y la vez, más *amados* dentro del mundo del desarrollo de software. En el momento de la escritura de este documento, la última actualización del [Índice TIOBE](#) es de *agosto de 2020* en la que Python ocupa el **tercer lugar de los lenguajes de programación**



Figura 4: Grandes empresas y organismos que usan Python

más usados, sólo por detrás de *C* y *Java*. Igualmente en la encuesta a desarrolladores de Stack Overflow hecha en 2020, Python ocupa el cuarto puesto de los lenguajes de programación más usados, sólo por detrás de *Javascript*, *HTML/CSS* y *SQL*.

### 1.3.3 Python2 vs Python3

En el momento de la escritura de este material, se muestra a continuación la evolución de las versiones mayores de Python a lo largo de la historia:<sup>3</sup>

Versión	Fecha de lanzamiento
Python 1.0	Enero 1994
Python 1.5	Diciembre 1997
Python 1.6	Septiembre 2000
Python 2.0	Octubre 2000
Python 2.1	Abril 2001
Python 2.2	Diciembre 2001
Python 2.3	Julio 2003
Python 2.4	Noviembre 2004
Python 2.5	Septiembre 2006
Python 2.6	Octubre 2008
Python 2.7	Julio 2010
Python 3.0	Diciembre 2008

continué en la próxima página

<sup>3</sup> Fuente: [python.org](https://python.org).

Tabla 3 – proviene de la página anterior

Versión	Fecha de lanzamiento
Python 3.1	Junio 2009
Python 3.2	Febrero 2011
Python 3.3	Septiembre 2012
Python 3.4	Marzo 2014
Python 3.5	Septiembre 2015
Python 3.6	Diciembre 2016
Python 3.7	Junio 2018
Python 3.8	Octubre 2019
Python 3.9	Octubre 2020

El cambio de **Python 2** a **Python 3** fue bastante «traumático» ya que se **perdió la compatibilidad** en muchas de las estructuras del lenguaje. Los «*core-developers*»<sup>4</sup>, con *Guido van Rossum* a la cabeza, vieron la necesidad de aplicar estas modificaciones en beneficio del rendimiento y expresividad del lenguaje de programación. Este cambio implicaba que el código escrito en Python 2 no funcionaría (de manera inmediata) en Python 3.

El pasado **1 de enero de 2020** finalizó oficialmente el **soporte a la versión 2.7** del lenguaje de programación Python. Es por ello que se recomienda lo siguiente:

- Si aún desarrollas aplicaciones escritas en Python 2, deberías migrar a Python 3.
- Si vas a desarrollar una nueva aplicación, deberías hacerlo directamente en Python 3.

---

**Importante:** Únete a **Python 3** y aprovecha todas sus ventajas.

---

### 1.3.4 Zen de Python

Existen una serie de *reglas* «filosóficas» que indican una manera de hacer y de pensar dentro del mundo **pitónico**<sup>2</sup> creadas por **Tim Peters**, llamadas el **Zen de Python** y que se pueden aplicar incluso más allá de la programación:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
```

(continué en la próxima página)

<sup>4</sup> Término que se refiere a los/las desarrolladores/as principales del lenguaje de programación.

<sup>2</sup> Dícese de algo/alguien que sigue las convenciones de Python.

(provien de la página anterior)

Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one-- and preferably only one --obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than \*right\* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!

En su [traducción de la Wikipedia](#):

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.

- Los espacios de nombres son una gran idea, ¡tengamos más de esos!



# CAPÍTULO 2

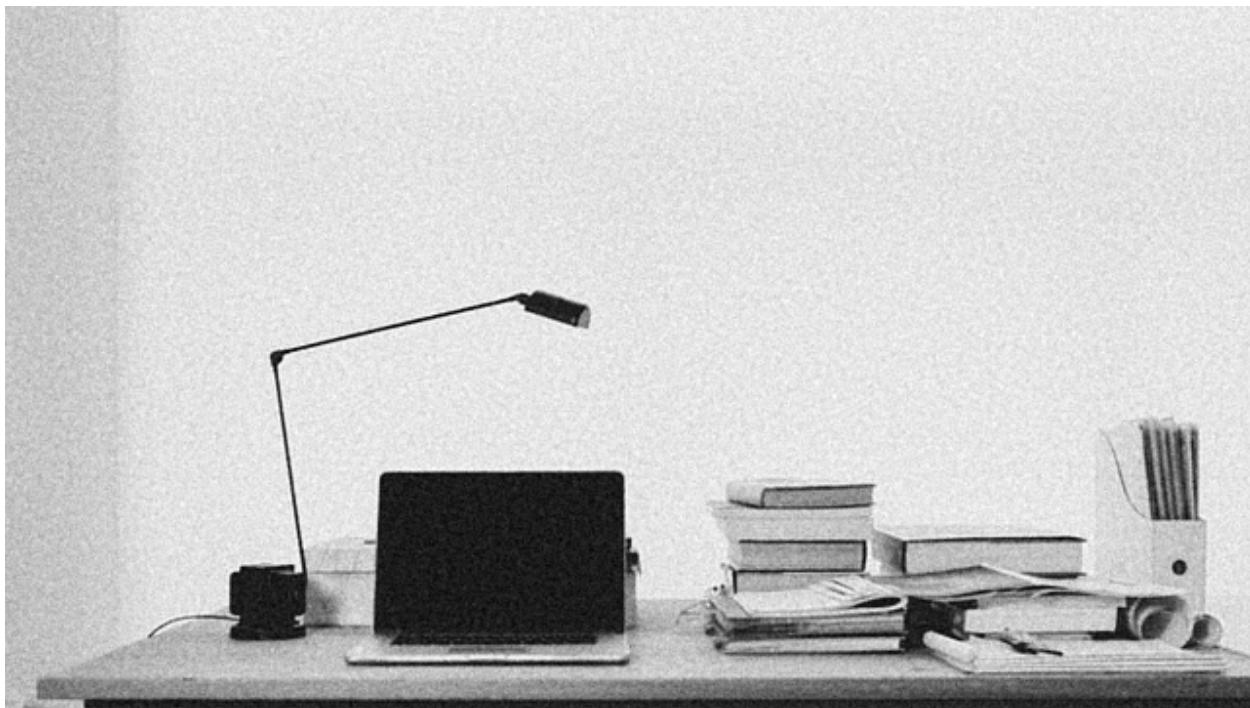
---

## Entornos de desarrollo

---

Para poder utilizar Python debemos preparar nuestra máquina con las herramientas necesarias. Este capítulo trata sobre la instalación y configuración de los elementos adecuados para el desarrollo con el lenguaje de programación Python.

## 2.1 Thonny



Cuando vamos a trabajar con Python debemos tener instalado, como mínimo, un *intérprete* del lenguaje (para otros lenguajes sería un *compilador*). El **intérprete** nos permitirá *ejecutar* nuestro código para obtener los resultados deseados. La idea del intérprete es lanzar instrucciones «sueltas» para probar determinados aspectos.

Pero normalmente queremos ir un poco más allá y poder escribir programas algo más largos, por lo que también necesitaremos un **editor**. Un editor es un programa que nos permite crear ficheros de código (en nuestro caso con extensión `*.py`), que luego son ejecutados por el intérprete.

Hay otra herramienta interesante dentro del entorno de desarrollo que sería el **depurador**. Lo podemos encontrar habitualmente en la bibliografía por su nombre inglés *debugger*. Es el módulo que nos permite ejecutar paso a paso nuestro código y visualizar qué está ocurriendo en cada momento. Se suele usar normalmente para encontrar fallos (*bugs*) en nuestros programas y poder solucionarlos (*debug/fix*).

Thonny es un programa muy interesante para empezar a aprender Python ya que engloba estas tres herramientas: intérprete, editor y depurador.<sup>1</sup>

---

<sup>1</sup> Foto original de portada por freddie marriage en Unsplash.

## 2.1.1 Instalación

Para instalar Thonny debemos acceder a su [web](#) y descargar la aplicación para nuestro sistema operativo. La ventaja es que está disponible tanto para **Windows**, **Mac** y **Linux**. Una vez descargado el fichero lo ejecutamos y seguimos su instalación paso por paso.

Una vez terminada la instalación ya podemos lanzar la aplicación que se verá parecida a la siguiente imagen:

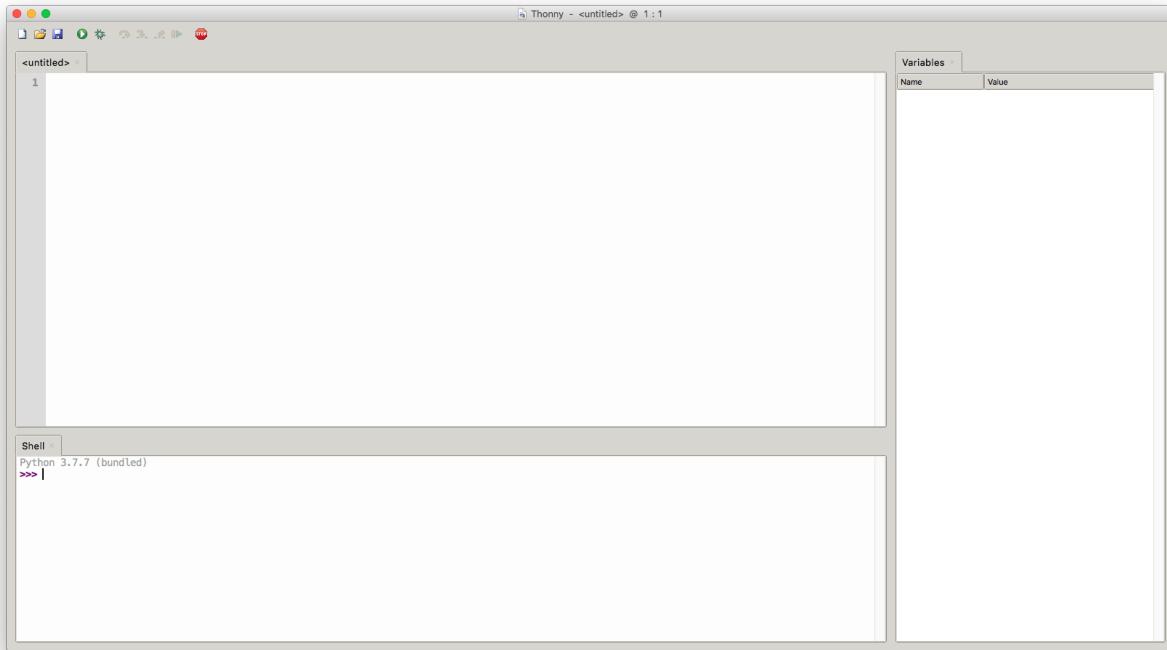


Figura 1: Aspecto de Thonny al arrancarlo

---

**Nota:** Es posible que el aspecto del programa varíe ligeramente según el sistema operativo, configuración de escritorio, versión utilizada o idioma (*en mi caso está en inglés*), pero a efectos de funcionamiento no hay diferencia.

---

Podemos observar que la pantalla está dividida en 3 paneles:

- *Panel principal* que contiene el **editor** e incluye la etiqueta **<untitled>** donde escribiremos nuestro *código fuente* Python.
- *Panel inferior* con la etiqueta **Shell** que contiene el **intérprete** de Python. En el momento de la escritura del presente documento, Thonny incluye la versión de Python 3.7.7.
- *Panel derecho* que contiene el **depurador**. Más concretamente se trata de la ventana de variables donde podemos *inspeccionar* el valor de las mismas.

## Versiones de Python

Existen múltiples versiones de Python. Desde el lanzamiento de la versión 1.0 en 1994 se han ido liberando versiones, cada vez, con nuevas características que aportan riqueza al lenguaje:

Versión	Fecha de lanzamiento
Python 1.0	Enero 1994
Python 1.5	Diciembre 1997
Python 1.6	Septiembre 2000
Python 2.0	Octubre 2000
Python 2.1	Abril 2001
Python 2.2	Diciembre 2001
Python 2.3	Julio 2003
Python 2.4	Noviembre 2004
Python 2.5	Septiembre 2006
Python 2.6	Octubre 2008
Python 2.7	Julio 2010
Python 3.0	Diciembre 2008
Python 3.1	Junio 2009
Python 3.2	Febrero 2011
Python 3.3	Septiembre 2012
Python 3.4	Marzo 2014
Python 3.5	Septiembre 2015
Python 3.6	Diciembre 2016
Python 3.7	Junio 2018
Python 3.8	Octubre 2019
Python 3.9	Octubre 2020

---

**Nota:** Para ser exactos, esta tabla (y en general todo este manual) versa sobre una implementación concreta de Python denominada **C**Python, pero existen [otras implementaciones alternativas de Python](#). A los efectos de aprendizaje del lenguaje podemos referirnos a *Python* (aunque realmente estaríamos hablando de *C*Python).

---

### 2.1.2 Probando el intérprete

El intérprete de Python (por lo general) se identifica claramente porque posee un **prompt**<sup>2</sup> con tres angulos hacia la derecha **>>>**. En Thonny lo podemos encontrar en el panel inferior, pero se debe tener en cuenta que el intérprete de Python es una herramienta autocontenido y que la podemos ejecutar desde el símbolo del sistema o la terminal:

---

<sup>2</sup> Término inglés que se refiere al símbolo que precede la línea de comandos.

Lista 1: Invocando el intérprete de Python 3.7 desde una terminal en MacOS

```
$ python3.7
Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Para hacer una prueba inicial del intérprete vamos a retomar el primer programa que se suele hacer. Es el llamado «*Hello, World*». Para ello escribimos lo siguiente en el intérprete y pulsamos la tecla ENTER:

```
>>> print('Hello, World')
Hello, World
```

Lo que hemos hecho es indicarle a Python que ejecute como **entrada** la instrucción `print('Hello, World')`. La **salida** es el texto `Hello, World` que lo vemos en la siguiente línea (*ya sin el prompt >>>*).

### 2.1.3 Probando el editor

Ahora vamos a realizar la misma operación, pero en vez de ejecutar la instrucción directamente en el intérprete, vamos a crear un fichero y guardarlo con la sentencia que nos interesa. Para ello escribimos `print('Hello, World')` en el panel de edición (*superior*) y luego guardamos el archivo con el nombre `helloworld.py`<sup>3</sup>:

---

**Importante:** Los ficheros que contienen programas hechos en Python siempre deben tener la extensión `.py`

---

Ahora ya podemos *ejecutar* nuestro fichero `helloworld.py`. Para ello pulsamos el botón verde con triángulo blanco (en la barra de herramientas) o bien damos a la tecla F5. Veremos que en el panel de *Shell* nos aparece la salida esperada. Lo que está pasando «entre bambalinas» es que el intérprete de Python está recibiendo como entrada el fichero que hemos creado; lo ejecuta y devuelve la salida para que Thonny nos lo muestre en el panel correspondiente.

---

<sup>3</sup> La carpeta donde se guarden los archivos de código no es crítico para su ejecución, pero sí es importante mantener un orden y una organización para tener localizados nuestros ficheros y proyectos.

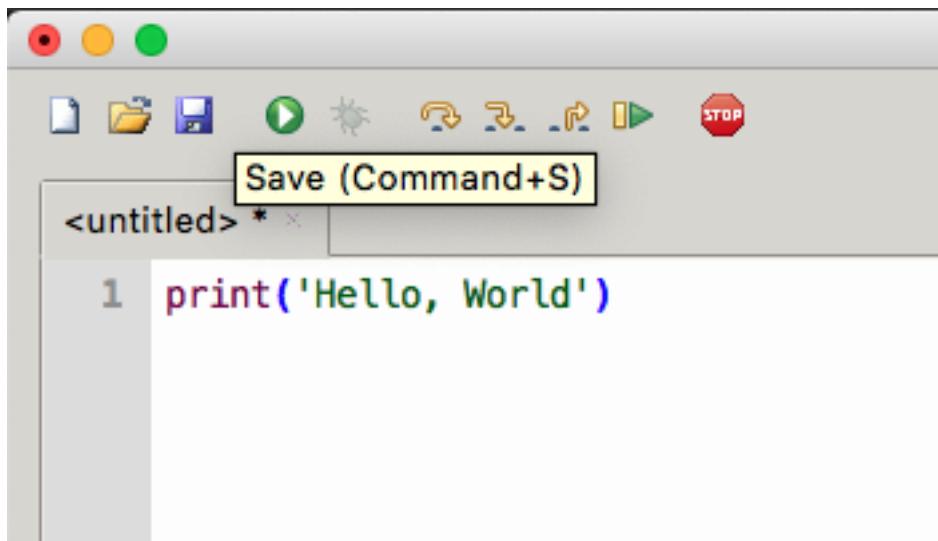


Figura 2: Guardando nuestro primer programa en Python

### 2.1.4 Probando el depurador

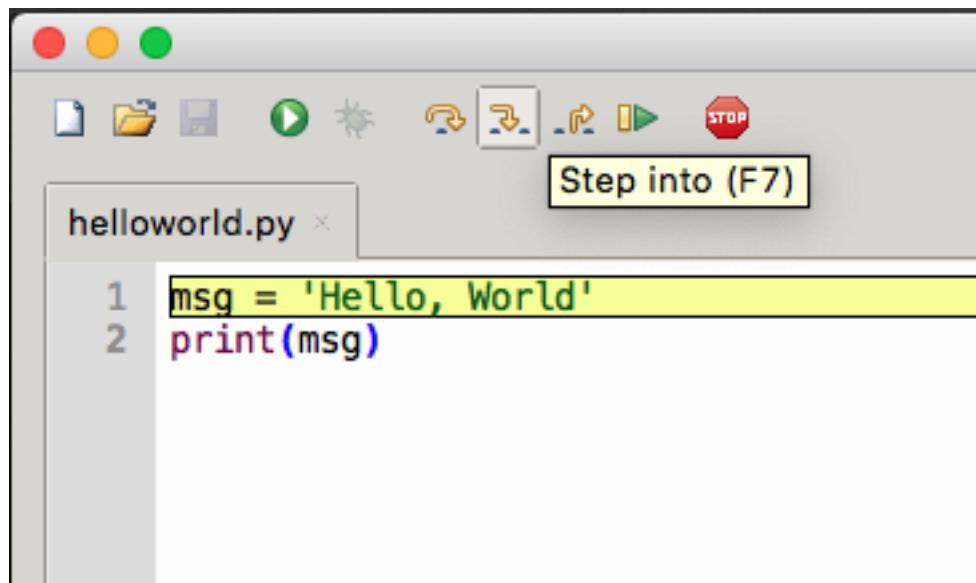
Nos falta por probar el depurador o «debugger». Aunque su funcionamiento va mucho más allá, de momento nos vamos a quedar en la posibilidad de inspeccionar las variables de nuestro programa. Desafortunadamente `helloworld.py` es muy simple y ni siquiera contiene variables, pero podemos hacer una pequeña modificación al programa para poder incorporarlas:

```
1 msg = 'Hello, World'  
2 print(msg)
```

Aunque ya lo veremos en profundidad, lo que hemos hecho es añadir una variable `msg` en la *línea 1* para luego utilizarla al mostrar por pantalla su contenido. Si ahora volvemos a ejecutar nuestro programa veremos que en el panel de variables nos aparece la siguiente información:

Name	Value
msg	'Hello, World'

También existe la posibilidad, a través del depurador, de ir ejecutando nuestro programa **paso a paso**. Para ello basta con pulsar en el botón que tiene un *insecto*. Ahí comienza la sesión de depuración y podemos avanzar instrucción por instrucción usando la tecla F7:



A screenshot of the Thonny Python IDE. The window title is "helloworld.py". The code editor contains the following Python script:

```
1 msg = 'Hello, World'
2 print(msg)
```

The first line of code, "msg = 'Hello, World'", is highlighted with a yellow background. In the toolbar at the top, the "Step into (F7)" button is highlighted with a yellow border.

Figura 3: Depurando nuestro primer programa en Python

## 2.2 Contexto real



Hemos visto que *Thonny* es una herramienta especialmente diseñada para el aprendizaje de Python, integrando diferentes módulos que facilitan su gestión. Si bien lo podemos utilizar para un desarrollo más «serio», se suele recurrir a un flujo de trabajo algo diferente en

contextos más reales.<sup>1</sup>

### 2.2.1 Python

La forma más habitual de instalar Python (junto con sus librerías) es descargarlo e instalarlo desde su página oficial:

- Versiones de Python para Windows
- Versiones de Python para Mac
- Versiones de Python para Linux

### Anaconda

Otra de las alternativas para disponer de Python es nuestro sistema y que además es muy utilizada, es **Anaconda**. Se trata de un *conjunto de herramientas*, orientadas en principio a la *ciencia de datos*, pero que podemos utilizarlas para desarrollo general en Python (junto con otras librerías adicionales). Existen versiones de pago, pero la distribución *Individual Edition* es «open-source» y gratuita. Se puede descargar desde los siguientes enlaces:

- Anaconda 64-bits para Windows
- Anaconda 64-bits para Mac
- Anaconda 64-bits para Linux

### Ver también:

Miniconda es un instalador mínimo que trae por defecto Python y un pequeño número de paquetes útiles.

### 2.2.2 Gestión de paquetes

La instalación limpia<sup>2</sup> de Python ya ofrece de por sí muchos paquetes y módulos que vienen por defecto. Es lo que se llama la *librería estándar*. Pero una de las características más destacables de Python es su inmenso «ecosistema» de paquetes disponibles en el *Python Package Index (PyPI)*.

Para gestionar los paquetes que tenemos en nuestro sistema se utiliza la herramienta *pip*, una utilidad que también se incluye en la instalación de Python. Con ella podremos instalar, desinstalar y actualizar paquetes, según nuestras necesidades. A continuación se muestran las instrucciones que usaríamos para cada una de estas operaciones:

---

<sup>1</sup> Foto original de portada por [SpaceX](#) en Unsplash.

<sup>2</sup> También llamada «vanilla installation» ya que es la que viene por defecto y no se hace ninguna personalización.

Lista 2: Instalación, desinstalación y actualización del paquete pandas utilizando pip

```
$ pip install pandas
$ pip uninstall pandas
$ pip install pandas --upgrade
```

### 2.2.3 Entornos virtuales

Cuando trabajamos en distintos proyectos, no todos ellos requieren los mismos paquetes ni siquiera la misma versión de Python. La gestión de estas situaciones no es sencilla si únicamente instalamos paquetes y manejamos configuraciones a nivel global (*a nivel de máquina*). Es por ello que surge el concepto de **entornos virtuales**. Como su propio nombre indica se trata de crear distintos entornos en función de las necesidades de cada proyecto, y esto nos permite establecer qué versión de Python usaremos y qué paquetes instalaremos.

#### **virtualenv**

El paquete «built-in» de Python que nos proporciona la funcionalidad de crear y gestionar entornos virtuales se denomina **virtualenv**. Su instalación es sencilla a través del gestor de paquetes pip:

```
$ pip install virtualenv
```

Si bien con **virtualenv** tenemos las funcionalidades necesarias para trabajar con entornos virtuales, destacaría una herramienta llamada **virtualenvwrapper** que funciona *por encima* de **virtualenv** y que facilita las operaciones sobre entornos virtuales. Su instalación es equivalente a cualquier otro paquete Python:

```
$ pip install virtualenvwrapper
```

Veamos a continuación algunos de los comandos que nos ofrece:

```
$ ~/project1 > mkvirtualenv env1
Using base prefix '/Library/Frameworks/Python.framework/Versions/3.7'
New python executable in /Users/sdelquin/.virtualenvs/env1/bin/python3.7
Also creating executable in /Users/sdelquin/.virtualenvs/env1/bin/python
Installing setuptools, pip, wheel...
done.
virtualenvwrapper.user_scripts creating /Users/sdelquin/.virtualenvs/env1/bin/
  ↵predeactivate
virtualenvwrapper.user_scripts creating /Users/sdelquin/.virtualenvs/env1/bin/
  ↵postdeactivate
```

(continué en la próxima página)

(proviene de la página anterior)

```
virtualenvwrapper.user_scripts creating /Users/sdelquin/.virtualenvs/env1/bin/
↳preactivate
virtualenvwrapper.user_scripts creating /Users/sdelquin/.virtualenvs/env1/bin/
↳postactivate
virtualenvwrapper.user_scripts creating /Users/sdelquin/.virtualenvs/env1/bin/get_
↳env_details
$ (env1) ~/project1 > pip install requests
Collecting requests
Using cached requests-2.24.0-py2.py3-none-any.whl (61 kB)
Collecting idna<3,>=2.5
Using cached idna-2.10-py2.py3-none-any.whl (58 kB)
Collecting certifi>=2017.4.17
Using cached certifi-2020.6.20-py2.py3-none-any.whl (156 kB)
Collecting urllib3!=1.25.0,!>1.25.1,<1.26,>=1.21.1
Using cached urllib3-1.25.10-py2.py3-none-any.whl (127 kB)
Collecting chardet<4,>=3.0.2
Using cached chardet-3.0.4-py2.py3-none-any.whl (133 kB)
Installing collected packages: idna, certifi, urllib3, chardet, requests
Successfully installed certifi-2020.6.20 chardet-3.0.4 idna-2.10 requests-2.24.0
↳urllib3-1.25.10
$ (env1) ~/project1 > deactivate
$ ~/project1 > workon env1
$ (env1) ~/project1 > lssitepackages
--pycache__          distutils-precedence.pth    pkg_resources
↳urllib3-1.25.10.dist-info
_distutils_hack      easy_install.py        requests
↳wheel
certifi              idna                  requests-2.24.0.dist-info
↳wheel-0.34.2.dist-info
certifi-2020.6.20.dist-info idna-2.10.dist-info setuptools
chardet              pip                   setuptools-49.3.2.dist-info
chardet-3.0.4.dist-info pip-20.2.2.dist-info urllib3
$ (env1) ~/project1 >
```

- \$ mkvirtualenv env1: crea un entorno virtual llamado env1
- \$ pip install requests: instala el paquete requests dentro del entorno virtual env1
- \$ workon env1: activa el entorno virtual env1
- \$ lssitepackages: lista los paquetes instalados en el entorno virtual activo

## pyenv

pyenv permite cambiar fácilmente entre múltiples versiones de Python en un mismo sistema. Su instalación engloba varios pasos y está bien explicada en la [página del proyecto](#).

La mayor diferencia con respecto a [virtualenv](#) es que no instala las distintas versiones de Python a nivel global del sistema. En vez de eso, se suele crear una carpeta `.pyenv` en el HOME del usuario, donde todo está aislado sin generar intrusión en el sistema operativo.

Podemos hacer cosas como:

- Listar las versiones de Python instaladas:

```
$ pyenv versions
3.7.4
* 3.5.0 (set by /Users/yuu/.pyenv/version)
miniconda3-3.16.0
pypy-2.6.0
```

- Descubrir la versión global «activa» de Python:

```
$ python --version
Python 3.5.0
```

- Cambiar la versión global «activa» de Python:

```
$ pyenv global 3.7.4
$ python --version
Python 3.7.4
```

- Instalar una nueva versión de Python:

```
$ pyenv install 3.9.1
...
```

- Activar una versión de Python local por carpetas:

```
$ cd /cool-project
$ pyenv local 3.9.1
$ python --version
Python 3.9.1
```

También existe un módulo denominado `pyenv-virtualenv` para manejar entornos virtuales utilizando las ventajas que proporciona `pyenv`.

### 2.2.4 Editores

Existen multitud de editores en el mercado que nos pueden servir perfectamente para escribir código Python. Algunos de ellos incorporan funcionalidades extra y otros simplemente nos permiten editar ficheros. Cabe destacar aquí el concepto de **Entorno de Desarrollo Integrado**, más conocido por sus siglas en inglés **IDE**<sup>3</sup>. Se trata de una [aplicación informática](#) que proporciona servicios integrales para el desarrollo de software.

Podríamos decir que [Thonny](#) es un IDE de aprendizaje, pero existen muchos otros. Veamos un listado de editores de código que se suelen utilizar para desarrollo en Python:

- **Editores generales o IDEs con soporte para Python:**

- Eclipse + PyDev
- Sublime Text
- Atom
- GNU Emacs
- Vi-Vim
- Visual Studio (+ Python Tools)
- Visual Studio Code (+ Python Tools)

- **Editores o IDEs específicos para Python:**

- PyCharm
- Spyder
- Thonny

Cada editor tiene sus características (ventajas e inconvenientes). Supongo que la preferencia por alguno de ellos estará en base a la experiencia y a las necesidades que surjan. La parte buena es que hay diversidad de opciones para elegir.

### 2.2.5 Jupyter Notebook

Jupyter Notebook es una aplicación «open-source» que permite crear y compartir documentos que contienen código, ecuaciones, visualizaciones y texto narrativo. Podemos utilizarlo para propósito general aunque suele estar más enfocado a *ciencia de datos*: limpieza y transformación de datos, simulación numérica, modelado estadístico, visualización o «machine-learning»<sup>4</sup>.

---

<sup>3</sup> Integrated Development Environment.

<sup>4</sup> Término inglés utilizado para hacer referencia a algoritmos de aprendizaje automático.

Podemos verlo como un intérprete de Python (contiene un «kernel»<sup>5</sup> que permite ejecutar código) con la capacidad de incluir documentación en formato [Markdown](#), lo que potencia sus funcionalidades y lo hace adecuado para preparar cualquier tipo de material vinculado con lenguajes de programación.

---

**Nota:** Aunque su uso está más extendido en el mundo Python, existen muchos otros «kernels» sobre los que trabajar en Jupyter Notebook.

---

## 2.2.6 WSL

Si estamos trabajando en un sistema **Windows 10** es posible que nos encontremos más cómodos usando una terminal tipo «Linux», entre otras cosas para poder usar con facilidad las herramientas vistas en esta sección y preparar el entorno de desarrollo Python. Durante mucho tiempo esto fue difícil de conseguir hasta que *Microsoft* sacó WSL.

[WSL](#)<sup>6</sup> nos proporciona una *consola con entorno Linux* que podemos utilizar en nuestro *Windows 10* sin necesidad de instalar una máquina virtual o crear una partición para un Linux nativo. Es importante también saber que existen dos versiones de WSL hoy en día: WSL y WSL2. La segunda es bastante reciente (publicada a mediados de 2019), tiene mejor rendimiento y se adhiere más al comportamiento de un Linux nativo.

Para la instalación de WSL<sup>7</sup> hay que seguir los siguientes pasos:

1. Lanzamos Powershell con permisos de administrador.
2. Activamos la característica de WSL:

```
$ Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-
↪Subsystem-Linux
```

3. Descargamos la imagen de Ubuntu 20.04 que usaremos:

```
$ Invoke-WebRequest -Uri https://aka.ms/wslubuntu2004 -OutFile Ubuntu.appx -
↪UseBasicParsing
```

4. Finalmente, la instalamos:

```
$ Add-AppxPackage .\Ubuntu.appx
```

En este punto, WSL debería estar instalado correctamente, y debería también aparecer en el menú *Inicio*.

---

<sup>5</sup> Proceso específico para un lenguaje de programación que ejecuta instrucciones y actúa como interfaz de entrada/salida.

<sup>6</sup> Windows Subsystem for Linux.

<sup>7</sup> Tutorial de [instalación de WSL](#).



# CAPÍTULO 3

---

## Tipos de datos

---

Igual que en el mundo real cada objeto pertenece a una categoría, en programación manejamos objetos que tienen asociado un tipo determinado. En este capítulo se verán los tipos de datos básicos con los que podemos trabajar en Python.

## 3.1 Datos



Los programas están formados por **código** y **datos**. Pero a nivel interno de la memoria del ordenador no son más que una secuencia de bits. La interpretación de estos bits depende del lenguaje de programación, que almacena en la memoria no sólo el puro dato sino distintos metadatos.<sup>1</sup>

Cada «trozo» de memoria contiene realmente un objeto, de ahí que se diga que en Python **todo son objetos**. Y cada objeto tiene, al menos, los siguientes campos:

- Un **tipo** del dato almacenado.
- Un **identificador** único para distinguirlo de otros objetos.
- Un **valor** consistente con su tipo.

### 3.1.1 Tipos de datos

A continuación se muestran los distintos **tipos de datos** que podemos encontrar en Python, sin incluir aquellos que proveen paquetes externos:

---

<sup>1</sup> Foto original de portada por Alexander Sinn en Unsplash.

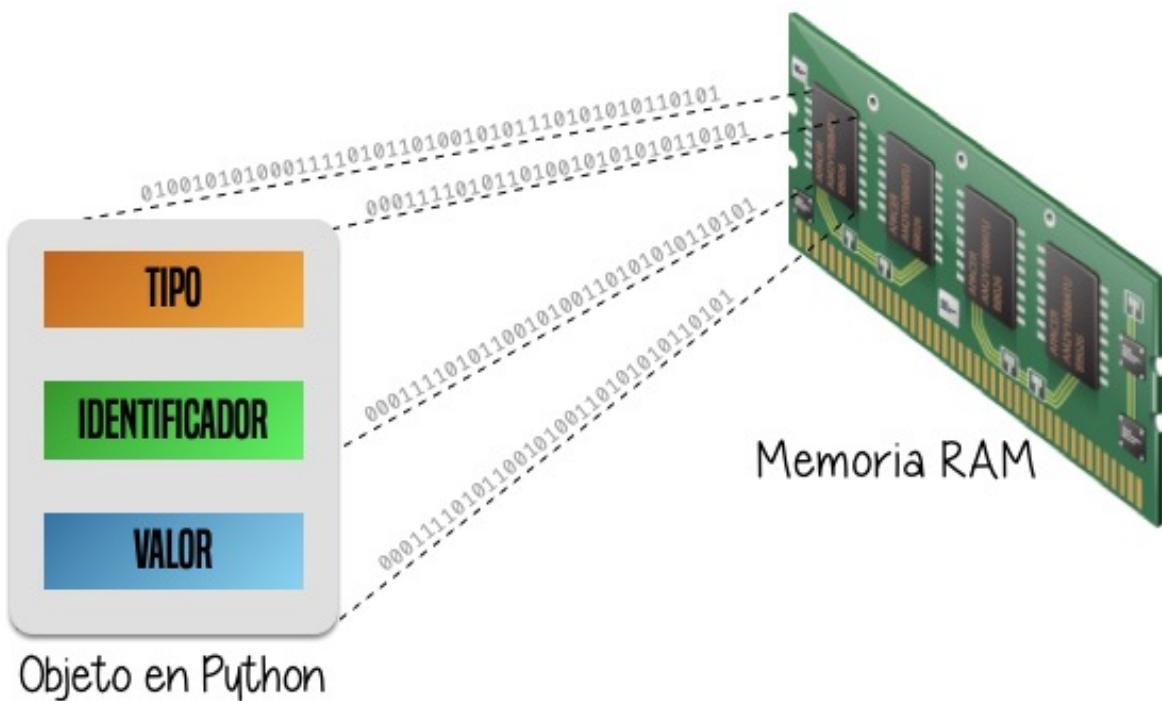


Figura 1: Esquema (*metadatos*) de un objeto en Python

Tabla 1: Tipos de datos en Python

Nombre	Tipo	Ejemplos
Booleano	bool	True, False
Entero	int	21, 34500, 34_500
Flotante	float	3.14, 1.5e3
Complejo	complex	2j, 3 + 5j
Cadena	str	'tfn', '''tenerife - islas canarias'''
Tupla	tuple	(1, 3, 5)
Lista	list	['Chrome', 'Firefox']
Conjunto	set	set([2, 4, 6])
Diccionario	dict	{'Chrome': 'v79', 'Firefox': 'v71'}

### 3.1.2 Variables

Las **variables** son fundamentales ya que permiten definir **nombres** para los **valores** que tenemos en memoria y que vamos a usar en nuestro programa.



nombre = valor

Figura 2: Uso de un *nombre* de variable

#### Reglas para nombrar variables

En Python existen una serie de reglas para los nombres de variables:

1. Sólo pueden contener los siguientes caracteres<sup>2</sup>:
  - Letras minúsculas.
  - Letras mayúsculas.
  - Dígitos.
  - Guiones bajos (\_).
2. Deben empezar con una letra o un guión bajo, nunca con un dígito.
3. No pueden ser una **palabra reservada** del lenguaje («keywords»).

<sup>2</sup> Para ser exactos, sí se pueden utilizar otros caracteres, e incluso *emojis* en los nombres de variables, aunque no suele ser una práctica extendida, ya que podría dificultar la legibilidad.

Podemos obtener un listado de las palabras reservadas del lenguaje de la siguiente forma:

```
>>> help('keywords')

Here is a list of the Python keywords. Enter any keyword to get more help.

False      class      from       or
None       continue   global     pass
True       def        if         raise
and        del        import    return
as         elif       in         try
assert    else       is         while
async     except    lambda   with
await     finally   nonlocal yield
break    for       not
```

---

**Nota:** Por lo general se prefiere dar nombres **en inglés** a las variables que utilicemos, ya que así hacemos nuestro código más «internacional» y con la posibilidad de que otras personas puedan leerlo, entenderlo y – llegado el caso – modificarlo. Es sólo una recomendación, nada impide que se haga en castellano.

---



---

**Importante:** Los nombres de variables son «case-sensitive»<sup>3</sup>. Por ejemplo, `stuff` y `Stuff` son nombres diferentes.

---

## Ejemplos de nombres de variables

Veamos a continuación una tabla con nombres de variables:

Tabla 2: Ejemplos de nombres de variables

Válido	Inválido	Razón
a	3	Empieza por un dígito
a3	3a	Empieza por un dígito
a_b_c___95	another-name	Contiene un carácter no permitido
_abc	with	Es una palabra reservada del lenguaje
_3a	3_a	Empieza por un dígito

<sup>3</sup> Sensible a cambios en mayúsculas y minúsculas.

## Convenciones para nombres

Mientras se sigan las *reglas* que hemos visto para nombrar variables no hay problema en la forma en la que se escriban, pero sí existe una convención para la **nomenclatura de las variables**. Se utiliza el llamado **snake\_case** en el que utilizamos **caracteres en minúsculas** (incluyendo dígitos si procede) junto con **guiones bajos** – cuando sean necesarios para su legibilidad –.<sup>4</sup> Por ejemplo, para nombrar una variable que almacene el número de canciones en nuestro ordenador, podríamos usar `num_songs`.

Esta convención, y muchas otras, están definidas en un documento denominado **PEP 8**. Se trata de una **guía de estilo** para escribir código en Python. Los **PEPs**<sup>5</sup> son las propuestas que se hacen para la mejora del lenguaje.

## Constantes

Un caso especial y que vale la pena destacar son las **constantes**. Podríamos decir que es un tipo de variable pero que su valor no cambia a lo largo de nuestro programa. Por ejemplo la velocidad de la luz. Sabemos que su valor es constante de 300.000 km/s. En el caso de las constantes utilizamos **mayúsculas** (incluyendo guiones bajos si es necesario) para nombrarlas. Para la velocidad de la luz nuestra constante se podría llamar: `LIGHT_SPEED`.

## Elegir buenos nombres

Se suele decir que una persona programadora (con cierta experiencia), a lo que dedica más tiempo, es a buscar un buen nombre para sus variables. Quizás pueda resultar algo excesivo pero da una idea de lo importante que es esta tarea. Es fundamental que los nombres de variables sean **autoexplicativos**, pero siempre llegando a un compromiso entre ser concisos y claros.

Supongamos que queremos buscar un nombre de variable para almacenar el número de elementos que se deben manejar en un pedido:

1. `n`
2. `num_elements`
3. `number_of_elements`
4. `number_of_elements_to_be_handled`

No existe una regla mágica que nos diga cuál es el nombre perfecto, pero podemos aplicar el *sentido común* y, a través de la experiencia, ir detectando aquellos nombres que sean más adecuados. En el ejemplo anterior, quizás podríamos descartar de principio la opción 1 y la 4 (por ser demasiado cortas o demasiado largas); nos quedaríamos con las otras dos. Si nos

<sup>4</sup> Más información sobre convenciones de nombres en [PEP 8](#).

<sup>5</sup> Del término inglés «Python Enhancement Proposals».

fijamos bien, casi no hay mucha información adicional de la opción 3 con respecto a la 2. Así que podríamos concluir que la opción 2 es válida para nuestras necesidades. En cualquier caso esto dependerá siempre del contexto del problema que estemos tratando.

### 3.1.3 Asignación

En Python se usa el símbolo `=` para **asignar** un valor a una variable:



Figura 3: Asignación de *valor* a *nombre* de variable

---

**Nota:** Hay que diferenciar la asignación en Python con la igualación en matemáticas. El símbolo `=` lo hemos aprendido desde siempre como una *equivalencia* entre dos *expresiones algebraicas*, sin embargo en Python nos indica una *sentencia de asignación*, del valor (en la derecha) al nombre (en la izquierda).

---

Algunos ejemplos de asignaciones a *variables*:

```
>>> total_population = 157503
>>> avg_temperature = 16.8
>>> city_name = 'San Cristóbal de La Laguna'
```

Algunos ejemplos de asignaciones a *constantes*:

```
>>> SOUND_SPEED = 343.2
>>> WATER_DENSITY = 997
>>> EARTH_NAME = 'La Tierra'
```

Python nos ofrece la posibilidad de hacer una **asignación múltiple** de la siguiente manera:

```
>>> tres = three = drei = 3
```

En este caso las tres variables utilizadas en el «lado izquierdo» tomarán el valor 3.

Recordemos que los nombres de variables deben seguir unas *reglas establecidas*, de lo contrario obtendremos un **error sintáctico** del intérprete de Python:

```
>>> 7floor = 40 # el nombre empieza por un dígito
  File "<stdin>", line 1
    7floor = 40
    ^
SyntaxError: invalid syntax

>>> for = 'Bucle' # el nombre usa la palabra reservada "for"
  File "<stdin>", line 1
    for = 'Bucle'
    ^
SyntaxError: invalid syntax

>>> screen-size = 14 # el nombre usa un carácter no válido
  File "<stdin>", line 1
SyntaxError: can't assign to operator
```

### Asignando una variable a otra variable

Las asignaciones que hemos hecho hasta ahora han sido de un **valor literal** a una variable. Pero nada impide que podamos hacer asignaciones de una variable a otra variable:

```
>>> people = 157503
>>> total_population = people
>>> total_population
157503
```

Eso sí, la variable que utilicemos como valor de asignación **debe existir previamente**, ya que si no es así, obtendremos un error informando de que no está definida:

```
>>> total_population = lot_of_people
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'lot_of_people' is not defined
```

De hecho, en el *lado derecho* de la asignación pueden aparecer *expresiones* más complejas que se verán en su momento.

## Conocer el valor de una variable

Hemos visto previamente cómo asignar un valor a una variable, pero aún no sabemos cómo «comprobar» el valor que tiene dicha variable. Para ello podemos utilizar dos estrategias:

1. Si estamos en una «shell» de Python, basta con que usemos el nombre de la variable:

```
>>> final_stock = 38934  
>>> final_stock  
38934
```

2. Si estamos escribiendo un programa desde el editor, podemos hacer uso de `print`:

```
final_stock = 38934  
print(final_stock)
```

---

**Nota:** `print` sirve también cuando estamos en una sesión interactiva de Python («shell»)

---

## Conocer el tipo de una variable

Para poder descubrir el tipo de un literal o una variable, Python nos ofrece la función `type()`. Veamos algunos ejemplos de su uso:

```
>>> type(9)  
int  
  
>>> type(1.2)  
float  
  
>>> height = 3718  
>>> type(height)  
int  
  
>>> sound_speed = 343.2  
>>> type(sound_speed)  
float
```

---

### Ejercicio

1. Asigna un valor entero 2001 a la variable `space_odyssey` y muestra su valor.
2. Descubre el tipo del literal 'Good night & Good luck'.
3. Identifica el tipo del literal `True`.

4. Asigna la expresión `10 * 3.0` a la variable `result` y muestra su tipo.
- 

### 3.1.4 Mutabilidad

#### Nivel avanzado

Las variables son nombres, no lugares. Detrás de esta frase se esconde la reflexión de que cuando asignamos un valor a una variable, lo que realmente está ocurriendo es que se hace **apuntar** el nombre de la variable a una zona de memoria en el que se representa el objeto (con su valor).

Si realizamos la asignación de una variable a un valor lo que está ocurriendo es que el nombre de la variable es una **referencia** al valor, no el valor en sí mismo:

```
>>> a = 5
```

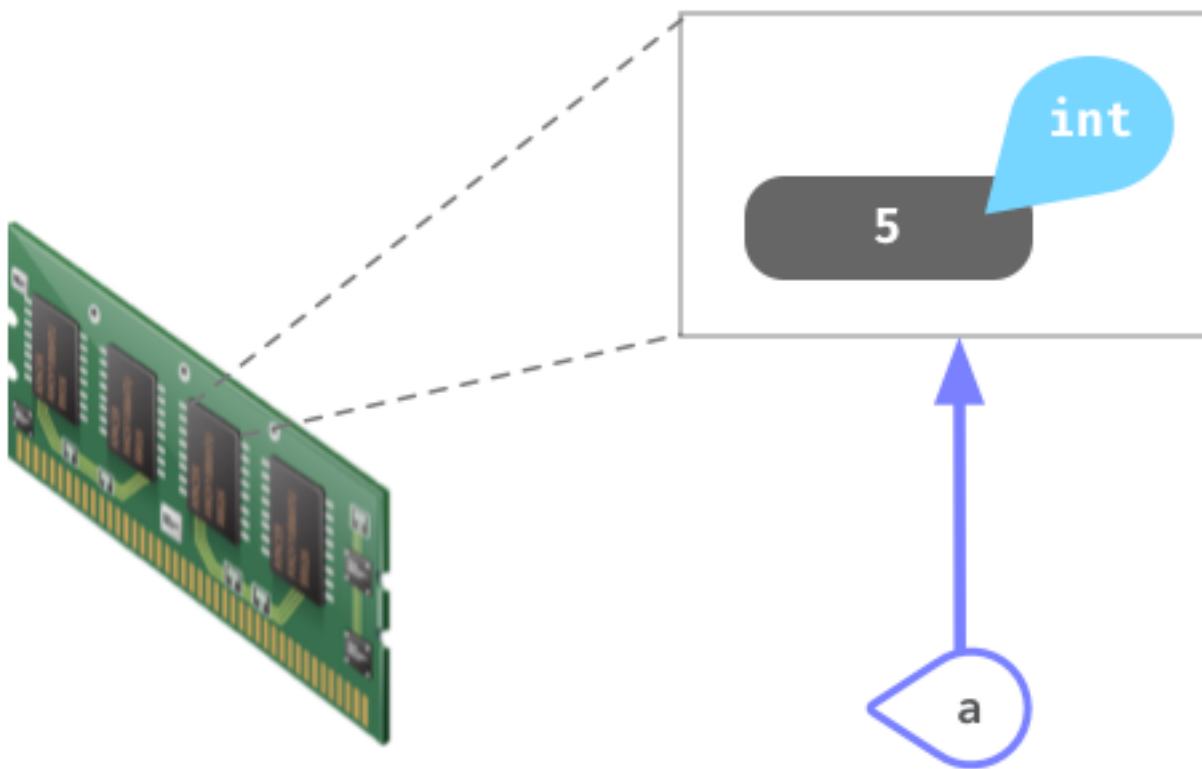


Figura 4: Representación de la asignación de valor a variable

Si ahora «copiamos» el valor de `a` en otra variable `b` se podría esperar que hubiera otro espacio en memoria para dicho valor, pero como ya hemos dicho, son referencias a memoria:

```
>>> b = a
```

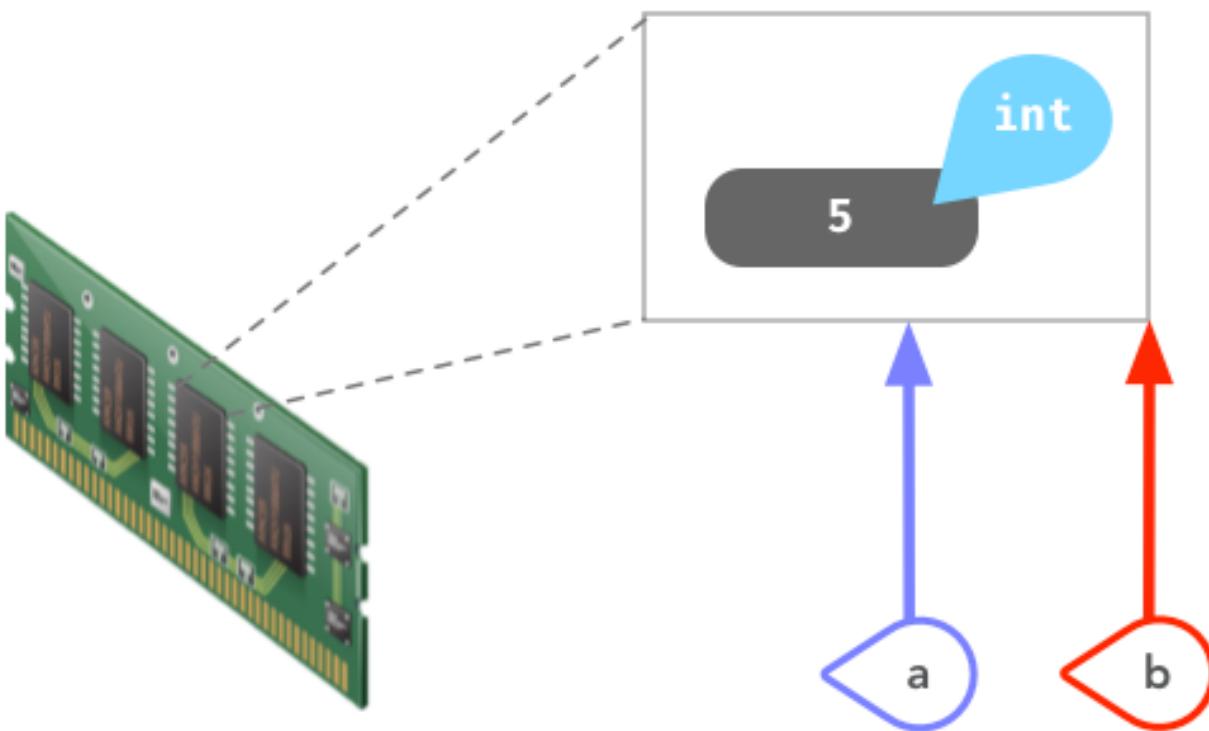


Figura 5: Representación de la asignación de una variable a otra variable

La función `id()` nos permite conocer la dirección de memoria<sup>6</sup> de un objeto en Python. A través de ella podemos comprobar que los dos objetos que hemos creado «apuntan» a la misma zona de memoria:

```
>>> id(a)
4445989712

>>> id(b)
4445989712
```

¿Y esto qué tiene que ver con la **mutabilidad**? Pues se dice, por ejemplo, que un **entero** es **inmutable** ya que a la hora de modificar su valor obtenemos una nueva *zona de memoria*, o lo que es lo mismo, un nuevo objeto:

```
>>> a = 5
>>> id(a)
4310690224
```

(continué en la próxima página)

<sup>6</sup> Esto es un detalle de implementación de CPython.

(provien de la página anterior)

```
>>> a = 7  
>>> id(a)  
4310690288
```

Sin embargo, si tratamos con **listas**, podemos ver que la modificación de alguno de sus valores no implica un cambio en la posición de memoria de la variable, por lo que se habla de objetos **mutables**.

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/lvCyXeL>

La característica de que los nombres de variables sean referencias a objetos en memoria es la que hace posible diferenciar entre **objetos mutables e inmutables**:

Inmutable	Mutable
bool	list
int	set
float	dict
str	
tuple	

---

**Importante:** El hecho de que un tipo de datos sea inmutable significa que no podemos modificar su valor «in-situ», pero siempre podremos asignarle un nuevo valor (hacerlo apuntar a otra zona de memoria).

---

## 3.1.5 Funciones «built-in»

### Nivel intermedio

Hemos ido usando una serie de *funciones* sin ser especialmente conscientes de ello. Esto se debe a que son funciones «built-in» o incorporadas por defecto en el propio lenguaje Python.

Tabla 3: Funciones «built-in»

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	any()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()

continué en la próxima página

Tabla 3 – proviene de la página anterior

breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	<u>__import__()</u>
complex()	hasattr()	max()	round()	

Los detalles de estas funciones se puede consultar en la [documentación oficial de Python](#).

## AMPLIAR CONOCIMIENTOS

- Basic Data Types in Python
- Variables in Python
- Immutability in Python

## 3.2 Números



En esta sección veremos los tipos de datos numéricos que ofrece Python centrándonos en **booleanos, enteros y flotantes**.<sup>1</sup>

### 3.2.1 Booleanos

George Boole es considerado como uno de los fundadores del campo de las ciencias de la computación y fue el creador del [Álgebra de Boole](#) que da lugar, entre otras estructuras algebraicas, a la [Lógica binaria](#). En esta lógica las variables sólo pueden tomar dos valores discretos: **verdadero** o **falso**.

El tipo de datos `bool` proviene de lo explicado anteriormente y admite dos posibles valores:

- `True` que se corresponde con *verdadero* (y también con **1** en su representación numérica).
- `False` que se corresponde con *falso* (y también con **0** en su representación numérica).

Veamos un ejemplo de su uso:

```
>>> is_opened = True
>>> is_opened
True

>>> has_sugar = False
>>> has_sugar
False
```

La primera variable `is_opened` está representando el hecho de que algo esté abierto, y al tomar el valor `True` podemos concluir que sí. La segunda variable `has_sugar` nos indica si una bebida tiene azúcar; dado que toma el valor `False` inferimos que no lleva azúcar.

**Atención:** Tal y como se explicó en [este apartado](#), los nombres de variables son «case-sensitive». De igual modo el tipo booleano toma valores `True` y `False` con **la primera letra en mayúsculas**. De no ser así obtendríamos un error sintáctico.

```
>>> is_opened = true
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
>>> has_sugar = false
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'false' is not defined
```

<sup>1</sup> Foto original de portada por Brett Jordan en Unsplash.

### 3.2.2 Enteros

Los números enteros no tienen decimales pero sí pueden contener signo y estar expresados en alguna base distinta de la usual (base 10).

#### Literales enteros

Veamos algunos ejemplos de números enteros:

```
>>> 8
8
>>> 0
0
>>> 08
File "<stdin>", line 1
  08
  ^
SyntaxError: invalid token
>>> 99
99
>>> +99
99
>>> -99
-99
>>> 3000000
3000000
>>> 3_000_000
3000000
```

Dos detalles a tener en cuenta:

- No podemos comenzar un número entero por `0`.
- Python permite dividir los números enteros con *guiones bajos* `_` para clarificar su lectura/escritura. A efectos prácticos es como si esos guiones bajos no existieran.

#### Operaciones con enteros

A continuación se muestra una tabla con las distintas operaciones sobre enteros que podemos realizar en Python:

Tabla 4: Operaciones con enteros en Python

Operador	Descripción	Ejemplo	Resultado
<code>+</code>	Suma	<code>3 + 9</code>	12

continué en la próxima página

Tabla 4 – proviene de la página anterior

Operador	Descripción	Ejemplo	Resultado
-	Resta	6 - 2	4
*	Multiplicación	5 * 5	25
/	División flotante	9 / 2	4.5
//	División entera	9 // 2	4
%	Módulo	9 % 4	1
**	Exponenciación	2 ** 4	16

Veamos algunas pruebas de estos operadores:

```
>>> 2 + 8 + 4  
14  
>>> 4 ** 4  
256  
>>> 7 / 3  
2.3333333333333335  
>>> 7 // 3  
2  
>>> 6 / 0  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ZeroDivisionError: division by zero
```

Es de buen estilo de programación **dejar un espacio** entre cada operador. Además hay que tener en cuenta que podemos obtener errores dependiendo de la operación (más bien de los *operando*s) que estemos utilizando, como es el caso de la *división por cero*.

## Asignación aumentada

Python nos ofrece la posibilidad de escribir una asignación aumentada mezclando la *asignación* y un *operador*.

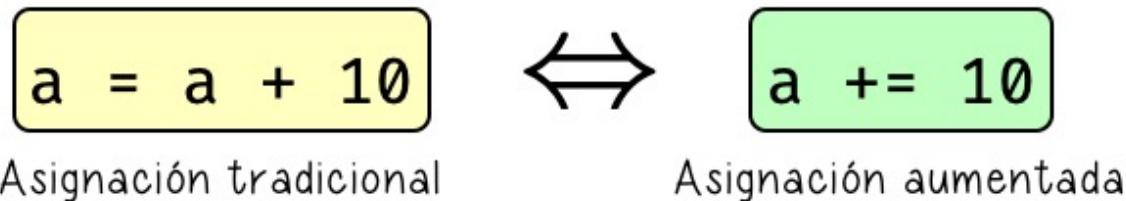


Figura 6: Asignación aumentada en Python

Supongamos que disponemos de 100 vehículos en stock y que durante el pasado mes se han

vendido 20 de ellos. Veamos cómo sería el código con asignación tradicional vs. asignación aumentada:

Lista 1: Asignación tradicional

```
>>> total_cars = 100
>>> sold_cars = 20
>>> total_cars = total_cars - sold_cars
>>> total_cars
80
```

Lista 2: Asignación aumentada

```
>>> total_cars = 100
>>> sold_cars = 20
>>> total_cars -= sold_cars
>>> total_cars
80
```

Estas dos formas son equivalentes a nivel de resultados y funcionalidad, pero obviamente tienen diferencias de escritura y legibilidad. De este mismo modo, podemos aplicar un formato compacto al resto de operaciones:

```
>>> random_number = 15

>>> random_number += 5
>>> random_number
20

>>> random_number *= 3
>>> random_number
60

>>> random_number //=
>>> random_number
15

>>> random_number **= 1
>>> random_number
15
```

## Módulo

La operación **módulo** (también llamado **resto**), cuyo símbolo en Python es %, se define como el resto de dividir dos números. Veamos un ejemplo para entender bien su funcionamiento:

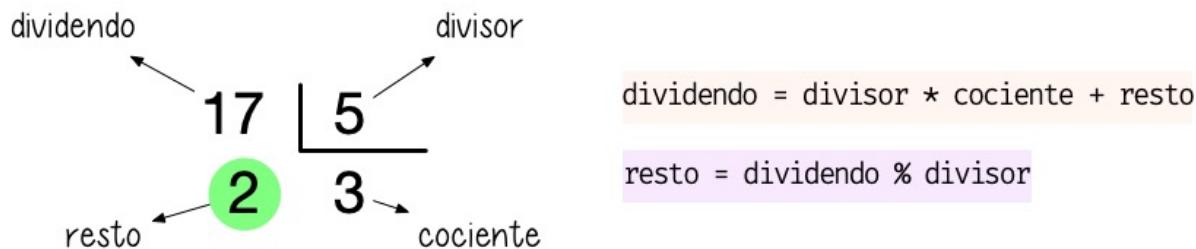


Figura 7: Operador «módulo» en Python

```
>>> dividendo = 17
>>> divisor = 5

>>> cociente = dividendo // divisor  # división entera
>>> resto = dividendo % divisor

>>> cociente
3
>>> resto
2
```

## Exponenciación

Para elevar un número a otro en Python utilizamos el operador de exponenciación \*\*:

```
>>> 4 ** 3
64
>>> 4 * 4 * 4
64
```

Se debe tener en cuenta que también podemos elevar un número entero a un **número decimal**. En este caso es como si estuviéramos haciendo una *raíz*<sup>2</sup>. Por ejemplo:

$$4^{\frac{1}{2}} = 4^{0.5} = \sqrt{4} = 2$$

Hecho en Python:

---

<sup>2</sup> No siempre es una raíz cuadrada porque se invierten numerador y denominador.

```
>>> 4 ** 0.5
2.0
```

## Ejercicio

Una ecuación de segundo grado se define como  $ax^2 + bx + c = 0$ , y (en determinados casos) tiene dos soluciones:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Dados los coeficientes  $a=4$ ,  $b=-6$  y  $c=2$  calcule sus dos soluciones. Tenga en cuenta subdividir los cálculos y reutilizar variables (por ejemplo el [discriminante](#)).

La solución para los valores anteriores es:

- $x1 = 1$
- $x2 = 0.5$

Recuerde que la **raíz cuadrada** se puede calcular como la exponenciación a  $\frac{1}{2}$ .

Puede comprobar los resultados para otros valores de entrada con esta [aplicación para resolver ecuaciones cuadráticas](#).

## Valor absoluto

Python ofrece la función `abs()` para obtener el valor absoluto de un número:

```
>>> abs(-1)
1

>>> abs(1)
1

>>> abs(-3.14)
3.14

>>> abs(3.14)
3.14
```

## Límite de un entero

## Nivel avanzado

¿Cómo de grande puede ser un `int` en Python? La respuesta es **de cualquier tamaño**. Por poner un ejemplo, supongamos que queremos representar un `centillón`. Este valor viene a ser un «1» seguido por ¡600 ceros! ¿Será capaz Python de almacenarlo?

**Nota:** En muchos lenguajes tratar con enteros tan largos causaría un «integer overflow». No es el caso de Python que puede manejar estos valores sin problema.

### 3.2.3 Flotantes

Los números en **punto flotante**<sup>3</sup> tienen **parte decimal**. Veamos algunos ejemplos de flotantes en Python.

<sup>3</sup> Punto o coma flotante es una notación científica usada por computadores.

Lista 3: Distintas formas de escribir el flotante *4.0*

```
>>> 4.0
4.0
>>> 4.
4.0
>>> 04.0
4.0
>>> 04.
4.0
>>> 4.000_000
4.0
>>> 4e0 # 4.0 * (10 ** 0)
4.0
```

## Conversión de tipos

El hecho de que existan distintos tipos de datos en Python (y en el resto de lenguajes de programación) es una ventaja a la hora de representar la información del mundo real de la mejor manera posible. Pero también se hace necesario buscar mecanismos para convertir unos tipos de datos en otros.

## Conversión implícita

Cuando mezclamos enteros, booleanos y flotantes, Python realiza automáticamente una conversión implícita (o **promoción**) de los valores al tipo de «mayor rango». Veamos algunos ejemplos de esto:

```
>>> True + 25
26
>>> 7 * False
0
>>> True + False
1
>>> 21.8 + True
22.8
>>> 10 + 11.3
21.3
```

Podemos resumir la conversión implícita en la siguiente tabla:

Tipo 1	Tipo 2	Resultado
bool	int	int
bool	float	float
int	float	float

Se puede ver claramente que la conversión numérica de los valores booleanos es:

- `True` ↗ 1
- `False` ↗ 0

## Conversión explícita

Aunque más adelante veremos el concepto de **función**, desde ahora podemos decir que existen una serie de funciones para realizar conversiones explícitas de un tipo a otro:

**bool()** Convierte el tipo a *booleano*.

**int()** Convierte el tipo a *entero*.

**float()** Convierte el tipo a *flotante*.

Veamos algunos ejemplos de estas funciones:

```
>>> bool(1)
True
>>> bool(0)
False
>>> int(True)
1
>>> int(False)
0
>>> float(1)
1.0
>>> float(0)
0.0
>>> float(True)
1.0
>>> float(False)
0.0
```

En el caso de que usemos la función `int()` sobre un valor flotante, nos retorna su **parte baja**:

$$\text{int}(x) = \lfloor x \rfloor$$

Por ejemplo:

```
>>> int(3.1)
3
>>> int(3.5)
3
>>> int(3.9)
3
```

Para **obtener el tipo** de una variable podemos hacer uso de la función `type()`:

```
>>> is_raining = False

>>> type(is_raining)
<class 'bool'>

>>> sound_level = 35
>>> type(sound_level)
<class 'int'>

>>> temperature = 36.6
>>> type(temperature)
<class 'float'>
```

Igualmente existe la posibilidad de **comprobar el tipo** que tiene una variable mediante la función `isinstance()`:

```
>>> isinstance(is_raining, bool)
True
>>> isinstance(sound_level, int)
True
>>> isinstance(temperature, float)
True
```

## Ejercicio

Existe una aproximación al seno de un ángulo  $x$  expresado en *grados*:

$$\sin(x) \approx \frac{4x(180 - x)}{40500 - x(180 - x)}$$

Calcule dicha aproximación utilizando operaciones en Python. Descomponga la expresión en subcálculos almacenados en variables. Tenga en cuenta aquellas expresiones comunes para no repetir cálculos y seguir el principio DRY.

¿Qué tal funciona la aproximación? Compare sus resultados con estos:

- $\sin(90) = 1.0$
- $\sin(45) = 0.7071067811865475$

- $\sin(50) = 0.766044443118978$
- 

## Errores de aproximación

### Nivel intermedio

Supongamos el siguiente cálculo:

```
>>> (19 / 155) * (155 / 19)
0.9999999999999999
```

Debería dar 1.0, pero no es así puesto que la representación interna de los valores en **coma flotante** sigue el estándar **IEEE 754** y estamos trabajando con **aritmética finita**.

Aunque existen distintas formas de solventar esta limitación, de momento veremos una de las más sencillas utilizando la función «built-in» `round()` que nos permite redondear un número flotante a un número determinado de decimales:

```
>>> pi = 3.14159265359
>>> round(pi)
3
>>> round(pi, 1)
3.1
>>> round(pi, 2)
3.14
>>> round(pi, 3)
3.142
>>> round(pi, 4)
3.1416
>>> round(pi, 5)
3.14159
```

Para el caso del error de aproximación que nos ocupa:

```
>>> result = (19 / 155) * (155 / 19)
>>> round(result, 1)
1.0
```

**Prudencia:** `round()` aproxima al valor más cercano, mientras que `int()` obtiene siempre el entero «por abajo».

### 3.2.4 Bases

#### Nivel intermedio

Los valores numéricos con los que estamos acostumbrados a trabajar están en **base 10** (o decimal). Esto indica que disponemos de 10 «símbolos» para representar las cantidades. En este caso del **0** al **9**.

Pero también es posible representar números en **otras bases**. Python nos ofrece una serie de **prefijos** y **funciones** para este cometido.

#### Base binaria

Cuenta con **2** símbolos para representar los valores: **0** y **1**.

##### Prefijo: `0b`

```
>>> 0b1001  
9  
>>> 0b1100  
12
```

##### Función: `bin()`

```
>>> bin(9)  
'0b1001'  
>>> bin(12)  
'0b1100'
```

#### Base octal

Cuenta con **8** símbolos para representar los valores: **0, 1, 2, 3, 4, 5, 6 y 7**.

##### Prefijo: `0o`

```
>>> 0o6243  
3235  
>>> 0o1257  
687
```

##### Función: `oct()`

```
>>> oct(3235)  
'0o6243'  
>>> oct(687)  
'0o1257'
```

## Base hexadecimal

Cuenta con **16** símbolos para representar los valores: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F.

Prefijo: 0x

```
>>> 0x7F2A  
32554  
>>> 0x48FF  
18687
```

Función: hex()

```
>>> hex(32554)  
'0x7f2a'  
>>> hex(18687)  
'0x48ff'
```

---

**Nota:** Las letras para la representación hexadecimal no atienden a mayúsculas y minúsculas.

---

## EJERCICIOS DE REPASO

1. Escriba un programa en Python que acepte el radio de un círculo y compute su área ([solución](#)).

Entrada: 5

Salida: 8.5

2. Escriba un programa en Python que acepte el radio de una esfera y obtenga su volumen ([solución](#)).

Entrada: 5

Salida: 523.3333333333334

3. Escriba un programa en Python que acepte la base y la altura de un triángulo y compute su área ([solución](#)).

Entrada: base=4; altura=5

Salida: 10.0

4. Escriba un programa en Python que compute el futuro valor de una cantidad de dinero, a partir del capital inicial, el tipo de interés y el número de años ([solución](#)).

Entrada: capital=10000; interés=3.5; años=7

Salida: 12722.792627665729

5. Escriba un programa en Python que calcule la distancia euclídea entre dos puntos  $(x_1, y_1)$  y  $(x_2, y_2)$  ([solución](#)).

Entrada:  $(x_1 = 3, y_1 = 5); (x_2 = -7, y_2 = -4)$

Salida: 13.45362404707371

## AMPLIAR CONOCIMIENTOS

- [The Python Square Root Function](#)
- [How to Round Numbers in Python](#)
- [Operators and Expressions in Python](#)

## 3.3 Cadenas de texto



Las cadenas de texto son **secuencias de caracteres**. También se les conoce como «strings» y nos permiten almacenar información textual de forma muy cómoda.<sup>1</sup>

Es importante destacar que Python 3 almacena los caracteres codificados en el estándar **Unicode**, lo que es una gran ventaja con respecto a versiones antiguas del lenguaje. Además permite representar una cantidad ingente de símbolos incluyendo los famosos emojis 😎.

---

<sup>1</sup> Foto original de portada por [Roman Kraft](#) en Unsplash.

### 3.3.1 Creando «strings»

Para escribir una cadena de texto en Python basta con rodear los caracteres con comillas simples<sup>6</sup>:

```
>>> 'Mi primera cadena en Python'  
'Mi primera cadena en Python'
```

Para incluir *comillas dobles* dentro de la cadena de texto no hay mayor inconveniente:

```
>>> 'Los llamados "strings" son secuencias de caracteres'  
'Los llamados "strings" son secuencias de caracteres'
```

Puede surgir la duda de cómo incluimos *comillas simples* dentro de la propia cadena de texto. Veamos soluciones para ello:

Lista 4: Comillas simples escapadas

```
>>> 'Los llamados \'strings\' son secuencias de caracteres'  
"Los llamados 'strings' son secuencias de caracteres"
```

Lista 5: Comillas simples dentro de comillas dobles

```
>>> "Los llamados 'strings' son secuencias de caracteres"  
"Los llamados 'strings' son secuencias de caracteres"
```

En la primera opción estamos **escapando** las comillas simples para que no sean tratadas como caracteres especiales. En la segunda opción estamos creando el «string» con comillas dobles (por fuera) para poder incluir directamente las comillas simples (por dentro). Python también nos ofrece esta posibilidad.

## Comillas triples

Hay una forma alternativa de crear cadenas de texto utilizando *comillas triples*. Su uso está pensado principalmente para **cadenas multilínea**:

```
>>> poem = '''To be, or not to be, that is the question:  
... Whether 'tis nobler in the mind to suffer  
... The slings and arrows of outrageous fortune,  
... Or to take arms against a sea of troubles'''
```

---

**Importante:** Los tres puntos ... que aparecen a la izquierda de las líneas no están incluidos en la cadena de texto. Es el símbolo que ofrece el intérprete de Python cuando saltamos de

<sup>6</sup> También es posible utilizar comillas dobles. Yo me he decantado por las comillas simples ya que quedan más limpias y suele ser el formato que devuelve el propio intérprete de Python.

línea.

---

### Cadena vacía

La cadena vacía es aquella que no contiene ningún carácter. Aunque a priori no lo pueda parecer, es un recurso importante en cualquier código. Su representación en Python es la siguiente:

```
>>> ''
''
```

### 3.3.2 Conversión

Podemos crear «strings» a partir de otros tipos de datos usando la función `str()`:

```
>>> str(True)
'True'
>>> str(10)
'10'
>>> str(21.7)
'21.7'
```

### 3.3.3 Secuencias de escape

Python permite **escapar** el significado de algunos caracteres para conseguir otros resultados. Si escribimos una barra invertida \ antes del carácter en cuestión, le otorgamos un significado especial.

Quizás la *secuencia de escape* más conocida es \n que representa un *salto de línea*, pero existen muchas otras:

```
# Salto de línea
>>> msg = 'Primera línea\nSegunda línea\nTercera línea'
>>> print(msg)
Primera línea
Segunda línea
Tercera línea

# Tabulador
>>> msg = 'Valor = \t40'
>>> print(msg)
Valor =      40
```

(continué en la próxima página)

(provien de la página anterior)

```
# Comilla simple
>>> msg = 'Necesitamos \'escapar\' la comilla simple'
>>> print(msg)
Necesitamos 'escapar' la comilla simple

# Barra invertida
>>> msg = 'Capítulo \\ Sección \\ Encabezado'
>>> print(msg)
Capítulo \ Sección \ Encabezado
```

---

**Nota:** Al utilizar la función `print()` es cuando vemos realmente el resultado de utilizar los caracteres escapados.

---

## Expresiones literales

### Nivel intermedio

Hay situaciones en las que nos interesa que los caracteres especiales pierdan ese significado y poder usarlos de otra manera. Existe un modificar de cadena que proporciona Python para tratar el texto *en bruto*. Es el llamado «raw data» y se aplica anteponiendo una `r` a la cadena de texto.

Veamos algunos ejemplos:

```
>>> text = 'abc\ndef'
>>> print(text)
abc
def

>>> text = r'abc\ndef'
>>> print(text)
abc\ndef

>>> text = 'a\tb\tc'
>>> print(text)
a    b    c

>>> text = r'a\tb\tc'
>>> print(text)
a\tb\tc
```

---

**Consejo:** El modificador `r''` es muy utilizado para la escritura de **expresiones regulares**.

---

### 3.3.4 Más sobre `print()`

Hemos estado utilizando la función `print()` de forma sencilla, pero admite `algunos parámetros interesantes`:

```

1  >>> msg1 = '¿Sabes por qué estoy acá?'
2  >>> msg2 = 'Porque me apasiona'
3
4  >>> print(msg1, msg2)
5  ¿Sabes por qué estoy acá? Porque me apasiona
6
7  >>> print(msg1, msg2, sep='|')
8  ¿Sabes por qué estoy acá?|Porque me apasiona
9
10 >>> print(msg2, end='!!!')
11 Porque me apasiona!!!

```

**Línea 4:** Podemos imprimir todas las variables que queramos separándolas por comas.

**Línea 7:** El *separador por defecto* entre las variables es un *espacio*, podemos cambiar el carácter que se utiliza como separador entre cadenas.

**Línea 10:** El *carácter de final de texto* es un *salto de línea*, podemos cambiar el carácter que se utiliza como final de texto.

### 3.3.5 Leer datos desde teclado

Los programas se hacen para tener interacción con el usuario. Una de las formas de interacción es solicitar la entrada de datos por teclado. Como muchos otros lenguajes de programación, Python también nos ofrece la posibilidad de leer la información introducida por teclado. Para ello se utiliza la función `input()`:

```

>>> name = input('Introduzca su nombre: ')
Introduzca su nombre: Sergio
>>> name
'Sergio'
>>> type(name)
str

>>> age = input('Introduzca su edad: ')
Introduzca su edad: 41

```

(continué en la próxima página)

(provien de la página anterior)

```
>>> age  
'41'  
>>> type(age)  
str
```

---

**Nota:** La función `input()` siempre nos devuelve un objeto de tipo cadena de texto o `str`. Tenerlo muy en cuenta a la hora de trabajar con números, ya que debemos realizar una *conversión explícita*.

---

### Ejercicio

Escriba un programa en Python que lea por teclado dos números enteros y muestre por pantalla el resultado de realizar las operaciones básicas entre ellos.

### Ejemplo

- Valores de entrada 7 y 4.
- Salida esperada:

```
7+4=11  
7-4=3  
7*4=28  
7/4=1.75
```

---

**Consejo:** Aproveche todo el potencial que ofrece `print()` para conseguir la salida esperada.

---

## 3.3.6 Operaciones con «strings»

### Combinar cadenas

Podemos combinar dos o más cadenas de texto utilizando el operador `+`:

```
>>> proverb1 = 'Cuando el río suena'  
>>> proverb2 = 'agua lleva'  
  
>>> proverb1 + proverb2  
'Cuando el río suenaagua lleva'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> proverb1 + ', ' + proverb2 # incluimos una coma
'Cuando el río suena, agua lleva'
```

## Repetir cadenas

Podemos repetir dos o más cadenas de texto utilizando el operador \*:

```
>>> reaction = 'Wow'
>>> reaction * 4
'WowWowWowWow'
```

## Obtener un carácter

Los «strings» están **indexados** y cada carácter tiene su propia posición. Para obtener un único carácter dentro de una cadena de texto es necesario especificar su **índice** dentro de corchetes [...].

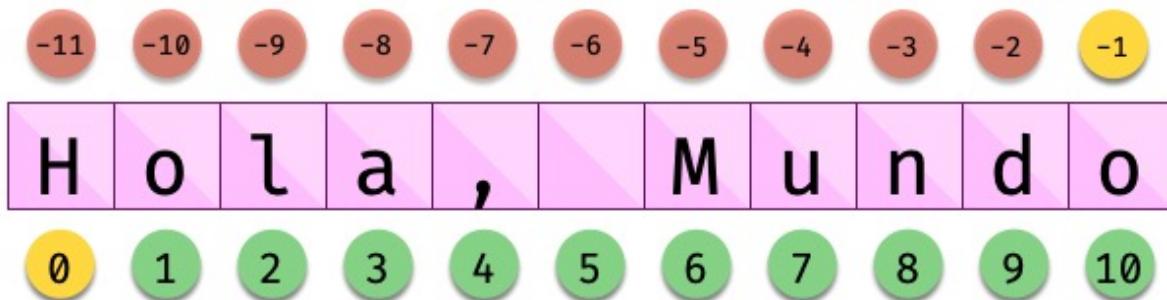


Figura 8: Indexado de una cadena de texto

Veamos algunos ejemplos de acceso a caracteres:

```
>>> sentence = 'Hola, Mundo'

>>> sentence[0]
'H'
>>> sentence[-1]
'o'
>>> sentence[4]
', '
>>> sentence[-5]
'M'
```

---

**Truco:** Nótese que existen tanto **índices positivos** como **índices negativos** para acceder a cada carácter de la cadena de texto. A priori puede parecer redundante, pero es muy útil en determinados casos.

---

En caso de que intentemos acceder a un índice que no existe, obtendremos un error por *fuera de rango*:

```
>>> sentence[50]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

**Advertencia:** Téngase en cuenta que el indexado de una cadena de texto siempre empieza en **0** y termina en **una unidad menos de la longitud** de la cadena.

Las cadenas de texto son tipos de datos *inmutables*. Es por ello que no podemos modificar un carácter directamente:

```
>>> song = 'Hey Jude'
>>> song[4] = 'D'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

---

**Truco:** Existen formas de modificar una cadena de texto que veremos más adelante, aunque realmente no estemos transformando el original sino creando un nuevo objeto con las modificaciones.

---

### Trocear una cadena

Es posible extraer «trozos» («rebanadas») de una cadena de texto<sup>2</sup>. Tenemos varias aproximaciones para ello:

**[:]** Extrae la secuencia entera desde el comienzo hasta el final. Es una especie de **copia** de toda la cadena de texto.

**[start:]** Extrae desde **start** hasta el final de la cadena.

**[:end]** Extrae desde el comienzo de la cadena hasta **end menos 1**.

---

<sup>2</sup> El término usado en inglés es *slice*.

**[start:end]** Extrae desde `start` hasta `end menos 1`.

**[start:end:step]** Extrae desde `start` hasta `end menos 1` haciendo saltos de tamaño `step`.

Veamos la aplicación de cada uno de estos accesos a través de un ejemplo:

```
>>> proverb = 'Agua pasada no mueve molino'  
  
>>> proverb[:]  
'Agua pasada no mueve molino'  
  
>>> proverb[12:]  
'no mueve molino'  
  
>>> proverb[:11]  
'Agua pasada'  
  
>>> proverb[5:11]  
'pasada'  
  
>>> proverb[5:11:2]  
'psd'
```

---

**Importante:** El troceado siempre llega a una unidad menos del índice final que hayamos especificado. Sin embargo el comienzo sí coincide con el que hemos puesto.

---

## Longitud de una cadena

Para obtener la longitud de una cadena podemos hacer uso de `len()`, una función común a prácticamente todos los tipos y estructuras de datos en Python:

```
>>> proberb = 'Lo cortés no quita lo valiente'  
>>> len(proverb)  
27  
  
>>> empty = ''  
>>> len(empty)  
0
```

## Pertenencia de un elemento

Si queremos comprobar que una determinada subcadena se encuentra en una cadena de texto utilizamos el operador `in` para ello. Se trata de una expresión que tiene como resultado un valor «booleano» verdadero o falso:

```
>>> proverb = 'Más vale malo conocido que bueno por conocer'  
  
>>> 'malo' in proverb  
True  
  
>>> 'bueno' in proverb  
True  
  
>>> 'regular' in proverb  
False
```

Habrá que prestar atención al caso en el que intentamos descubrir si una subcadena **no está** en la cadena de texto:

```
>>> dna_sequence = 'ATGAAATTGAAATGGGA'  
  
>>> not('C' in dna_sequence) # Primera aproximación  
True  
  
>>> 'C' not in dna_sequence # Forma pitónica  
True
```

## Dividir una cadena

Una tarea muy común al trabajar con cadenas de texto es dividirlas por algún tipo de *separador*. En este sentido, Python nos ofrece la función `split()`, que debemos usar anteponiendo el «string» que queramos dividir:

```
>>> proverb = 'No hay mal que por bien no venga'  
>>> proverb.split()  
['No', 'hay', 'mal', 'que', 'por', 'bien', 'no', 'venga']  
  
>>> tools = 'Martillo,Sierra,Destornillador'  
>>> tools.split(',')  
['Martillo', 'Sierra', 'Destornillador']
```

---

**Nota:** Si no se especifica un separador, `split()` usa por defecto cualquier secuencia de espacios en blanco, tabuladores y saltos de línea.

---

Aunque aún no lo hemos visto, lo que devuelve `split()` es una *lista* (otro tipo de datos en Python) donde cada elemento es una parte de la cadena de texto original:

```
>>> game = 'piedra-papel-tijera'  
  
>>> type(game.split('-'))  
list
```

---

## Ejercicio

Sabiendo que la longitud de una lista se calcula igual que la *longitud de una cadena de texto*, obtenga el número de palabras que contiene la siguiente cadena de texto:

```
quote = 'Before software can be reusable, it first has to be usable'
```

---

## Limpiar cadenas

Cuando leemos datos del usuario o de cualquier fuente externa de información, es bastante probable que se incluyan en esas cadenas de texto, *caracteres de relleno*<sup>3</sup> al comienzo y al final. Python nos ofrece la posibilidad de eliminar estos caracteres u otros que no nos interesen.

La función `strip()` se utiliza para eliminar caracteres del principio y del final de un «string». También existen variantes de esta función para aplicarla únicamente al comienzo o únicamente al final de la cadena de texto.

Supongamos que debemos procesar un fichero con números de serie de un determinado artículo. Cada línea contiene el valor que nos interesa pero se han «colado» ciertos caracteres de relleno que debemos limpiar:

```
>>> serial_number = '\n\t  \n 48374983274832  \n\n\t  \t  \n'  
  
>>> serial_number.strip()  
'48374983274832'
```

---

**Nota:** Si no se especifican los caracteres a eliminar, `strip()` usa por defecto cualquier combinación de *espacios en blanco, saltos de línea \n y tabuladores \t*.

---

A continuación vamos a hacer «limpieza» por la izquierda (*comienzo*) y por la derecha (*final*) utilizando la función `lstrip()` y `rstrip()` respectivamente:

<sup>3</sup> Se suele utilizar el término inglés «padding» para referirse a estos caracteres.

Lista 6: «Left strip»

```
>>> serial_number.lstrip()
'48374983274832    \n\n\t  \t  \n'
```

Lista 7: «Right strip»

```
>>> serial_number.rstrip()
'\n\t  \n 48374983274832'
```

Como habíamos comentado, también existe la posibilidad de especificar los caracteres que queremos borrar:

```
>>> serial_number.strip('\n')
'\t  \n 48374983274832    \n\n\t  \t  '
```

---

**Importante:** La función `strip()` no modifica la cadena que estamos usando (*algo obvio porque los «strings» son inmutables*) sino que devuelve una nueva cadena de texto con las modificaciones pertinentes.

---

## Realizar búsquedas

Aunque hemos visto que la forma pitónica de saber si una subcadena se encuentra dentro de otra es *a través del operador in*, Python nos ofrece distintas alternativas para realizar búsquedas en cadenas de texto.

Vamos a partir de una variable que contiene un trozo de la canción *Mediterráneo* de *Joan Manuel Serrat* para ejemplificar las distintas opciones que tenemos:

```
>>> lyrics = '''Quizás porque mi niñez
... Sigue jugando en tu playa
... Y escondido tras las cañas
... Duerme mi primer amor
... Llevo tu luz y tu olor
... Por dondequiera que vaya'''
```

Comprobar si una cadena de texto empieza o termina por alguna subcadena:

```
>>> lyrics.startswith('Quizás')
True

>>> lyrics.endswith('Final')
False
```

Encontrar la **primera ocurrencia** de alguna subcadena:

```
>>> lyrics.find('amor')
93

>>> lyrics.index('amor') # Same behaviour?
93
```

Tanto `find()` como `index()` devuelven el **índice** de la primera ocurrencia de la subcadena que estemos buscando, pero se diferencian en su comportamiento cuando la subcadena buscada no existe:

```
>>> lyrics.find('universo')
-1

>>> lyrics.index('universo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Contabilizar el **número de veces que aparece** una subcadena:

```
>>> lyrics.count('mi')
2

>>> lyrics.count('tu')
3

>>> lyrics.count('él')
0
```

## Ejercicio

Dada la siguiente letra<sup>5</sup>, obtenga la misma pero sustituyendo la palabra `voices` por `sounds`:

```
>>> song = '''You look so beautiful in this light
... Your silhouette over me
... The way it brings out the blue in your eyes
... Is the Tenerife sea
... And all of the voices surrounding us here
... They just fade out when you take a breath
... Just say the word and I will disappear
... Into the wilderness'''
```

Utilice para ello únicamente búsqueda, concatenación y troceado de cadenas de texto.

<sup>5</sup> «Tenerife Sea» por Ed Sheeran.

## Reemplazar elementos

Podemos usar la función `replace()` indicando la *subcadena a reemplazar*, la *subcadena de reemplazo* y *cuántas instancias* se deben reemplazar. Si no se especifica este último argumento, la sustitución se hará en todas las instancias encontradas:

```
>>> proverb = 'Quien mal anda mal acaba'  
  
>>> proverb.replace('mal', 'bien')  
'Quien bien anda bien acaba'  
  
>>> proverb.replace('mal', 'bien', 1) # sólo 1 reemplazo  
'Quien bien anda mal acaba'
```

## Mayúsculas y minúsculas

Python nos permite realizar variaciones en los caracteres de una cadena de texto para pasarlos a mayúsculas y/o minúsculas. Veamos las distintas opciones disponibles:

```
>>> proverb = 'quien a buen árbol se arrima Buena Sombra le cobija'  
  
>>> proverb  
'quien a buen árbol se arrima Buena Sombra le cobija'  
  
>>> proverb.capitalize()  
'Quien a buen árbol se arrima buena sombra le cobija'  
  
>>> proverb.title()  
'Quien A Buen Árbol Se Arrima Buena Sombra Le Cobija'  
  
>>> proverb.upper()  
'QUIEN A BUEN ÁRBOL SE ARRIMA BUENA SOMBRA LE COBIJA'  
  
>>> proverb.lower()  
'quien a buen árbol se arrima buena sombra le cobija'  
  
>>> proverb.swapcase()  
'QUIEN A BUEN ÁRBOL SE ARRIMA bUENA sOMBRA LE COBIJA'
```

## Identificando caracteres

Hay veces que recibimos información textual de distintas fuentes de las que necesitamos identificar qué tipo de caracteres contienen. Para ello Python nos ofrece un grupo de funciones.

Veamos **algunas** de estas funciones:

Lista 8: Detectar si todos los caracteres son letras o números

```
>>> 'R2D2'.isalnum()
True
>>> 'C3-PO'.isalnum()
False
```

Lista 9: Detectar si todos los caracteres son números

```
>>> '314'.isnumeric()
True
>>> '3.14'.isnumeric()
False
```

Lista 10: Detectar si todos los caracteres son letras

```
>>> 'abc'.isalpha()
True
>>> 'a-b-c'.isalpha()
False
```

Lista 11: Detectar mayúsculas/minúsculas

```
>>> 'BIG'.isupper()
True
>>> 'small'.islower()
True
>>> 'First Heading'.istitle()
True
```

### 3.3.7 Interpolación de cadenas

En este apartado veremos cómo **interpolar** valores dentro de cadenas de texto utilizando diferentes formatos. Interpolar (en este contexto) significa sustituir una variable por su valor dentro de una cadena de texto.

Veamos los estilos que proporciona Python para este cometido:

Nombre	Símbolo	Soportado
Estilo antiguo	%	>= Python2
Estilo «nuevo»	.format	>= Python2.6
«f-strings»	f ''	>= Python3.6

Aunque aún podemos encontrar código con el **estilo antiguo** y el **estilo nuevo** en el **formateo de cadenas**, vamos a centrarnos en el análisis de los **«f-strings»** que se están utilizando bastante en la actualidad.

#### «f-strings»

Los **f-strings** aparecieron en **Python 3.6** y se suelen usar en código de nueva creación. Es la forma más potente – y en muchas ocasiones más eficiente – de formar cadenas de texto incluyendo valores de otras variables.

La **interpolación** en cadenas de texto es un concepto que existe en la gran mayoría de lenguajes de programación y hace referencia al hecho de sustituir los nombres de variables por sus valores cuando se construye un «string».

Para indicar en Python que una cadena es un «f-string» basta con precederla de una **f** e incluir las variables o expresiones a interpolar entre llaves **{...}**.

Supongamos que disponemos de los datos de una persona y queremos formar una frase de bienvenida con ellos:

```
>>> name = 'Elon Musk'
>>> age = 49
>>> fortune = 43_300

>>> f'Me llamo {name}, tengo {age} años y una fortuna de {fortune} millones'
'Me llamo Elon Musk, tengo 49 años y una fortuna de 43300 millones'
```

**Advertencia:** Si olvidamos poner la **f** delante del «string» no conseguiremos sustitución de variables.

Podría surgir la duda de cómo incluir llaves dentro de la cadena de texto, teniendo en cuenta que las llaves son símbolos especiales para la interpolación de variables. La respuesta es duplicar las llaves:

```
>>> x = 10

>>> f'The variable is {{ x = {x} }}'
'The variable is { x = 10 }'
```

## Formateando cadenas

### Nivel intermedio

Los «f-strings» proporcionan una gran variedad de **opciones de formateado**: ancho del texto, número de decimales, tamaño de la cifra, alineación, etc. Muchas de estas facilidades se pueden consultar en el artículo [Best of Python3.6 f-strings<sup>4</sup>](#)

#### Dando formato a valores enteros:

```
>>> mount_height = 3718

>>> f'{mount_height:10d}'
'      3718'

>>> f'{mount_height:010d}'
'0000003718'
```

#### Dando formato a otras bases:

```
>>> value = 0b10010011
```

(continué en la próxima página)

<sup>4</sup> Escrito por Nirant Kasliwal en Medium.

(provien de la página anterior)

```
>>> f'{value}'  
'147'  
>>> f'{value:b}'  
'10010011'  
  
>>> value = 0o47622  
>>> f'{value}'  
'20370'  
>>> f'{value:o}'  
'47622'  
  
>>> value = 0xab217  
>>> f'{value}'  
'700951'  
>>> f'{value:x}'  
'ab217'
```

## Dando formato a valores flotantes:

```
>>> pi = 3.14159265  
  
>>> f'{pi:f}' # 6 decimales por defecto (se rellenan con ceros si procede)  
'3.141593'  
  
>>> f'{pi:.3f}'  
'3.142'  
  
>>> f'{pi:12f}'  
'      3.141593'  
  
>>> f'{pi:7.2f}'  
'    3.14'  
  
>>> f'{pi:07.2f}'  
'0003.14'  
  
>>> f'{pi:.010f}'  
'3.1415926500'  
  
>>> f'{pi:e}'  
'3.141593e+00'
```

## Alineando valores:

```
>>> text1 = 'how'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> text2 = 'are'
>>> text3 = 'you'

>>> f'{text1:<7s}|{text2:^11s}|{text3:>7s}'
'how      |    are      |    you'

>>> f'{text1:-<7s}|{text2:·^11s}|{text3:->7s}'
'how----|···are···|---you'
```

## Modo «debug»

A partir de Python 3.8, los «f-strings» permiten imprimir el nombre de la variable y su valor, como un atajo para depurar nuestro código. Para ello sólo tenemos que incluir un símbolo = después del nombre de la variable:

```
>>> serie = 'The Simpsons'
>>> imdb_rating = 8.7
>>> num_seasons = 30

>>> f'{serie=}'
"serie='The Simpsons'

>>> f'{imdb_rating=}'
'imdb_rating=8.7'

>>> f'{serie[4:]=}' # incluso podemos añadir expresiones!
"serie[4:]='Simpsons'

>>> f'{imdb_rating / num_seasons=}'
'imdb_rating / num_seasons=0.29'
```

---

## Ejercicio

Dada la variable:

```
e = 2.71828
```

, obtenga los siguientes resultados utilizando «f-strings»:

```
'2.718'
'2.718280'
'    2.72' # 4 espacios en blanco
'2.718280e+00'
```

(continué en la próxima página)

(provien de la página anterior)

```
'00002.7183'  
' 2.71828' # 12 espacios en blanco
```

### 3.3.8 Caracteres Unicode

Python trabaja *por defecto* con caracteres **Unicode**. Eso significa que tenemos acceso a la amplia carta de caracteres que nos ofrece este estándar de codificación.

Supongamos un ejemplo sobre el típico «emoji» de un **cohete** definido [en este cuadro](#):



Figura 9: Representación Unicode del carácter ROCKET

La función `chr()` permite representar un carácter **a partir de su código**:

```
>>> rocket_code = 0x1F680  
>>> rocket = chr(rocket_code)  
>>> rocket  
'🚀'
```

La función `ord()` permite obtener el código (decimal) de un carácter **a partir de su representación**:

```
>>> rocket_code = hex(ord(rocket))  
>>> rocket_code  
'0x1f680'
```

El modificador `\N{ROCKET}` permite representar un carácter **a partir de su nombre**:

```
>>> '\N{ROCKET}'  
'🚀'
```

### 3.3.9 Casos de uso

#### Nivel avanzado

Hemos estado usando muchas funciones de objetos tipo «string» (y de otros tipos previamente). Pero quizás no sabemos aún como podemos descubrir todo lo que podemos hacer con ellos y los **casos de uso** que nos ofrece.

Python proporciona una *función «built-in»* llamada `dir()` para inspeccionar un determinado tipo de objeto:

```
>>> text = 'This is it!'

>>> dir(text)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

(continué en la próxima página)

(provienе de la pgina anterior)

```
'capitalize',
'casefold',
'center',
'count',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'format_map',
'index',
'isalnum',
'isalpha',
'isascii',
'isdecimal',
'isdigit',
'isidentifier',
'islower',
'isnumeric',
'isprintable',
'isspace',
'istitle',
'isupper',
'join',
'ljust',
'lower',
'lstrip',
'maketrans',
'partition',
'replace',
'rfind',
'rindex',
'rjust',
'rpartition',
'rsplit',
'rstrip',
'split',
'splitlines',
'startswith',
'strip',
'swapcase',
'title',
'translate',
'upper',
'zfill']
```

Esto es aplicable tanto a variables como a literales e incluso a tipos de datos (clases) explícitos:

```
>>> dir(10)
['__abs__', '__add__', '__and__', '__bool__', ...
 'imag', 'numerator', 'real', 'to_bytes']

>>> dir(float)
['__abs__', '__add__', '__bool__', '__class__', ...
 'hex', 'imag', 'is_integer', 'real']
```

## EJERCICIOS DE REPASO

1. Escriba un programa en Python que acepte el nombre y los apellidos de una persona y los imprima en orden inverso separados por una coma. Utilice *f-strings* para implementarlo ([solución](#)).

Entrada: nombre=Sergio; apellidos=Delgado Quintero

Salida: Delgado Quintero, Sergio

2. Escriba un programa en Python que acepte una ruta remota de recurso samba, y lo separe en nombre(IP) del equipo y ruta (**solución**).

Entrada: //1.1.1.1/eoi/python

Salida: equipo=1.1.1.1; ruta=/eoi/python

3. Escriba un programa en Python que acepte un «string» con los 8 dígitos de un NIF, y calcule su dígito de control (solución).

Entrada: 12345678

Salida: 12345678Z

4. Escriba un programa en Python que acepte un entero  $n$  y compute el valor de  $n + nn + nnn$  (**solución**).

Entrada: 5

Salida: 615

5. Escriba un programa en Python que acepte una palabra en castellano y calcule una métrica que sea el número total de caracteres de la palabra multiplicado por el número total de vocales que contiene la palabra (**solución**).

Entrada: ordenador

Salida: 36

## AMPLIAR CONOCIMIENTOS

- [A Guide to the Newer Python String Format Techniques](#)
- [Strings and Character Data in Python](#)
- [How to Convert a Python String to int](#)
- [Your Guide to the Python print<> Function](#)
- [Basic Input, Output, and String Formatting in Python](#)
- [Unicode & Character Encodings in Python: A Painless Guide](#)
- [Python String Formatting Tips & Best Practices](#)
- [Python 3's f-Strings: An Improved String Formatting Syntax](#)
- [Splitting, Concatenating, and Joining Strings in Python](#)
- [Conditional Statements in Python](#)
- [Python String Formatting Best Practices](#)

# CAPÍTULO 4

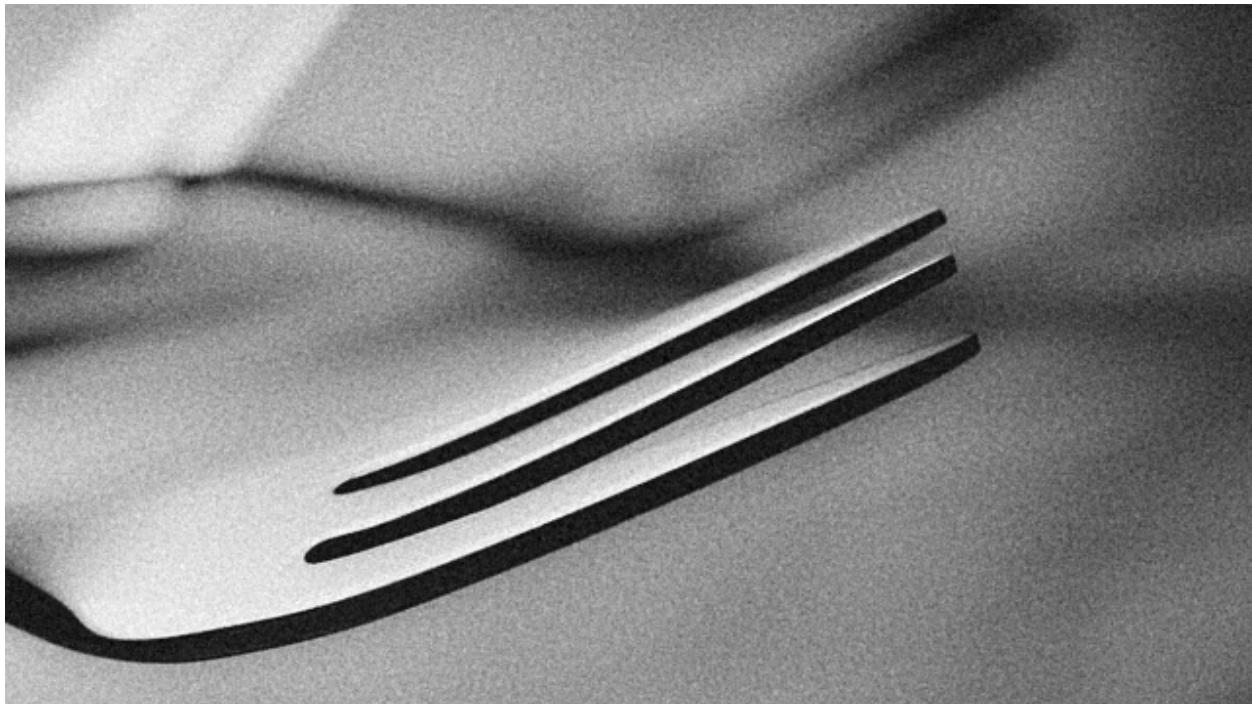
---

## Control de flujo

---

Todo programa informático está formado por *instrucciones* que se ejecutan en forma secuencial de «arriba» a «abajo», de igual manera que leeríamos un libro. Este orden constituye el llamado **flujo** del programa. Es posible modificar este flujo secuencial para que tome *bifurcaciones* o repita ciertas instrucciones. Las sentencias que nos permiten hacer estas modificaciones se engloban en el *control de flujo*.

## 4.1 Condicionales



En esta sección veremos la sentencia condicional `if` y las distintas variantes que puede asumir, pero antes de eso introduciremos algunas cuestiones generales de *escritura de código*.<sup>1</sup>

### 4.1.1 Definición de bloques

A diferencia de otros lenguajes que utilizan llaves para definir los bloques de código, cuando Guido Van Rossum *creó el lenguaje* quiso evitar estos caracteres por considerarlos innecesarios. Es por ello que en Python los bloques de código se definen a través de **espacios en blanco, preferiblemente 4**.<sup>2</sup> En términos técnicos se habla del **tamaño de indentación**.

---

**Consejo:** Esto puede resultar extraño e incómodo a personas que vienen de otros lenguajes de programación pero desaparece rápido y se siente natural a medida que se escribe código.

---

<sup>1</sup> Foto original de portada por ali nafezarefi en Unsplash.

<sup>2</sup> Reglas de indentación definidas en PEP 8

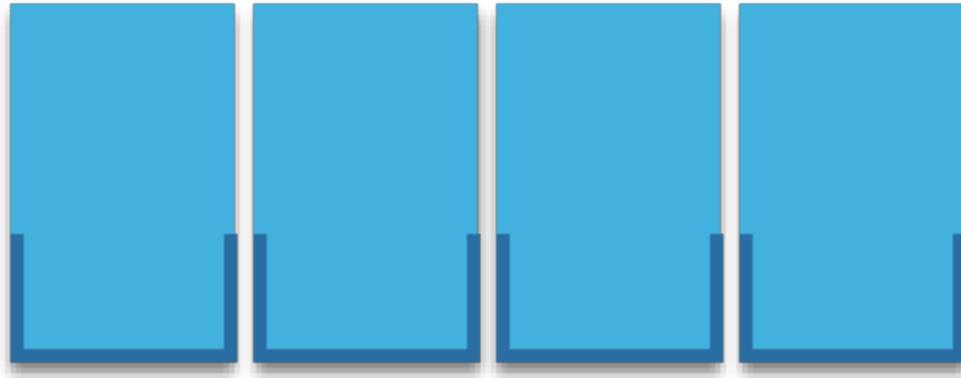


Figura 1: Python recomienda 4 espacios en blanco para indentar

### 4.1.2 Comentarios

Los comentarios son anotaciones que podemos incluir en nuestro programa y que nos permiten aclarar ciertos aspectos del código. Estas indicaciones son ignoradas por el intérprete de Python.

Los comentarios se incluyen usando el símbolo almohadilla # y comprenden hasta el final de la línea.

Lista 1: Comentario en bloque

```
# Universe age expressed in days
universe_age = 13800 * (10 ** 6) * 365
```

Los comentarios también pueden aparecer en la misma línea de código, aunque [la guía de estilo de Python](#) no aconseja usarlos en demasia:

Lista 2: Comentario en línea

```
stock = 0    # Release additional articles
```

### 4.1.3 Ancho del código

Los programas suelen ser más legibles cuando las líneas no son excesivamente largas. La longitud máxima de línea recomendada por [la guía de estilo de Python](#) es de **80 caracteres**.

Sin embargo, esto genera una cierta polémica hoy en día, ya que los tamaños de pantalla han aumentado y las resoluciones son mucho mayores que hace años. Así las líneas de más de 80 caracteres se siguen visualizando correctamente. Hay personas que son más estrictas en este límite y otras más flexibles.

En caso de que queramos **romper una línea de código** demasiado larga, tenemos dos

opciones:

1. Usar la *barra invertida* \:

```
>>> factorial = 4 * 3 * 2 * 1  
  
>>> factorial = 4 * \  
...      3 * \  
...      2 * \  
...      1
```

2. Usar los *paréntesis* (...):

```
>>> factorial = 4 * 3 * 2 * 1  
  
>>> factorial = (4 *  
...      3 *  
...      2 *  
...      1)
```

### 4.1.4 La sentencia if

La sentencia condicional en Python (al igual que en muchos otros lenguajes de programación) es **if**. En su escritura debemos añadir una **expresión de comparación** terminando con dos puntos al final de la línea. Veamos un ejemplo:

```
>>> temperature = 40  
  
>>> if temperature > 35:  
...     print('Aviso por alta temperatura')  
  
Aviso por alta temperatura
```

---

**Nota:** Nótese que en Python no es necesario incluir paréntesis ( y ) al escribir condiciones. Hay veces que es recomendable por claridad o por establecer prioridades.

---

En el caso anterior se puede ver claramente que la condición se cumple y por tanto se ejecuta la instrucción que tenemos dentro del cuerpo de la condición. Pero podría no ser así. Para controlar ese caso existe la sentencia **else**. Veamos el mismo ejemplo anterior pero añadiendo esta variante:

```
>>> temperature = 20  
  
>>> if temperature > 35:
```

(continué en la próxima página)

(provine de la página anterior)

```

...     print('Aviso por alta temperatura')
... else:
...     print('Parámetros normales')
...
Parámetros normales

```

Podríamos tener incluso condiciones dentro de condiciones, lo que se viene a llamar técnicamente **condiciones anidadas**<sup>3</sup>. Veamos un ejemplo ampliando el caso anterior:

```

>>> temperature = 28

>>> if temperature < 20:
...     if temperature < 10:
...         print('Nivel azul')
...     else:
...         print('Nivel verde')
... else:
...     if temperature < 30:
...         print('Nivel naranja')
...     else:
...         print('Nivel rojo')
...
Nivel naranja

```

Python nos ofrece una mejora en la escritura de condiciones anidadas cuando aparecen consecutivamente un `else` y un `if`. Podemos sustituirlos por la sentencia `elif`:

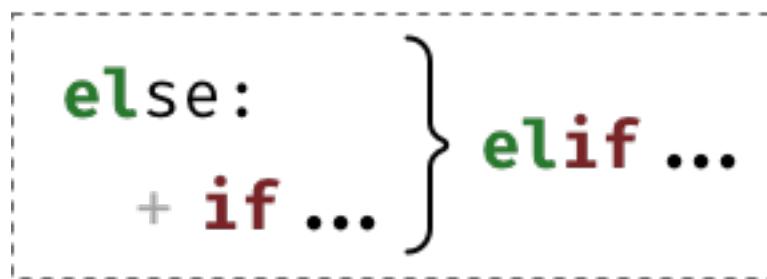


Figura 2: Construcción de la sentencia `elif`

Apliquemos esta mejora al código del ejemplo anterior:

```

>>> temperature = 28

>>> if temperature < 20:

```

(continué en la próxima página)

<sup>3</sup> El anidamiento (o «nesting») hace referencia a incorporar sentencias unas dentro de otras mediante la inclusión de diversos niveles de profundidad (indentación).

(provien de la página anterior)

```
...     if temperature < 10:
...         print('Nivel azul')
...     else:
...         print('Nivel verde')
... elif temperature < 30:
...     print('Nivel naranja')
... else:
...     print('Nivel rojo')
...
Nivel naranja
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/wd58B4t>

## 4.1.5 Operadores de comparación

Cuando escribimos condiciones debemos incluir alguna expresión de comparación. Para usar estas expresiones es fundamental conocer los operadores que nos ofrece Python:

Operador	Símbolo
Igualdad	<code>==</code>
Desigualdad	<code>!=</code>
Menor que	<code>&lt;</code>
Menor o igual que	<code>&lt;=</code>
Mayor que	<code>&gt;</code>
Mayor o igual que	<code>&gt;=</code>

A continuación vamos a ver una serie de ejemplos con expresiones de comparación. Téngase en cuenta que estas expresiones habría que incluirlas dentro de la sentencia condicional en el caso de que quisiéramos tomar una acción concreta:

```
# Asignación de valor inicial
>>> value = 8

>>> value == 8
True

>>> value != 8
False

>>> value < 12
True
```

(continué en la próxima página)

(provine de la página anterior)

```
>>> value <= 7
False

>>> value > 4
True

>>> value >= 9
False
```

Podemos escribir condiciones más complejas usando los operadores lógicos:

- and
- or
- not

```
# Asignación de valor inicial
>>> x = 8

>>> x > 4 or x > 12 # True or False
True

>>> x < 4 or x > 12 # False or False
False

>>> x > 4 and x > 12 # True and False
False

>>> x > 4 and x < 12 # True and True
True

>>> not(x != 8) # not False
True
```

Python ofrece la posibilidad de ver si un valor está entre dos límites de manera directa. Así, por ejemplo, para descubrir si `value` está entre `4` y `12` haríamos:

```
>>> 4 <= value <= 12
True
```

### Nota:

1. Una expresión de comparación siempre devuelve un valor *booleano*, es decir `True` o `False`.

2. El uso de paréntesis, en función del caso, puede aclarar la expresión de comparación.

---

## Ejercicio

Dada una variable `year` con un valor entero, compruebe si dicho año es **bisiesto** o no lo es.

**i** Un año es bisiesto en el calendario Gregoriano, si es divisible entre 4 y no divisible entre 100, o bien si es divisible entre 400. Puedes hacer la comprobación en [esta lista de años bisiestos](#).

## Ejemplo

- Entrada: 2008
  - Salida: Es un año bisiesto
- 

## «Booleanos» en condiciones

Cuando queremos preguntar por la **veracidad** de una determinada variable «booleana» en una condición, la primera aproximación que parece razonable es la siguiente:

```
>>> is_cold = True

>>> if is_cold == True:
...     print('Coge chaqueta')
... else:
...     print('Usa camiseta')
...
Coge chaqueta
```

Pero podemos *simplificar* esta condición tal que así:

```
>>> if is_cold:
...     print('Coge chaqueta')
... else:
...     print('Usa camiseta')
...
Coge chaqueta
```

Hemos visto una comparación para un valor «booleano» verdadero (`True`). En el caso de que la comparación fuera para un valor falso lo haríamos así:

```
>>> is_cold = False

>>> if not is_cold: # Equivalente a if is_cold == False
```

(continué en la próxima página)

(provien de la página anterior)

```
...     print('Usa camiseta')
... else:
...     print('Coge chaqueta')
...
Usa camiseta
```

De hecho, si lo pensamos, estamos reproduciendo bastante bien el *lenguaje natural*:

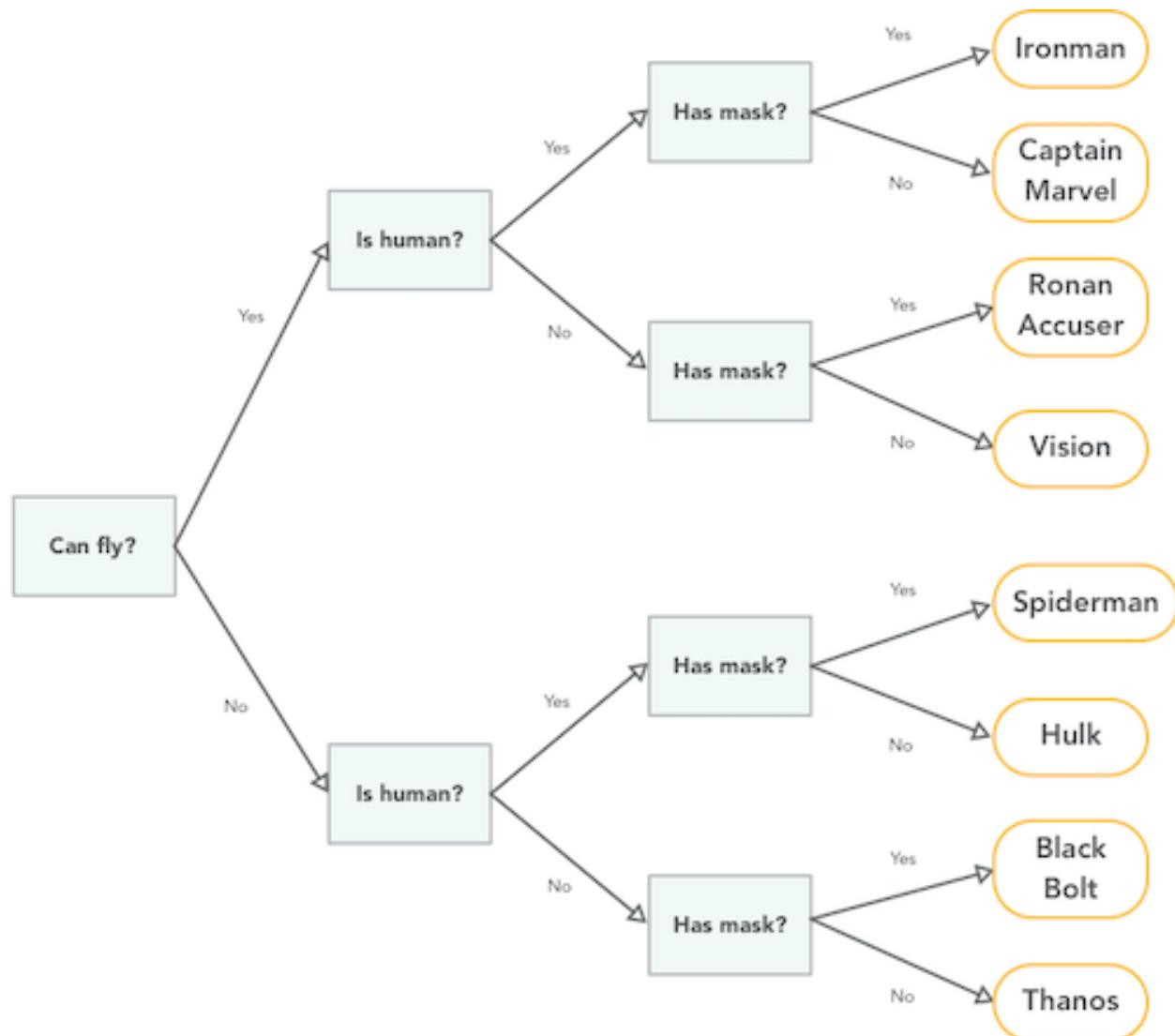
- Si hace frío, coge chaqueta.
- Si no hace frío, usa camiseta.

---

### Ejercicio

Escriba un programa que permita adivinar un personaje de Marvel en base a las tres preguntas siguientes:

1. ¿Puede volar?
2. ¿Es humano?
3. ¿Tiene máscara?



## Ejemplo

- Entrada: can\_fly = True, is\_human = True y has\_mask = True
- Salida: Ironman

Es una especie de Akinator para personajes de Marvel...

---

## Valor nulo

### Nivel intermedio

`None` es un valor especial de Python que almacena el **valor nulo**<sup>4</sup>. Veamos cómo se comporta al incorporarlo en condiciones de veracidad:

```
>>> value = None

>>> if value:
...     print('Value has some useful value')
... else:
...     # value podría contener None, False (u otro)
...     print('Value seems to be void')
...
Value seems to be void
```

Para distinguir `None` de los valores propiamente booleanos, se recomienda el uso del operador `is`. Veamos un ejemplo en el que tratamos de averiguar si un valor **es nulo**:

```
>>> value = None

>>> if value is None:
...     print('Value is clearly None')
... else:
...     # value podría contener True, False (u otro)
...     print('Value has some useful value')
...
Value is clearly void
```

De igual forma, podemos usar esta construcción para el caso contrario. La forma «pitónica» de preguntar si algo **no es nulo** es la siguiente:

```
>>> value = 99

>>> if value is not None:
...     print(f'{value=}')
...
value=99
```

<sup>4</sup> Lo que en otros lenguajes se conoce como `nil`, `null`, `nothing`.

### 4.1.6 Operador morsa

#### Nivel avanzado

A partir de Python 3.8 se incorpora el **operador morsa**<sup>5</sup> que permite unificar **sentencias de asignación dentro de expresiones**. Su nombre proviene de la forma que adquiere :=

Supongamos un ejemplo en el que computamos el perímetro de una circunferencia, indicando al usuario que debe incrementarlo siempre y cuando no llegue a un mínimo establecido.

#### Versión tradicional

```
>>> radius = 4.25
... perimeter = 2 * 3.14 * radius
... if perimeter < 100:
...     print('Increase radius to reach minimum perimeter')
...     print('Actual perimeter: ', perimeter)
...
Increase radius to reach minimum perimeter
Actual perimeter: 26.69
```

#### Versión con operador morsa

```
>>> radius = 4.25
... if (perimeter := 2 * 3.14 * radius) < 100:
...     print('Increase radius to reach minimum perimeter')
...     print('Actual perimeter: ', perimeter)
...
Increase radius to reach minimum perimeter
Actual perimeter: 26.69
```

---

**Consejo:** Como hemos comprobado, el operador morsa permite realizar asignaciones dentro de expresiones, lo que, en muchas ocasiones, permite obtener un código más compacto. Sería conveniente encontrar un equilibrio entre la expresividad y la legibilidad.

---

---

<sup>5</sup> Se denomina así porque el operador := tiene similitud con los colmillos de una morsa.

## EJERCICIOS DE REPASO

1. Escriba un programa en Python que acepte la opción de dos jugadoras en Piedra-Papel-Tijera y decida el resultado ([solución](#)).

Entrada: persona1=piedra; persona2=papel

Salida: Gana persona2: El papel envuelve a la piedra

2. Escriba un programa en Python que acepte 3 números y calcule el mínimo ([solución](#)).

Entrada: 7, 4, 9

Salida: 4

3. Escriba un programa en Python que acepte un país (como «string») y muestre por pantalla su bandera (como «emoji»). *Puede restringirlo a un conjunto limitado de países* ([solución](#)).

Entrada: Italia

Salida: 

4. Escriba un programa en Python que acepte 3 códigos de teclas y muestre por pantalla la acción que se lleva a cabo en sistemas Ubuntu Linux ([solución](#)).

Entrada: tecla1=Ctrl; tecla2=Alt; tecla3=Del;

Salida: Log out

5. Escriba un programa en Python que acepte edad, peso, pulso y plaquetas, y determine si una persona cumple con estos [requisitos](#) para donar sangre.

Entrada: edad=34; peso=81; heartbeat=70; plaquetas=150000

Salida: Apto para donar sangre

## AMPLIAR CONOCIMIENTOS

- [How to Use the Python or Operator](#)
- [Conditional Statements in Python \(if/elif/else\)](#)

## 4.2 Bucles



Cuando queremos hacer algo más de una vez, necesitamos recurrir a un **bucle**. En esta sección veremos las distintas sentencias en Python que nos permiten repetir un bloque de código.<sup>1</sup>

### 4.2.1 La sentencia while

El primer mecanismo que existe en Python para repetir instrucciones es usar la sentencia `while`. La semántica tras esta sentencia es: «Mientras se cumpla la condición haz algo». Veamos un sencillo bucle que muestra por pantalla los números del 1 al 4:

```
>>> value = 1

>>> while value <= 4:
...     print(value)
...     value += 1
...
1
2
3
4
```

---

<sup>1</sup> Foto original de portada por Gary Lopater en Unsplash.

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/RgM2HYn>

La condición del bucle se comprueba en cada nueva repetición. En este caso chequeamos que la variable `value` sea menor o igual que 4. Dentro del cuerpo del bucle estamos incrementando esa variable en 1 unidad.

## Romper un bucle while

Python ofrece la posibilidad de *romper* o finalizar un bucle *antes de que se cumpla la condición de parada*. Supongamos un ejemplo en el que estamos buscando el primer número múltiplo de 3 yendo desde 20 hasta 1:

```
>>> num = 20

>>> while num >= 1:
...     if num % 3 == 0:
...         print(num)
...         break
...     num -= 1
...
18
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/wfrKnHl>

Como hemos visto en este ejemplo, `break` nos permite finalizar el bucle una vez que hemos encontrado nuestro objetivo: el primer múltiplo de 3. Pero si no lo hubiéramos encontrado, el bucle habría seguido decrementando la variable `num` hasta valer 0, y la condición del bucle `while` hubiera resultado falsa.

## Comprobar la rotura

### Nivel intermedio

Python nos ofrece la posibilidad de **detectar si el bucle ha acabado de forma ordinaria**, esto es, ha finalizado por no cumplirse la condición establecida. Para ello podemos hacer uso de la sentencia `else` como parte del propio bucle. Si el bucle `while` finaliza normalmente (sin llamada a `break`) el flujo de control pasa a la sentencia opcional `else`.

Veamos un ejemplo en el que tratamos de encontrar un múltiplo de 9 en el rango [1, 8] (es obvio que no sucederá):

```
>>> num = 8

>>> while num >= 1:
...     if num % 9 == 0:
...         print(f'{num} is a multiple of 9!')
...         break
...     num -= 1
... else:
...     print('No multiples of 9 found!')
...
No multiples of 9 found!
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/CgYQFiA>

## Continuar un bucle

### Nivel intermedio

Hay situaciones en las que, en vez de romper un bucle, nos interesa **saltar adelante hacia la siguiente repetición**. Para ello Python nos ofrece la sentencia `continue` que hace precisamente eso, descartar el resto del código del bucle y saltar a la siguiente iteración.

Veamos un ejemplo en el que usaremos esta estrategia para mostrar todos los números en el rango [1, 20] ignorando aquellos que sean múltiplos de 3:

```
>>> num = 21

>>> while num >= 1:
...     num -= 1
...     if num % 3 == 0:
...         continue
...     print(num, end=' ',) # Evitar salto de línea
...
20, 19, 17, 16, 14, 13, 11, 10, 8, 7, 5, 4, 2, 1,
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/YgYQ3m6>

## Bucle infinito

Si no establecemos correctamente la **condición de parada** o bien el valor de alguna variable está fuera de control, es posible que lleguemos a una situación de bucle infinito, del que nunca podamos salir. Veamos un ejemplo de esto:

```
>>> num = 1

>>> while num != 10:
...     num += 2
...
# CTRL-C
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

El problema que surje es que la variable `num` toma los valores 1, 3, 5, 7, 9, 11, ... por lo que nunca se cumple la condición de parada del bucle. Esto hace que repitamos «eternamente» la instrucción de incremento.

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/AfrZroa>

Una posible solución a este error es reescribir la condición de parada en el bucle:

```
>>> num = 1

>>> while num < 10:
...     num += 2
...
```

---

**Truco:** Para abortar una situación de *bucle infinito* podemos pulsar en el teclado la combinación CTRL-C. Se puede ver reflejado en el intérprete de Python por `KeyboardInterrupt`.

---

## Ejercicio

Escriba un programa que calcule la *distancia hamming* entre dos *cadenas de texto* de la misma longitud.

### Ejemplo

- Entrada: 0001010011101 y 0000110010001
  - Salida: 4
-

### 4.2.2 La sentencia for

Python permite recorrer aquellos tipos de datos que sean **iterables**, es decir, que admitan *iterar*<sup>2</sup> sobre ellos. Algunos ejemplos de tipos y estructuras de datos que permiten ser iteradas (*recorridas*) son: cadenas de texto, listas, diccionarios, ficheros, etc. La sentencia **for** nos permite realizar esta acción.

A continuación se plantea un ejemplo en el que vamos a recorrer (iterar) una cadena de texto:

```
>>> word = 'Python'

>>> for letter in word:
...     print(letter)
...
P
y
t
h
o
n
```

La clave aquí está en darse cuenta que el bucle va tomando, en cada iteración, cada uno de los elementos de la variable que especifiquemos. En este caso concreto **letter** va tomando cada una de las letras que existen en **word**, porque una cadena de texto está formada por elementos que son caracteres.

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/Pft6R2e>

---

**Importante:** La variable que utilizamos en el bucle **for** para ir tomando los valores puede tener **cualquier nombre**. Al fin y al cabo es una variable que definimos según nuestras necesidades. Tener en cuenta que se suele usar un nombre en singular.

---

### Romper un bucle for

Una sentencia **break** dentro de un **for** rompe el bucle, *igual que veíamos* para los bucles **while**. Veamos un ejemplo con el código anterior. En este caso vamos a recorrer una cadena de texto y pararemos el bucle cuando encontramos una letra *t* minúscula:

```
>>> word = 'Python'
```

(continué en la próxima página)

---

<sup>2</sup> Realizar cierta acción varias veces. En este caso la acción es tomar cada elemento.

(provine de la página anterior)

```
>>> for letter in word:
...     if letter == 't':
...         break
...     print(letter)
...
P
y
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/zfyqkbJ>

---

**Truco:** Tanto la *comprobación de rotura de un bucle* como la *continuación a la siguiente iteración* se llevan a cabo del mismo modo que hemos visto con los bucles de tipo `while`.

---

## Ejercicio

Dada una cadena de texto, indique el número de vocales que tiene.

### Ejemplo

- Entrada: Supercalifragilisticocoespialidoso
  - Salida: 15
- 

## Secuencias de números

Es muy habitual hacer uso de secuencias de números en bucles. Python no tiene una instrucción específica para ello. Lo que sí aporta es una función `range()` que devuelve un *flujo de números* en el rango especificado. Una de las grandes ventajas es que la «lista» generada no se construye explícitamente, sino que cada valor se genera bajo demanda. Esta técnica mejora el consumo de recursos, especialmente en términos de memoria.

La técnica para la generación de secuencias de números es muy similar a la utilizada en los «*slices*» de cadenas de texto. En este caso disponemos de la función `range(start, stop, step)`:

- **start**: Es *opcional* y tiene valor por defecto **0**.
- **stop**: es *obligatorio* (siempre se llega a 1 menos que este valor).
- **step**: es *opcional* y tiene valor por defecto **1**.

`range()` devuelve un *objeto iterable*, así que iremos obteniendo los valores paso a paso con una sentencia `for ... in3`. Veamos diferentes ejemplos de uso:

**Rango:** [0, 1, 2]

```
>>> for i in range(0, 3):
...     print(i)
...
0
1
2

>>> for i in range(3):
...     print(i)
...
0
1
2
```

**Rango:** [1, 3, 5]

```
>>> for i in range(1, 6, 2):
...     print(i)
...
1
3
5
```

**Rango:** [2, 1, 0]

```
>>> for i in range(2, -1, -1):
...     print(i)
...
2
1
0
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/vfywE45>

---

**Truco:** Se suelen utilizar nombres de variables `i`, `j`, `k` para lo que se denominan **contadores**. Este tipo de variables toman valores numéricos enteros como en los ejemplos anteriores. No conviene generalizar el uso de estas variables a situaciones en las que, claramente, tenemos la posibilidad de asignar un nombre semánticamente más significativo. Esto viene de tiempos

---

<sup>3</sup> O convertir el objeto a una secuencia como una lista.

antiguos en FORTRAN donde `i` era la primera letra que tenía valor entero por defecto.

---

## Ejercicio

Determine si un número dado es un [número primo](#).

*No es necesario implementar ningún algoritmo en concreto. La idea es probar los números menores al dado e ir viendo si las divisiones tienen resto cero o no.*

¿Podrías optimizar tu código? ¿Realmente es necesario probar con tantos divisores?

## Ejemplo

- Entrada: 11
  - Salida: Es primo
- 

## Usando el guión bajo

### Nivel avanzado

Hay situaciones en las que **no necesitamos usar la variable** que toma valores en el rango, sino que únicamente queremos repetir una acción un número determinado de veces.

Para estos casos se suele recomendar usar el **guión bajo `_`** como **nombre de variable**, que da a entender que no estamos usando esta variable de forma explícita:

```
>>> for _ in range(10):
...     print('Repeat me 10 times!')
...
Repeat me 10 times!
```

### 4.2.3 Bucles anidados

Como ya vimos en las *sentencias condicionales*, el *anidamiento* es una técnica por la que incluimos distintos niveles de encapsulamiento de sentencias, unas dentro de otras, con mayor nivel de profundidad. En el caso de los bucles también es posible hacer anidamiento.

Veamos un ejemplo de 2 bucles anidados en el que generamos todas las tablas de multiplicar:

```
>>> for i in range(1, 10):
...     for j in range(1, 10):
...         result = i * j
...         print(f'{i} * {j} = {result}')
...
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
4 x 6 = 24
4 x 7 = 28
```

(continuó en la próxima página)

(provien de la página anterior)

```
4 x 8 = 32
4 x 9 = 36
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
6 x 1 = 6
6 x 2 = 12
6 x 3 = 18
6 x 4 = 24
6 x 5 = 30
6 x 6 = 36
6 x 7 = 42
6 x 8 = 48
6 x 9 = 54
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72
9 x 1 = 9
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
```

(continué en la próxima página)

(provien de la página anterior)

```
9 * 8 = 72  
9 * 9 = 81
```

Lo que está ocurriendo en este código es que, para cada valor que toma la variable `i`, la otra variable `j` toma todos sus valores. Como resultado tenemos una combinación completa de los valores en el rango especificado.

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/vfyewv>

---

### Nota:

- Podemos añadir todos los niveles de anidamiento que queramos. Eso sí, hay que tener en cuenta que cada nuevo nivel de anidamiento supone un importante aumento de la **complejidad ciclomática** de nuestro código, lo que se traduce en mayores tiempos de ejecución.
  - Los bucles anidados también se pueden aplicar en la sentencia `while`.
- 

---

### Ejercicio

Imprima los 100 primeros números de la sucesión de Fibonacci:  
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

---

---

### EJERCICIOS DE REPASO

1. Escriba un programa en Python que realice las siguientes 9 multiplicaciones. ¿Nota algo raro? ([solución](#))

$$\begin{aligned} & 1 \cdot 1 \\ & 11 \cdot 11 \\ & 111 \cdot 111 \\ & \vdots \\ & 111111111 \cdot 111111111 \end{aligned}$$

2. Escriba un programa en Python que acepte una cadena de texto e indique si todos sus caracteres son alfabéticos ([solución](#)).

Entrada: hello-world

Salida: Se han encontrado caracteres no alfabéticos

3. Escriba un programa en Python que acepte un número entero  $n$  y realice el siguiente cálculo de productos sucesivos ([solución](#)):

$$\prod_{i=1}^n x_i^2 = x_0^2 \cdot x_1^2 \cdot x_2^2 \cdot \dots \cdot x_n^2$$

4. Escriba un programa en Python que acepte dos cadenas de texto y compute el [producto cartesiano](#) letra a letra entre ellas ([solución](#)).

Entrada: cadena1=abc; cadena2=123

Salida: a1 a2 a3 b1 b2 b3 c1 c2 c3

5. Escriba un programa en Python que acepte dos valores enteros ( $x$  e  $y$ ) que representarán un punto (objetivo) en el plano. El programa simulará el movimiento de un «caballo» de ajedrez moviéndose de forma alterna: 2 posiciones en  $x + 1$  posición en  $y$ . El siguiente movimiento que toque sería para moverse 1 posición en  $x + 2$  posiciones en  $y$ . El programa deberá ir mostrando los puntos por los que va pasando el «caballo» hasta llegar al punto objetivo ([solución](#)).

Entrada: objetivo\_x=7; objetivo\_y=8;

Salida: (0, 0) (1, 2) (3, 3) (4, 5) (6, 6) (7, 8)

## AMPLIAR CONOCIMIENTOS

- The Python range() Function
- How to Write Pythonic Loops
- For Loops in Python (Definite Iteration)
- Python «while» Loops (Indefinite Iteration)



# CAPÍTULO 5

---

## Estructuras de datos

---

Si bien ya hemos visto una sección sobre *Tipos de datos*, podríamos hablar de tipos de datos más complejos en Python que se constituyen en **estructuras de datos**. Si pensamos en estos elementos como *átomos*, las estructuras de datos que vamos a ver sería *moléculas*. Es decir, combinamos los tipos básicos de formas más complejas. De hecho, esta distinción se hace en el [Tutorial oficial de Python](#). Trataremos distintas estructuras de datos como listas, tuplas, diccionarios y conjuntos.

## 5.1 Listas



Las listas permiten **almacenar objetos** mediante un **orden definido** y con posibilidad de duplicados. Las listas son estructuras de datos **mutables**, lo que significa que podemos añadir, eliminar o modificar sus elementos.<sup>1</sup>

### 5.1.1 Creando listas

Una lista está compuesta por *cero o más elementos*. En Python debemos escribir estos elementos separados por *comas* y dentro de *corchetes*. Veamos algunos ejemplos de listas:

```
>>> empty_list = []

>>> languages = ['Python', 'Ruby', 'Javascript']

>>> fibonacci = [0, 1, 1, 2, 3, 5, 8, 13]

>>> data = ['Tenerife', {'cielo': 'limpio', 'temp': 24}, 3718, (28.2933947, -16.
↪5226597)]
```

---

**Nota:** Una lista puede contener tipos de **datos heterogéneos**, lo que la hace una estructura

<sup>1</sup> Foto original de portada por [Mike Arney](#) en Unsplash.

de datos muy versátil.

---

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/Ofiare>

### 5.1.2 Conversión

Para convertir otros tipos de datos en una lista podemos usar la función `list()`:

```
>>> # conversión desde una cadena de texto
>>> list('Python')
['P', 'y', 't', 'h', 'o', 'n']
```

Si nos fijamos en lo que ha pasado, al convertir la cadena de texto Python se ha creado una lista con 6 elementos, donde cada uno de ellos representa un carácter de la cadena. Podemos *extender* este comportamiento a cualquier otro tipo de datos que permita ser iterado (*iterables*).

### Lista vacía

Existe una manera particular de usar `list()` y es no pasarle ningún argumento. En este caso estaremos queriendo convertir el «vacío» en una lista, con lo que obtendremos una *lista vacía*:

```
>>> list()
[]
```

---

**Truco:** Para crear una lista vacía, se suele recomendar el uso de `[]` frente a `list()`, no sólo por ser más *pitónico* sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

---

### 5.1.3 Operaciones con listas

#### Obtener un elemento

Igual que en el caso de las *cadenas de texto*, podemos obtener un elemento de una lista a través del **índice** (lugar) que ocupa. Veamos un ejemplo:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping[0]
'Agua'

>>> shopping[1]
'Huevos'

>>> shopping[2]
'Aceite'

>>> shopping[-1] # acceso con índice negativo
'Aceite'
```

El *índice* que usemos para acceder a los elementos de una lista tiene que estar comprendido entre los límites de la misma. Si usamos un índice antes del comienzo o después del final obtendremos un error (*excepción*):

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> shopping[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

### Trocear una lista

El troceado de listas funciona de manera totalmente análoga al *troceado de cadenas*. Veamos algunos ejemplos:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[0:3]
['Agua', 'Huevos', 'Aceite']

>>> shopping[:3]
['Agua', 'Huevos', 'Aceite']

>>> shopping[2:4]
['Aceite', 'Sal']
```

(continué en la próxima página)

(provine de la página anterior)

```
>>> shopping[-1:-4:-1]
['Limón', 'Sal', 'Aceite']

>>> # Equivale a invertir la lista
>>> shopping[::-1]
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

En el troceado de listas, a diferencia de lo que ocurre al obtener elementos, no debemos preocuparnos por acceder a *índices inválidos* (fuera de rango) ya que Python los restringirá a los límites de la lista:

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[10:]
[]

>>> shopping[-100:2]
['Agua', 'Huevos']

>>> shopping[2:100]
['Aceite', 'Sal', 'Limón']
```

---

**Importante:** Ninguna de las operaciones anteriores modifican la lista original, simplemente devuelven una lista nueva.

---

## Invertir una lista

Python nos ofrece, al menos, tres mecanismos para invertir los elementos de una lista:

**Conservando la lista original:** Mediante *troceado* de listas con *step* negativo:

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[::-1]
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

**Conservando la lista original:** Mediante la función `reversed()`:

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
```

(continué en la próxima página)

(provine de la página anterior)

```
>>> list(reversed(shopping))
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

**Modificando la lista original:** Utilizando la función `reverse()` (nótese que es sin «*d*» al final):

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.reverse()

>>> shopping
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

## Añadir al final de la lista

Una de las operaciones más utilizadas en listas es añadir elementos al final de las mismas. Para ello Python nos ofrece la función `append()`. Se trata de un método *destructivo* que modifica la lista original:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.append('Atún')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Atún']
```

## Creando desde vacío

Una forma muy habitual de trabajar con listas es empezar con una vacía e ir añadiendo elementos poco a poco. Se podría hablar de un **patrón creación**.

Supongamos un ejemplo en el que queremos construir una lista con los números pares del 1 al 20:

```
>>> even_numbers = []

>>> for i in range(20):
...     if i % 2 == 0:
...         even_numbers.append(i)
... 
```

(continué en la próxima página)

(provine de la página anterior)

```
>>> even_numbers
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/2fiS9Ax>

## Añadir en cualquier posición de una lista

Ya hemos visto cómo añadir elementos al final de una lista. Sin embargo, Python ofrece una función `insert()` que vendría a ser una generalización de la anterior, para incorporar elementos en cualquier posición. Simplemente debemos especificar el índice de inserción y el elemento en cuestión. También se trata de una función *destructiva*<sup>2</sup>:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.insert(1, 'Jamón')

>>> shopping
['Agua', 'Jamón', 'Huevos', 'Aceite']

>>> shopping.insert(3, 'Queso')

>>> shopping
['Agua', 'Jamón', 'Huevos', 'Queso', 'Aceite']
```

---

**Nota:** El índice que especificamos en la función `insert()` lo podemos intepretar como la posición *delante* (a la izquierda) de la cual vamos a colocar el nuevo valor en la lista.

---

Al igual que ocurría con el *troceado de listas*, en este tipo de inserciones no obtendremos un error si especificamos índices fuera de los límites de la lista. Estos se ajustarán al principio o al final en función del valor que indiquemos:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.insert(100, 'Mermelada')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Mermelada']
```

(continué en la próxima página)

---

<sup>2</sup> Cuando hablamos de que una función/método es «destructiva/o» significa que modifica la lista (objeto) original, no que la destruye.

(provien de la página anterior)

```
>>> shopping.insert(-100, 'Arroz')

>>> shopping
['Arroz', 'Agua', 'Huevos', 'Aceite', 'Mermelada']
```

---

**Consejo:** Aunque es posible utilizar `insert()` para añadir **elementos al final de una lista**, siempre se recomienda usar `append()` por su mayor legibilidad:

```
>>> values = [1, 2, 3]
>>> values.append(4)
>>> values
[1, 2, 3, 4]

>>> values = [1, 2, 3]
>>> values.insert(len(values), 4) # don't do it!
>>> values
[1, 2, 3, 4]
```

---

## Repetir elementos

Al igual que con las *cadenas de texto*, el operador `*` nos permite repetir los elementos de una lista:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping * 3
['Agua',
 'Huevos',
 'Aceite',
 'Agua',
 'Huevos',
 'Aceite',
 'Agua',
 'Huevos',
 'Aceite']
```

## Combinar listas

Python nos ofrece dos aproximaciones para combinar listas:

**Conservando la lista original:** Mediante el operador + o +=:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> fruitshop = ['Naranja', 'Manzana', 'Piña']

>>> shopping + fruitshop
['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']
```

**Modificando la lista original:** Mediante la función extend():

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> fruitshop = ['Naranja', 'Manzana', 'Piña']

>>> shopping.extend(fruitshop)

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']
```

Hay que tener en cuenta que `extend()` funciona adecuadamente si pasamos una **lista como argumento**. En otro caso, quizás los resultados no sean los esperados. Veamos un ejemplo:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.extend('Limón')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'L', 'í', 'm', 'ó', 'n']
```

El motivo es que `extend()` «recorre» (o itera) sobre cada uno de los elementos del objeto en cuestión. En el caso anterior, al ser una cadena de texto, está formada por caracteres. De ahí el resultado que obtenemos.

Se podría pensar en el uso de `append()` para combinar listas. La realidad es que no funciona exactamente como esperamos; la segunda lista se añadiría como una *sublista* de la principal:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> fruitshop = ['Naranja', 'Manzana', 'Piña']

>>> shopping.append(fruitshop)

>>> shopping
['Agua', 'Huevos', 'Aceite', ['Naranja', 'Manzana', 'Piña']]
```

## Modificar una lista

Del mismo modo que se *accede a un elemento* utilizando su índice, también podemos modificarlo:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping[0]
'Agua'

>>> shopping[0] = 'Jugo'

>>> shopping
['Jugo', 'Huevos', 'Aceite']
```

En el caso de acceder a un *índice no válido* de la lista, incluso para modificar, obtendremos un error:

```
>>> shopping[100] = 'Chocolate'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

## Modificar con troceado

No sólo es posible modificar un elemento de cada vez, sino que podemos asignar valores a trozos de una lista:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[1:4]
['Huevos', 'Aceite', 'Sal']

>>> shopping[1:4] = ['Atún', 'Pasta']

>>> shopping
['Agua', 'Atún', 'Pasta', 'Limón']
```

---

**Nota:** La lista que asignamos no necesariamente debe tener la misma longitud que el trozo que sustituimos.

---

## Borrar elementos

Python nos ofrece, al menos, cuatro formas para borrar elementos en una lista:

**Por su índice:** Mediante la función `del()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> del(shopping[3])

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Limón']
```

**Por su valor:** Mediante la función `remove()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.remove('Sal')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Limón']
```

**Advertencia:** Si existen valores duplicados, la función `remove()` sólo borrará la primera ocurrencia.

**Por su índice (con extracción):** Las dos funciones anteriores `del()` y `remove()` efectivamente borran el elemento indicado de la lista, pero no «devuelven»<sup>3</sup> nada. Sin embargo, Python nos ofrece la función `pop()` que además de borrar, nos «recupera» el elemento; algo así como una *extracción*. Lo podemos ver como una combinación de *acceso + borrado*:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.pop()
'Limón'

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal']

>>> shopping.pop(2)
'Aceite'
```

(continué en la próxima página)

<sup>3</sup> Más adelante veremos el comportamiento de las funciones. Devolver o retornar un valor es el resultado de aplicar una función.

(provine de la página anterior)

```
>>> shopping  
['Agua', 'Huevos', 'Sal']
```

---

**Nota:** Si usamos la función `pop()` sin pasarle ningún argumento, por defecto usará el índice `-1`, es decir, el último elemento de la lista. Pero también podemos indicarle el índice del elemento a extraer.

---

**Por su rango:** Mediante troceado de listas:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']  
  
>>> shopping[1:4] = []  
  
>>> shopping  
['Agua', 'Limón']
```

## Borrado completo de la lista

Python nos ofrece, al menos, dos formas para borrar una lista por completo:

1. Utilizando la función `clear()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']  
  
>>> shopping.clear() # Borrado in-situ  
  
>>> shopping  
[]
```

2. «Reinicializando» la lista a vacío con `[]`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']  
  
>>> shopping = [] # Nueva zona de memoria  
  
>>> shopping  
[]
```

---

**Nota:** La diferencia entre ambos métodos tiene que ver con cuestiones internas de gestión de memoria y de rendimiento.

---

## Encontrar un elemento

Si queremos descubrir el **índice** que corresponde a un determinado valor dentro la lista podemos usar la función **index()** para ello:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']  
  
>>> shopping.index('Huevos')  
1
```

Tener en cuenta que si el elemento que buscamos no está en la lista, obtendremos un error:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']  
  
>>> shopping.index('Pollo')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: 'Pollo' is not in list
```

---

**Nota:** Si buscamos un valor que existe más de una vez en una lista, la función **index()** sólo nos devolverá el índice de la primera ocurrencia.

---

## Pertenencia de un elemento

Si queremos comprobar la existencia de un determinado elemento en una lista, podríamos buscar su índice, pero la forma **pitónica** de hacerlo es utilizar el operador **in**:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']  
  
>>> 'Aceite' in shopping  
True  
  
>>> 'Pollo' in shopping  
False
```

---

**Nota:** El operador **in** siempre devuelve un valor booleano, es decir, verdadero o falso.

---

---

## Ejercicio

Determine si una cadena de texto dada es un **isograma**, es decir, no se repite ninguna letra.

Ejemplos válidos de isogramas:

- *lumberjacks*
  - *background*
  - *downstream*
  - *six-year-old*
- 

### Número de ocurrencias

Para contar cuántas veces aparece un determinado valor dentro de una lista podemos usar la función `count()`:

```
>>> sheldon_greeting = ['Penny', 'Penny', 'Penny']

>>> sheldon_greeting.count('Howard')
0

>>> sheldon_greeting.count('Penny')
3
```

### Convertir lista a cadena de texto

Dada una lista, podemos convertirla a una cadena de texto, uniendo todos sus elementos mediante algún **separador**. Para ello hacemos uso de la función `join()` con la siguiente estructura:



Figura 1: Estructura de llamada a la función `join()`

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> ', '.join(shopping)
'Agua,Huevos,Aceite,Sal,Limón'
```

(continué en la próxima página)

(provien de la página anterior)

```
>>> ' '.join(shopping)
'Agua Huevos Aceite Sal Limón'
```

```
>>> '|'.join(shopping)
'Agua|Huevos|Aceite|Sal|Limón'
```

Hay que tener en cuenta que `join()` sólo funciona si *todos sus elementos son cadenas de texto*:

```
>>> ', '.join([1, 2, 3, 4, 5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found
```

---

**Truco:** Esta función `join()` es realmente la **opuesta** a la de `split()` para *dividir una cadena*.

---

## Ejercicio

Consiga la siguiente transformación:

12/31/20 → 31-12-2020

---

## Ordenar una lista

Python proporciona, al menos, dos formas de ordenar los elementos de una lista:

**Conservando lista original:** Mediante la función `sorted()` que devuelve una nueva lista ordenada:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> sorted(shopping)
['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']
```

**Modificando la lista original:** Mediante la función `sort()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.sort()
```

(continué en la próxima página)

(provien de la página anterior)

```
>>> shopping  
['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']
```

**Ambos métodos** admiten un *parámetro booleano* `reverse` para indicar si queremos que la ordenación se haga en **sentido inverso**:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']  
  
>>> sorted(shopping, reverse=True)  
['Sal', 'Limón', 'Huevos', 'Agua', 'Aceite']
```

## Longitud de una lista

Podemos conocer el número de elementos que tiene una lista con la función `len()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']  
  
>>> len(shopping)  
5
```

## Iterar sobre una lista

Al igual que *hemos visto con las cadenas de texto*, también podemos *iterar* sobre los elementos de una lista utilizando la sentencia `for`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']  
  
>>> for product in shopping:  
...     print(product)  
  
Agua  
Huevos  
Aceite  
Sal  
Limón
```

---

**Nota:** También es posible usar la sentencia `break` en este tipo de bucles para abortar su ejecución en algún momento que nos interese.

---

## Iterar usando enumeración

Hay veces que no sólo nos interesa «visitar» cada uno de los elementos de una lista, sino que también queremos **saber su índice** dentro de la misma. Para ello Python nos ofrece la función `enumerate()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> for i, product in enumerate(shopping):
...     print(i, product)
...
0 Agua
1 Huevos
2 Aceite
3 Sal
4 Limón
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/TfiuIZ0>

## Iterar sobre múltiples listas

Python ofrece la posibilidad de iterar sobre **múltiples listas en paralelo** utilizando la función `zip()`:

```
>>> shopping = ['Agua', 'Aceite', 'Arroz']
>>> details = ['mineral natural', 'de oliva virgen', 'basmati']

>>> for product, detail in zip(shopping, details):
...     print(product, detail)
...
Agua mineral natural
Aceite de oliva virgen
Arroz basmati
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/lfioilG>

---

**Nota:** En el caso de que las listas no tengan la misma longitud, la función `zip()` realiza la combinación hasta que se agota la lista más corta.

---

Dado que `zip()` produce un *iterador*, si queremos obtener una **lista explícita** con la combinación en paralelo de las listas, debemos construir dicha lista de la siguiente manera:

```
>>> shopping = ['Agua', 'Aceite', 'Arroz']
>>> details = ['mineral natural', 'de oliva virgen', 'basmati']

>>> list(zip(shopping, details))
[('Agua', 'mineral natural'),
 ('Aceite', 'de oliva virgen'),
 ('Arroz', 'basmati')]
```

---

### Ejercicio

Dados dos vectores (listas) de la misma dimensión, utilice la función `zip()` para calcular su producto escalar.

### Ejemplo

- Entrada:

```
v1 = [4, 3, 8, 1]
v2 = [9, 2, 7, 3]
```

- Salida: 101

$$v1 \times v2 = [4 \cdot 9 + 3 \cdot 2 + 8 \cdot 7 + 1 \cdot 3] = 101$$

---

### 5.1.4 Cuidado con las copias

#### Nivel intermedio

Las listas son estructuras de datos *mutables* y esta característica nos obliga a tener cuidado cuando realizamos copias de listas, ya que la modificación de una de ellas puede afectar a la otra.

Veamos un ejemplo sencillo:

```
>>> original_list = [4, 3, 7, 1]

>>> copy_list = original_list

>>> original_list[0] = 15

>>> original_list
[15, 3, 7, 1]

>>> copy_list
[15, 3, 7, 1]
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/pfi5PC5>

---

**Nota:** A través de *Python Tutor* se puede ver claramente el motivo de por qué ocurre esto. Dado que las variables «apuntan» a la misma zona de memoria, al modificar una de ellas, el cambio también se ve reflejado en la otra.

---

Una  **posible solución** a este problema es hacer una «copia dura». Para ello Python proporciona la función `copy()`:

```
>>> original_list = [4, 3, 7, 1]

>>> copy_list = original_list.copy()

>>> original_list[0] = 15

>>> original_list
[15, 3, 7, 1]

>>> copy_list
[4, 3, 7, 1]
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/Dfi6oLk>

---

**Truco:** En el caso de que estemos trabajando con listas que contienen elementos mutables, debemos hacer uso de la función `deepcopy()` dentro del módulo `copy` de la librería estándar.

---

## 5.1.5 Veracidad múltiple

Si bien podemos usar *sentencias condicionales* para comprobar la veracidad de determinadas expresiones, Python nos ofrece dos funciones «built-in» con las que podemos evaluar si se cumplen **todas** las condiciones `all()` o si se cumple **alguna** condición `any()`. Estas funciones trabajan sobre iterables, y el caso más evidente es una **lista**.

Supongamos un ejemplo en el que queremos comprobar si una determinada palabra cumple las siguientes condiciones:

- Su longitud total es mayor que 4.
- Empieza por «p».
- Contiene, al menos, una «y».

## Aprende Python

---

Veamos la **versión clásica**:

```
>>> word = 'python'

>>> if len(word) > 4 and word.startswith('p') and word.count('y') >= 1:
...     print('Cool word!')
... else:
...     print('No thanks')
...
Cool word!
```

Veamos la **versión con veracidad múltiple** usando `all()`, donde se comprueba que se cumplan **todas** las expresiones:

```
>>> word = 'python'

>>> enough_length = len(word) > 4           # True
>>> right_beginning = word.startswith('p')    # True
>>> min_ys = word.count('y') >= 1           # True

>>> is_cool_word = all([enough_length, right_beginning, min_ys])

>>> if is_cool_word:
...     print('Cool word!')
... else:
...     print('No thanks')
...
Cool word!
```

Veamos la **versión con veracidad múltiple** usando `any()`, donde se comprueba que se cumpla **alguna** expresión:

```
>>> word = 'yeah'

>>> enough_length = len(word) > 4           # False
>>> right_beginning = word.startswith('p')    # False
>>> min_ys = word.count('y') >= 1           # True

>>> is_fine_word = any([enough_length, right_beginning, min_ys])

>>> if is_fine_word:
...     print('Fine word!')
... else:
...     print('No thanks')
...
Fine word!
```

**Consejo:** Este enfoque puede ser interesante cuando se manejan muchas condiciones o bien cuando queremos separar las condiciones y agruparlas en una única lista.

## 5.1.6 Listas por comprensión

### Nivel intermedio

Las **listas por comprensión** establecen una técnica para crear listas de forma más **compacta** basándose en el concepto matemático de conjuntos definidos por comprensión.

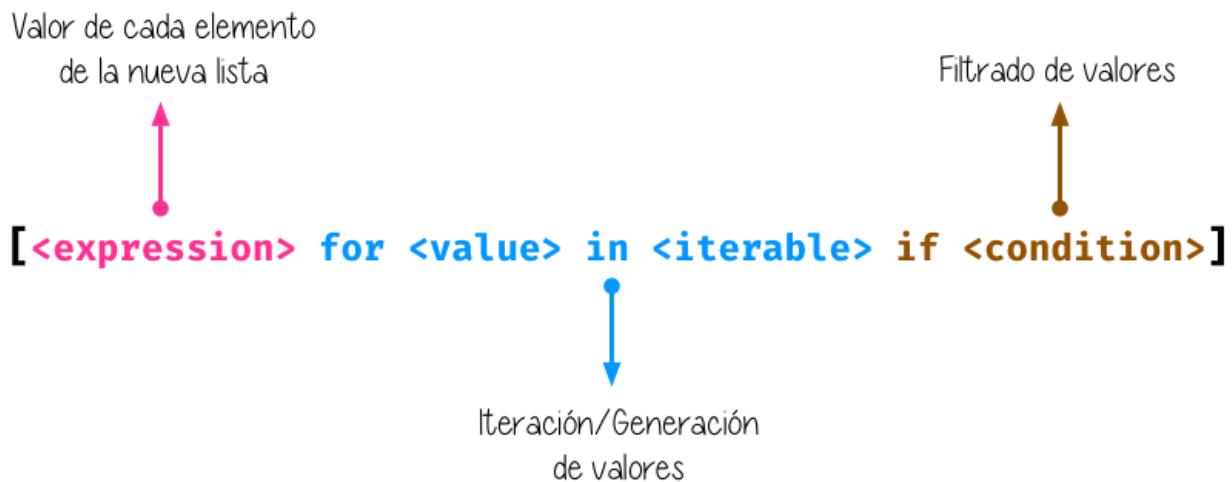


Figura 2: Estructura de una lista por comprensión

En primer lugar veamos un ejemplo en el que convertimos una cadena de texto con valores numéricos en una lista con los mismos valores pero convertidos a enteros. En su **versión clásica** haríamos algo tal que así:

```
>>> values = '32,45,11,87,20,48'

>>> int_values = []

>>> for value in values.split(','):
...     int_value = int(value)
...     int_values.append(int_value)
...

>>> int_values
[32, 45, 11, 87, 20, 48]
```

Ahora veamos el código utilizando una **lista por comprensión**:

```
>>> values = '32,45,11,87,20,48'  
  
>>> int_values = [int(value) for value in values.split(',')]  
  
>>> int_values  
[32, 45, 11, 87, 20, 48]
```

## Condiciones en comprensiones

También existe la posibilidad de incluir condiciones en las **listas por comprensión**.

Continuando con el ejemplo anterior, supongamos que sólo queremos crear la lista con aquellos valores que empiecen por el dígito 4:

```
>>> values = '32,45,11,87,20,48'  
  
>>> int_values = [int(v) for v in values.split(',') if v.startswith('4')]  
  
>>> int_values  
[45, 48]
```

## Anidamiento en comprensiones

### Nivel avanzado

En la iteración que usamos dentro de la lista por comprensión es posible usar *bucles anidados*.

Veamos un ejemplo en el que generamos todas las combinaciones de una serie de valores:

```
>>> values = '32,45,11,87,20,48'  
>>> svalues = values.split(',')  
  
>>> combinations = [f'{v1}{x}{v2}' for v1 in svalues for v2 in svalues]  
  
>>> combinations  
['32x32',  
 '32x45',  
 '32x11',  
 '32x87',  
 '32x20',  
 '32x48',  
 '45x32',  
 '45x45',  
 ...]
```

(continué en la próxima página)

(provine de la página anterior)

```
'48x45',
'48x11',
'48x87',
'48x20',
'48x48']
```

**Consejo:** Las listas por comprensión son una herramienta muy potente y nos ayuda en muchas ocasiones, pero hay que tener cuidado de no generar **expresiones excesivamente complejas**. En estos casos es mejor una *aproximación clásica*.

### Ejercicio

Utilizando listas por comprensión, cree una lista que contenga el resultado de aplicar la función  $f(x) = 3x + 2$  para  $x \in [0, 20]$ .

**Salida esperada:** [2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50, 53, 56, 59]

### 5.1.7 sys.argv

Cuando queramos ejecutar un programa Python desde **línea de comandos**, tendremos la posibilidad de acceder a los argumentos de dicho programa. Para ello se utiliza una lista que la encontramos dentro del módulo **sys** y que se denomina **argv**:

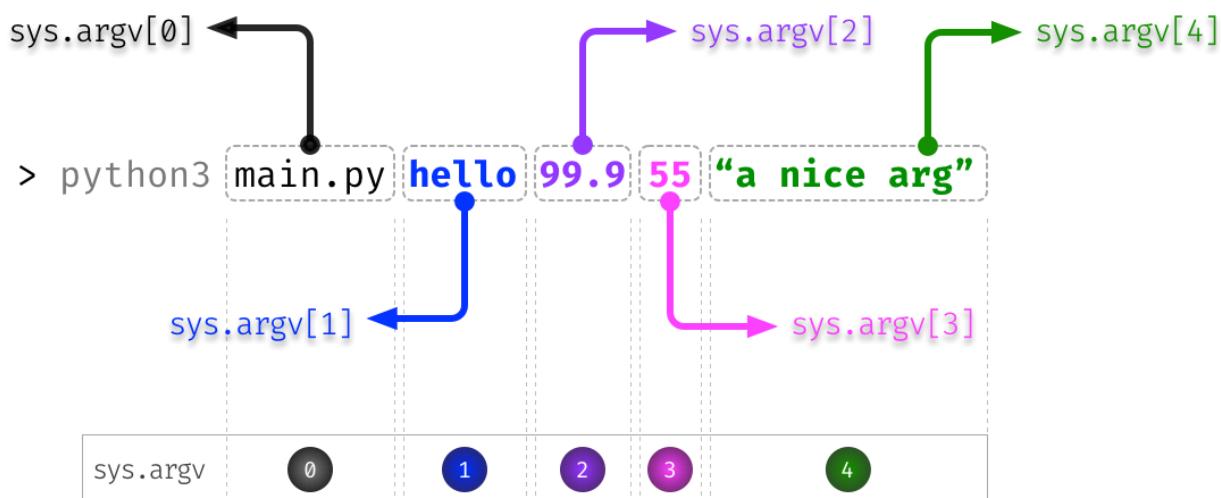


Figura 3: Acceso a parámetros en línea de comandos

Veamos un ejemplo de código en el que simulamos el paso de parámetros recogido en la figura anterior:

get-args.py

```
1 import sys
2
3 filename = sys.argv[0]
4 arg1 = sys.argv[1]
5 arg2 = float(sys.argv[2])
6 arg3 = int(sys.argv[3])
7 arg4 = sys.argv[4]
8
9 print(f'{arg1=}')
10 print(f'{arg2=}')
11 print(f'{arg3=}')
12 print(f'{arg4=}')
```

Si lo ejecutamos obtenemos lo siguiente:

```
$ python3 get-args.py hello 99.9 55 "a nice arg"
arg1='hello'
arg2=99.9
arg3=55
arg4='a nice arg'
```

### 5.1.8 Funciones matemáticas

Python nos ofrece, entre otras<sup>4</sup>, estas tres funciones matemáticas básicas que se pueden aplicar sobre listas.

**Suma de todos los valores:** Mediante la función `sum()`:

```
>>> data = [5, 3, 2, 8, 9, 1]
>>> sum(data)
28
```

**Mínimo de todos los valores:** Mediante la función `min()`:

```
>>> data = [5, 3, 2, 8, 9, 1]
>>> min(data)
1
```

<sup>4</sup> Existen multitud de paquetes científicos en Python para trabajar con listas o vectores numéricos. Una de las más famosas es la librería `Numpy`.

**Máximo de todos los valores:** Mediante la función `max()`:

```
>>> data = [5, 3, 2, 8, 9, 1]
>>> max(data)
9
```

---

## Ejercicio

Lea *desde línea de comandos* una serie de números y obtenga la media de dichos valores (*muestre el resultado con 2 decimales*).

La llamada se haría de la siguiente manera:

```
$ python3 avg.py 32 56 21 99 12 17
```

Plantilla de código para el programa:

```
import sys

# En values tendremos una lista con los valores (como strings)
values = sys.argv[1:]

# Su código debajo de aquí
```

---

## Ejemplo

- Entrada: 32 56 21 99 12 17
  - Salida: 39.50
- 

### 5.1.9 Listas de listas

#### Nivel intermedio

Como ya hemos visto en varias ocasiones, las listas son estructuras de datos que pueden contener elementos heterogéneos. Estos elementos pueden ser a su vez listas.

A continuación planteamos un ejemplo del mundo deportivo. Un equipo de fútbol suele tener una disposición en el campo organizada en líneas de jugadores. En aquella alineación con la que España ganó la copa del mundo en 2010 había una disposición 4-3-3 con los siguientes jugadores:

Veamos una posible representación de este equipo de fútbol usando una lista compuesta de listas. Primero definimos cada una de las líneas:



Figura 4: Lista de listas (como equipo de fútbol)

```
>>> goalkeeper = 'Casillas'
>>> defenders = ['Capdevila', 'Piqué', 'Puyol', 'Ramos']
>>> midfielders = ['Xabi', 'Busquets', 'X. Alonso']
>>> forwards = ['Iniesta', 'Villa', 'Pedro']
```

Y ahora las juntamos en una única lista:

```
>>> team = [goalkeeper, defenders, midfielders, forwards]

>>> team
['Casillas',
 ['Capdevila', 'Piqué', 'Puyol', 'Ramos'],
 ['Xabi', 'Busquets', 'X. Alonso'],
 ['Iniesta', 'Villa', 'Pedro']]
```

Podemos comprobar el acceso a distintos elementos:

```
>>> team[0] # portero
'Casillas'

>>> team[1][0] # lateral izquierdo
'Capdevila'

>>> team[2] # centrocampistas
['Xabi', 'Busquets', 'X. Alonso']

>>> team[3][1] # delantero centro
'Villa'
```

## Ejercicio

Escriba un programa que permita multiplicar únicamente matrices de 2 filas por 2 columnas. Veamos un ejemplo concreto:

```
A = [[6, 4], [8, 9]]
B = [[3, 2], [1, 7]]
```

El producto  $\mathbb{P} = A \times B$  se calcula siguiendo la multiplicación de matrices tal y como se indica a continuación:

$$\begin{aligned}\mathbb{P} &= \begin{pmatrix} 6_{[00]} & 4_{[01]} \\ 8_{[10]} & 9_{[11]} \end{pmatrix} \times \begin{pmatrix} 3_{[00]} & 2_{[01]} \\ 1_{[10]} & 7_{[11]} \end{pmatrix} = \\ &\quad \begin{pmatrix} 6 \cdot 3 + 4 \cdot 1 & 6 \cdot 2 + 4 \cdot 7 \\ 8 \cdot 3 + 9 \cdot 1 & 8 \cdot 2 + 9 \cdot 7 \end{pmatrix} = \begin{pmatrix} 22 & 40 \\ 33 & 79 \end{pmatrix}\end{aligned}$$

### EJERCICIOS DE REPASO

1. Escriba un programa en Python que acepte una lista de valores numéricos y obtenga su valor máximo **sin utilizar** la función «built-in» `max()` ([solución](#)).

Entrada: [6, 3, 9, 2, 10, 31, 15, 7]

Salida: 2

2. Escriba un programa en Python que acepte una lista y elimine sus elementos duplicados ([solución](#)).

Entrada: ["this", "is", "a", "real", "real", "real", "story"]

Salida: ["this", "is", "a", "real", "story"]

3. Escriba un programa en Python que acepte una lista – que puede contener sublistas (sólo en 1 nivel de anidamiento) – y genere otra lista «aplanada» ([solución](#)).

Entrada: [0, 10, [20, 30], 40, 50, [60, 70, 80], [90, 100, 110, 120]]

Salida: [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120]

4. Escriba un programa en Python que acepte una lista y genere otra lista eliminando los elementos duplicados consecutivos ([solución](#)).

Entrada: [0, 0, 1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 8, 9, 4, 4]

Salida: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 4]

5. Escriba un programa en Python que acepte una lista de listas representando una matriz numérica y compute la suma de los elementos de la diagonal principal ([solución](#)).

Entrada: [[4, 6, 1], [2, 9, 3], [1, 7, 7]]

Salida: 20

### AMPLIAR CONOCIMIENTOS

- [Linked Lists in Python: An Introduction](#)
- [Python Command Line Arguments](#)
- [Sorting Data With Python](#)
- [When to Use a List Comprehension in Python](#)
- [Using the Python zip\(\) Function for Parallel Iteration](#)
- [Lists and Tuples in Python](#)
- [How to Use sorted\(\) and sort\(\) in Python](#)
- [Using List Comprehensions Effectively](#)

## 5.2 Tuplas



El concepto de **tupla** es muy similar al de *lista*. Aunque hay algunas diferencias menores, lo fundamental es que, mientras una *lista* es mutable y se puede modificar, una *tupla* no admite cambios y por lo tanto, es **inmutable**.<sup>1</sup>

### 5.2.1 Creando tuplas

Podemos pensar en crear tuplas tal y como *lo hacíamos con listas*, pero usando **paréntesis** en lugar de *corchetes*:

```
>>> empty_tuple = ()  
  
>>> tenerife_geoloc = (28.46824, -16.25462)  
  
>>> three_wise_men = ('Melchor', 'Gaspar', 'Baltasar')
```

<sup>1</sup> Foto original de portada por engin akyurt en Unsplash.

### Tuplas de un elemento

Hay que prestar especial atención cuando vamos a crear una **tupla de un único elemento**. La intención primera sería hacerlo de la siguiente manera:

```
>>> one_item_tuple = ('Papá Noel')

>>> one_item_tuple
'Papá Noel'

>>> type(one_item_tuple)
str
```

Realmente, hemos creado una variable de tipo **str** (cadena de texto). Para crear una tupla de un elemento debemos añadir una **coma** al final:

```
>>> one_item_tuple = ('Papá Noel',)

>>> one_item_tuple
('Papá Noel',)

>>> type(one_item_tuple)
tuple
```

### Tuplas sin paréntesis

Según el caso, hay veces que nos podemos encontrar con tuplas que no llevan paréntesis. Quizás no está tan extendido, pero a efectos prácticos tiene el mismo resultado. Veamos algunos ejemplos de ello:

```
>>> one_item_tuple = 'Papá Noel',

>>> three_wise_men = 'Melchor', 'Gaspar', 'Baltasar'

>>> tenerife_geoloc = 28.46824, -16.25462
```

### 5.2.2 Modificar una tupla

Como ya hemos comentado previamente, las tuplas con estructuras de datos **inmutables**. Una vez que las creamos con un valor, no podemos modificarlas. Veamos qué ocurre si lo intentamos:

```
>>> three_wise_men = 'Melchor', 'Gaspar', 'Baltasar'  
  
>>> three_wise_men[0] = 'Tom Hanks'  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

### 5.2.3 Conversión

Para convertir otros tipos de datos en una tupla podemos usar la función `tuple()`:

```
>>> shopping = ['Agua', 'Aceite', 'Arroz']  
  
>>> tuple(shopping)  
('Agua', 'Aceite', 'Arroz')
```

Esta conversión es válida para aquellos tipos de datos que sean *iterables*: cadenas de caracteres, listas, diccionarios, conjuntos, etc. Un ejemplo que no funciona es intentar convertir un número en una tupla:

```
>>> tuple(5)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'int' object is not iterable
```

El uso de la función `tuple()` sin argumentos equivale a crear una tupla vacía:

```
>>> tuple()  
()
```

---

**Truco:** Para crear una tupla vacía, se suele recomendar el uso de `()` frente a `tuple()`, no sólo por ser más *pitónico* sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

---

### 5.2.4 Operaciones con tuplas

Con las tuplas podemos realizar *todas las operaciones que vimos con listas salvo las que conlleven una modificación «in-situ» de la misma:*

- `reverse()`
- `append()`
- `extend()`
- `remove()`
- `clear()`
- `sort()`

### 5.2.5 Desempaquetado de tuplas

El **desempaquetado** es una característica de las tuplas que nos permite *asignar una tupla a variables independientes*:

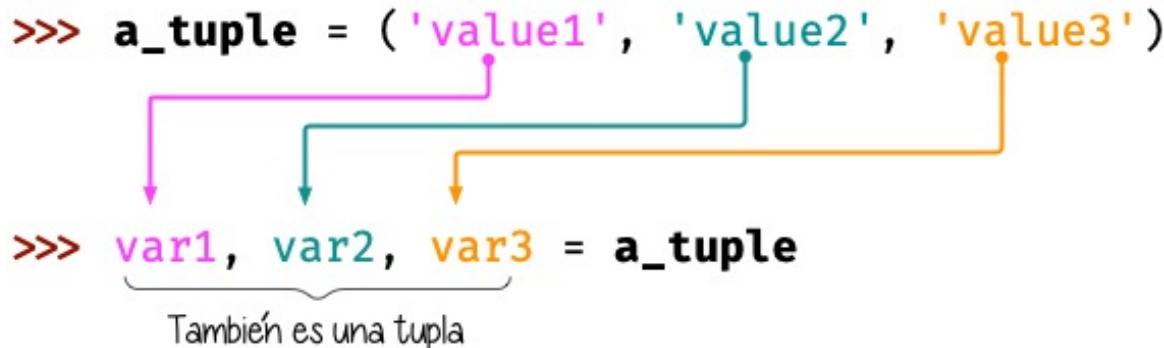


Figura 5: Desempaquetado de tuplas

Veamos un ejemplo con código:

```
>>> three_wise_men = ('Melchor', 'Gaspar', 'Baltasar')

>>> king1, king2, king3 = three_wise_men

>>> king1
'Melchor'
>>> king2
'Gaspar'
```

(continué en la próxima página)

(provine de la página anterior)

```
>>> king3  
'Baltasar'
```

Python proporciona la función «built-in» `divmod()` que devuelve el cociente y el resto de una división usando una única llamada. Lo interesante (para el caso que nos ocupa) es que se suele utilizar el desempaquetado de tuplas para obtener los valores:

```
>>> quotient, remainder = divmod(7, 3)  
  
>>> quotient  
2  
>>> remainder  
1
```

## Intercambio de valores

A través del desempaquetado de tuplas podemos llevar a cabo *el intercambio de los valores de dos variables* de manera directa:

```
>>> value1 = 40  
>>> value2 = 20  
  
>>> value1, value2 = value2, value1  
  
>>> value1  
20  
>>> value2  
40
```

---

**Nota:** A priori puede parecer que esto es algo «natural», pero en la gran mayoría de lenguajes de programación no es posible hacer este intercambio de forma «directa» ya que necesitamos recurrir a una tercera variable «auxiliar» como almacén temporal en el paso intermedio de traspaso de valores.

## Desempaquetado extendido

No tenemos que ceñirnos a realizar desempaquetado uno a uno. También podemos extenderlo e indicar ciertos «grupos» de elementos mediante el operador \*.

Veamos un ejemplo:

```
>>> ranking = ('G', 'A', 'R', 'Y', 'W')

>>> head, *body, tail = ranking

>>> head
'G'

>>> body
['A', 'R', 'Y']

>>> tail
'W'
```

## Desempaquetado genérico

El desempaquetado de tuplas es extensible a cualquier tipo de datos que sea **iterable**. Veamos algunos ejemplos de ello.

Sobre cadenas de texto:

```
>>> oxygen = 'O2'
>>> first, last = oxygen
>>> first, last
('O', '2')

>>> text = 'Hello, World!'
>>> head, *body, tail = text
>>> head, body, tail
('H', ['e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r', 'l', 'd'], '!')
```

Sobre listas:

```
>>> writer1, writer2, writer3 = ['Virginia Woolf', 'Jane Austen', 'Mary Shelley']
>>> writer1, writer2, writer3
('Virginia Woolf', 'Jane Austen', 'Mary Shelley')

>>> text = 'Hello, World!'
>>> word1, word2 = text.split()
>>> word1, word2
('Hello,', 'World!')
```

## 5.2.6 ¿Tuplas por comprensión?

Los tipos de datos mutables (*listas, diccionarios y conjuntos*) sí permiten comprensiones pero no así los tipos de datos inmutables como *cadenas de texto y tuplas*.

Si intentamos crear una **tupla por comprensión** utilizando paréntesis alrededor de la expresión, vemos que no obtenemos ningún error al ejecutarlo:

```
>>> myrange = (number for number in range(1, 6))
```

Sin embargo no hemos conseguido una tupla por comprensión sino un generador:

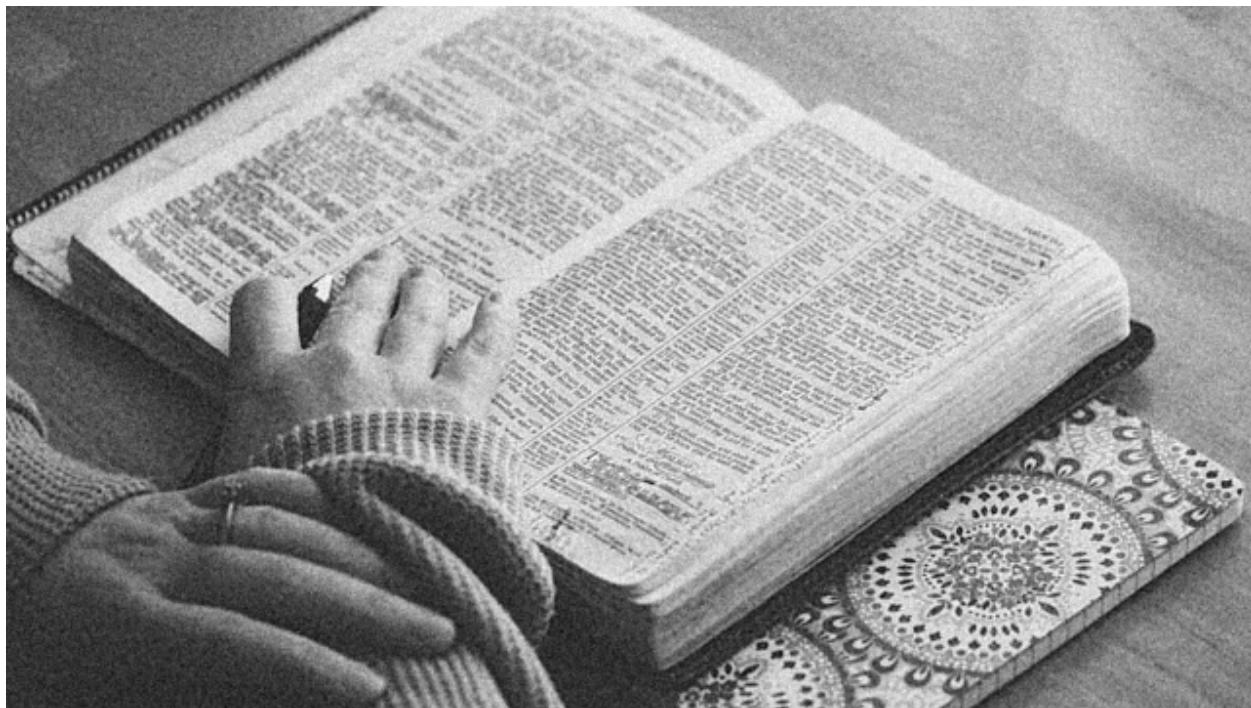
```
>>> myrange
<generator object <genexpr> at 0x10b3732e0>
```

## 5.2.7 Tuplas vs Listas

Aunque puedan parecer estructuras de datos muy similares, sabemos que las tuplas carecen de ciertas operaciones, especialmente las que tienen que ver con la modificación de sus valores, ya que no son inmutables. Si las listas son más flexibles y potentes, ¿por qué íbamos a necesitar tuplas? Veamos 4 potenciales ventajas del uso de tuplas frente a las listas:

1. Las tuplas ocupan **menos espacio** en memoria.
2. En las tuplas existe **protección** frente a cambios indeseados.
3. Las tuplas se pueden usar como **claves de diccionarios** (son «*hashables*»).
4. Las **namedtuples** son una alternativa sencilla a los objetos.

## 5.3 Diccionarios



Podemos trasladar el concepto de *diccionario* de la vida real al de *diccionario* en Python. Al fin y al cabo un diccionario es un objeto que contiene palabras, y cada palabra tiene asociado un significado. Haciendo el paralelismo, diríamos que en Python un diccionario es también un objeto indexado por **claves** (las palabras) que tienen asociados unos **valores** (los significados).<sup>1</sup>

Los diccionarios en Python tienen las siguientes *características*:

- Mantienen el **orden** en el que se insertan las claves.<sup>2</sup>
- Son **mutables**, con lo que admiten añadir, borrar y modificar sus elementos.
- Las **claves** deben ser **únicas**. A menudo se utilizan las *cadenas de texto* como claves, pero en realidad podría ser cualquier tipo de datos inmutable: enteros, flotantes, tuplas (entre otros).
- Tienen un **acceso muy rápido** a sus elementos, debido a la forma en la que están implementados internamente.<sup>3</sup>

---

**Nota:** En otros lenguajes de programación, a los diccionarios se les conoce como *arrays*

<sup>1</sup> Foto original de portada por Aaron Burden en Unsplash.

<sup>2</sup> Aunque históricamente Python no establecía que las claves de los diccionarios tuvieran que mantener su orden de inserción, a partir de Python 3.7 este comportamiento cambió y se garantizó el orden de inserción de las claves como parte oficial de la especificación del lenguaje.

<sup>3</sup> Véase este análisis de complejidad y rendimiento de distintas estructuras de datos en CPython.

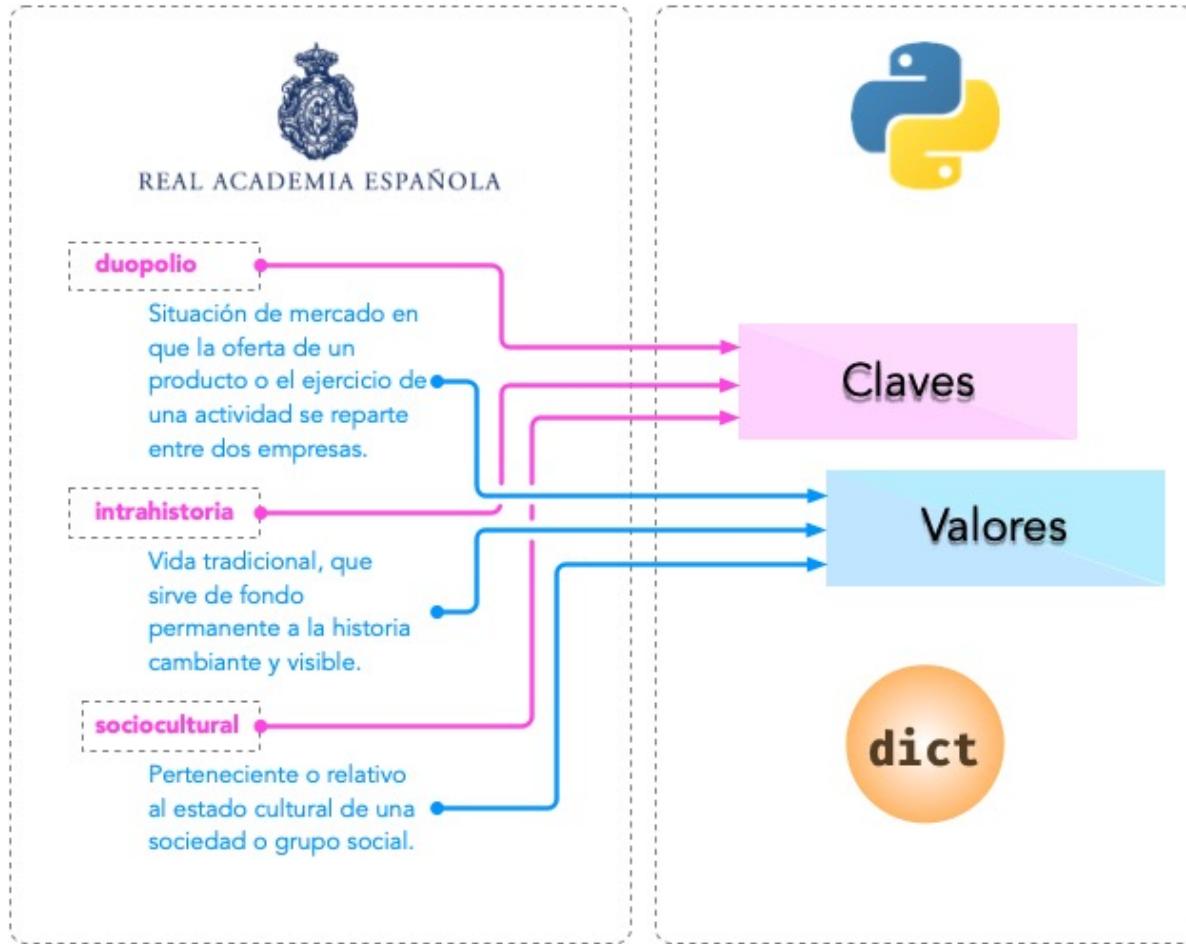


Figura 6: Analogía de un diccionario en Python

asociativos, «hashes» o «hashmaps».

---

### 5.3.1 Creando diccionarios

Para crear un diccionario usamos llaves {} rodeando asignaciones clave: valor que están separadas por comas. Veamos algunos ejemplos de diccionarios:

```
>>> empty_dict = {}

>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> population_can = {
...     2015: 2_135_209,
...     2016: 2_154_924,
...     2017: 2_177_048,
...     2018: 2_206_901,
...     2019: 2_220_270
... }
```

En el código anterior podemos observar la creación de un diccionario vacío, otro donde sus claves y sus valores son cadenas de texto y otro donde las claves y los valores son valores enteros.

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/Sfav2Yw>

### 5.3.2 Conversión

Para convertir otros tipos de datos en un diccionario podemos usar la función dict():

```
>>> # Diccionario a partir de una lista de cadenas de texto
>>> dict(['a1', 'b2'])
{'a': '1', 'b': '2'}

>>> # Diccionario a partir de una tupla de cadenas de texto
>>> dict(('a1', 'b2'))
{'a': '1', 'b': '2'}

>>> # Diccionario a partir de una lista de listas
```

(continué en la próxima página)

(provine de la página anterior)

```
>>> dict([['a', 1], ['b', 2]])
{'a': 1, 'b': 2}
```

---

**Nota:** Si nos fijamos bien, cualquier iterable que tenga una estructura interna de 2 elementos es susceptible de convertirse en un diccionario a través de la función `dict()`.

---

## Diccionario vacío

Existe una manera particular de usar `dict()` y es no pasarle ningún argumento. En este caso estaremos queriendo convertir el «vacío» en un diccionario, con lo que obtendremos un *diccionario vacío*:

```
>>> dict()
{}
```

---

**Truco:** Para crear un diccionario vacío, se suele recomendar el uso de `{}` frente a `dict()`, no sólo por ser más *pitónico* sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

---

## Creación con `dict()`

También es posible utilizar la función `dict()` para crear diccionarios y no tener que utilizar llaves y comillas:

Supongamos que queremos transformar la siguiente tabla en un diccionario:

Atributo	Valor
<code>name</code>	Guido
<code>surname</code>	Van Rossum
<code>job</code>	Python creator

Utilizando la construcción mediante `dict` podemos pasar clave y valor como **argumentos** de la función:

```
>>> person = dict(
...     name='Guido',
...     surname='Van Rossum',
...     job='Python creator'
```

(continué en la próxima página)

(provine de la página anterior)

```
... )  
  
=> person  
{'name': 'Guido', 'surname': 'Van Rossum', 'job': 'Python creator'}
```

El inconveniente que tiene esta aproximación es que las **claves deben ser identificadores válidos** en Python. Por ejemplo, no se permiten espacios:

```
=gt; person = dict(  
...     name='Guido van Rossum',  
...     date of birth='31/01/1956'  
File "<stdin>", line 3  
    date of birth='31/01/1956'  
          ^  
SyntaxError: invalid syntax
```

### 5.3.3 Operaciones con diccionarios

#### Obtener un elemento

Para obtener un elemento de un diccionario basta con escribir la **clave** entre corchetes. Veamos un ejemplo:

```
=gt; rae = {  
...     'bifronte': 'De dos frentes o dos caras',  
...     'anarcoide': 'Que tiende al desorden',  
...     'montuvio': 'Campesino de la costa'  
... }  
  
=> rae['anarcoide']  
'Que tiende al desorden'
```

Si intentamos acceder a una clave que no existe, obtendremos un error:

```
=gt; rae['acceso']  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'acceso'
```

## Usando get()

Existe una función muy útil para «superar» los posibles errores de acceso por claves inexistentes. Se trata de `get()` y su comportamiento es el siguiente:

1. Si la clave que buscamos existe, nos devuelve su valor.
2. Si la clave que buscamos no existe, nos devuelve `None`<sup>4</sup> salvo que le indiquemos otro valor por defecto, pero en ninguno de los dos casos obtendremos un error.

```

1  >>> rae
2  {'bifronte': 'De dos frentes o dos caras',
3  'anarcoide': 'Que tiende al desorden',
4  'montuvio': 'Campesino de la costa'}
5
6  >>> rae.get('bifronte')
7  'De dos frentes o dos caras'
8
9  >>> rae.get('programación')
10
11 >>> rae.get('programación', 'No disponible')
12  'No disponible'
```

**Línea 6:** Equivalente a `rae['bifronte']`.

**Línea 9:** La clave buscada no existe y obtenemos `None`.<sup>5</sup>

**Línea 11:** La clave buscada no existe y nos devuelve el valor que hemos aportado por defecto.

## Añadir o modificar un elemento

Para añadir un elemento a un diccionario sólo es necesario hacer referencia a la *clave* y asignarle un *valor*:

- Si la clave **ya existía** en el diccionario, **se reemplaza** el valor existente por el nuevo.
- Si la clave **es nueva**, **se añade** al diccionario con su valor. *No vamos a obtener un error a diferencia de las listas.*

Partimos del siguiente diccionario para exemplificar estas acciones:

```

>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
```

(continué en la próxima página)

<sup>4</sup> `None` es la palabra reservada en Python para la «nada». Más información en [esta web](#).

<sup>5</sup> Realmente no estamos viendo nada en la consola de Python porque la representación en cadena de texto es vacía.

(provine de la página anterior)

```
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }
```

Vamos a **añadir** la palabra *enjuiciar* a nuestro diccionario de la Real Academia de La Lengua:

```
>>> rae['enjuiciar'] = 'Someter una cuestión a examen, discusión y juicio'

>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'}
```

Supongamos ahora que queremos **modificar** el significado de la palabra *enjuiciar* por otra acepción:

```
>>> rae['enjuiciar'] = 'Instruir, juzgar o sentenciar una causa'

>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}
```

---

### Ejercicio

Construya un diccionario partiendo de una cadena de texto con el siguiente formato:

<city>:<population>;<city>:<population>;<city>:<population>;....

- Claves: **ciudades**.
- Valores: **habitantes** (*como enteros*).

### Ejemplo

- Entrada: Tokyo:38\_140\_000;Delhi:26\_454\_000;Shanghai:24\_484\_000;  
Mumbai:21\_357\_000;São Paulo:21\_297\_000
- Salida: {'Tokyo': 38140000, 'Delhi': 26454000, 'Shanghai': 24484000,  
'Mumbai': 21357000, 'São Paulo': 21297000}

## Creando desde vacío

Una forma muy habitual de trabajar con diccionarios es utilizar el **patrón creación** partiendo de uno vacío e ir añadiendo elementos poco a poco.

Supongamos un ejemplo en el que queremos construir un diccionario donde las claves son las letras vocales y los valores son sus posiciones:

```
>>> VOWELS = 'aeiou'

>>> enum_vowels = {}

>>> for i, vowel in enumerate(VOWELS):
...     enum_vowels[vowel] = i + 1
...

>>> enum_vowels
{'a': 1, 'e': 2, 'i': 3, 'o': 4, 'u': 5}
```

---

**Nota:** Hemos utilizando la función `enumerate()` que ya vimos para las listas en el apartado: *Iterar usando enumeración*.

---

## Pertenencia de una clave

La forma **pitónica** de comprobar la existencia de una clave dentro de un diccionario, es utilizar el operador `in`:

```
>>> 'bifronte' in rae
True

>>> 'almohada' in rae
False

>>> 'montuvio' not in rae
False
```

---

**Nota:** El operador `in` siempre devuelve un valor booleano, es decir, verdadero o falso.

---

---

## Ejercicio

Usando un diccionario, cuente el número de veces que se repite cada letra en una cadena de texto dada.

## Ejemplo

- Entrada: 'boom'
  - Salida: { 'b': 1, 'o': 2, 'm': 1}
- 

## Obtener todos los elementos

Python ofrece mecanismos para obtener todos los elementos de un diccionario. Partimos del siguiente diccionario:

```
>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}
```

**Obtener todas las claves de un diccionario:** Mediante la función `keys()`:

```
>>> rae.keys()
dict_keys(['bifronte', 'anarcoide', 'montuvio', 'enjuiciar'])
```

**Obtener todos los valores de un diccionario:** Mediante la función `values()`:

```
>>> rae.values()
dict_values([
    'De dos frentes o dos caras',
    'Que tiende al desorden',
    'Campesino de la costa',
    'Instruir, juzgar o sentenciar una causa'
])
```

**Obtener todos los pares «clave-valor» de un diccionario:** Mediante la función `items()`:

```
>>> rae.items()
dict_items([
    ('bifronte', 'De dos frentes o dos caras'),
    ('anarcoide', 'Que tiende al desorden'),
    ('montuvio', 'Campesino de la costa'),
    ('enjuiciar', 'Instruir, juzgar o sentenciar una causa')
])
```

---

**Nota:** Para este último caso cabe destacar que los «`items`» se devuelven como una lista de *tuplas*, donde cada tupla tiene dos elementos: el primero representa la clave y el segundo

representa el valor.

## Longitud de un diccionario

Podemos conocer el número de elementos («clave-valor») que tiene un diccionario con la función `len()`:

```
>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}

>>> len(rae)
4
```

## Iterar sobre un diccionario

En base a *los elementos que podemos obtener*, Python nos proporciona tres maneras de iterar sobre un diccionario.

**Iterar sobre claves:**

```
>>> for word in rae.keys():
...     print(word)
...
bifronte
anarcoide
montuvio
enjuiciar
```

**Iterar sobre valores:**

```
>>> for meaning in rae.values():
...     print(meaning)
...
De dos frentes o dos caras
Que tiende al desorden
Campesino de la costa
Instruir, juzgar o sentenciar una causa
```

**Iterar sobre «clave-valor»:**

```
>>> for word, meaning in rae.items():
...     print(f'{word}: {meaning}')
...
bifronte: De dos frentes o dos caras
anarcoide: Que tiende al desorden
montuvio: Campesino de la costa
enjuiciar: Instruir, juzgar o sentenciar una causa
```

---

**Nota:** En este último caso, recuerde el uso de los «*f-strings*» para formatear cadenas de texto.

---

### Ejercicio

Dado el diccionario de ciudades y poblaciones ya visto, y suponiendo que estas ciudades son las únicas que existen en el planeta, calcule el porcentaje de población relativo de cada una de ellas con respecto al total.

### Ejemplo

- Entrada: Tokyo:38\_140\_000;Delhi:26\_454\_000;Shanghai:24\_484\_000; Mumbai:21\_357\_000;São Paulo:21\_297\_000
  - Salida: {'Tokyo': 28.952722193544467, 'Delhi': 20.081680988673973, 'Shanghai': 18.58622050830474, 'Mumbai': 16.212461664591746, 'São Paulo': 16.16691464488507}
- 

### Combinar diccionarios

Dados dos (o más) diccionarios, es posible «mezclarlos» para obtener una combinación de los mismos. Esta combinación se basa en dos premisas:

1. Si la clave no existe, se añade con su valor.
2. Si la clave ya existe, se añade con el valor del «último» diccionario en la mezcla.<sup>6</sup>

Python ofrece dos mecanismos para realizar esta combinación. Vamos a partir de los siguientes diccionarios para ejemplificar su uso:

```
>>> rae1 = {
...     'bifronte': 'De dos frentes o dos caras',
...     'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'
... }
```

(continué en la próxima página)

---

<sup>6</sup> En este caso «último» hace referencia al diccionario que se encuentra más a la derecha en la expresión.

(provine de la página anterior)

```
>>> rae2 = {
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa',
...     'enjuiciar': 'Instruir, juzgar o sentenciar una causa'
... }
```

**Sin modificar los diccionarios originales:** Mediante el operador \*\*:

```
>>> {**rae1, **rae2}
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

A partir de **Python 3.9** podemos utilizar el operador | para combinar dos diccionarios:

```
>>> rae1 | rae2
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

**Modificando los diccionarios originales:** Mediante la función update():

```
>>> rae1.update(rae2)

>>> rae1
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

---

**Nota:** Tener en cuenta que el orden en el que especificamos los diccionarios a la hora de su combinación (mezcla) es relevante en el resultado final. En este caso *el orden de los factores sí altera el producto.*

---

## Borrar elementos

Python nos ofrece, al menos, tres formas para borrar elementos en un diccionario:

**Por su clave:** Mediante la sentencia `del`:

```
>>> rae = {  
...     'bifronte': 'De dos frentes o dos caras',  
...     'anarcoide': 'Que tiende al desorden',  
...     'montuvio': 'Campesino de la costa'  
... }  
  
>>> del(rae['bifronte'])  
  
>>> rae  
{'anarcoide': 'Que tiende al desorden', 'montuvio': 'Campesino de la costa'}
```

**Por su clave (con extracción):** Mediante la función `pop()` podemos extraer un elemento del diccionario por su clave. Vendría a ser una combinación de `get() + del`:

```
>>> rae = {  
...     'bifronte': 'De dos frentes o dos caras',  
...     'anarcoide': 'Que tiende al desorden',  
...     'montuvio': 'Campesino de la costa'  
... }  
  
>>> rae.pop('anarcoide')  
'Que tiende al desorden'  
  
>>> rae  
{'bifronte': 'De dos frentes o dos caras', 'montuvio': 'Campesino de la costa'}  
  
>>> rae.pop('bucle')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'bucle'
```

**Advertencia:** Si la clave que pretendemos extraer con `pop()` no existe, obtendremos un error.

## Borrado completo del diccionario:

1. Utilizando la función `clear()`:

```
>>> rae = {  
...     'bifronte': 'De dos frentes o dos caras',  
... }
```

(continué en la próxima página)

(proviene de la página anterior)

```

...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> rae.clear()

>>> rae
{}

```

2. «Reinicializando» el diccionario a vacío con {}:

```

>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> rae = {}

>>> rae
{}

```

**Nota:** La diferencia entre ambos métodos tiene que ver con cuestiones internas de gestión de memoria y de rendimiento.

### 5.3.4 Cuidado con las copias

#### Nivel intermedio

Al igual que ocurría con *las listas*, si hacemos un cambio en un diccionario, se verá reflejado en todas las variables que hagan referencia al mismo. Esto se deriva de su propiedad de ser *mutable*. Veamos un ejemplo concreto:

```

>>> original_rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> copy_rae = original_rae

>>> original_rae['bifronte'] = 'bla bla bla'

```

(continué en la próxima página)

(provien de la página anterior)

```
>>> original_rae
{'bifronte': 'bla bla bla',
'anarcoide': 'Que tiende al desorden',
'montuvio': 'Campesino de la costa'}
```

```
>>> copy_rae
{'bifronte': 'bla bla bla',
'anarcoide': 'Que tiende al desorden',
'montuvio': 'Campesino de la costa'}
```

Una  **posible solución** a este problema es hacer una «copia dura». Para ello Python proporciona la función `copy()`:

```
>>> original_rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> copy_rae = original_rae.copy()

>>> original_rae['bifronte'] = 'bla bla bla'

>>> original_rae
{'bifronte': 'bla bla bla',
'anarcoide': 'Que tiende al desorden',
'montuvio': 'Campesino de la costa'}
```

```
>>> copy_rae
{'bifronte': 'De dos frentes o dos caras',
'anarcoide': 'Que tiende al desorden',
'montuvio': 'Campesino de la costa'}
```

---

**Truco:** En el caso de que estemos trabajando con diccionarios que contienen elementos mutables, debemos hacer uso de la función `deepcopy()` dentro del módulo `copy` de la librería estándar.

---

### 5.3.5 Diccionarios por comprensión

#### Nivel intermedio

De forma análoga a cómo se escriben las *listas por comprensión*, podemos aplicar este método a los diccionarios usando llaves { }.

Veamos un ejemplo en el que creamos un **diccionario por comprensión** donde las claves son palabras y los valores son sus longitudes:

```
>>> words = ('sun', 'space', 'rocket', 'earth')

>>> words_length = {word: len(word) for word in words}

>>> words_length
{'sun': 3, 'space': 5, 'rocket': 6, 'earth': 5}
```

También podemos aplicar **condiciones** a estas comprensiones. Continuando con el ejemplo anterior, podemos incorporar la restricción de sólo incluir palabras que no empiecen por vocal:

```
>>> words = ('sun', 'space', 'rocket', 'earth')

>>> words_length = {w: len(w) for w in words if w[0] not in 'aeiou'}

>>> words_length
{'sun': 3, 'space': 5, 'rocket': 6}
```

---

**Nota:** Se puede consultar el [PEP-274](#) para ver más ejemplos sobre diccionarios por comprensión.

---

### 5.3.6 Objetos «hashables»

#### Nivel avanzado

La única restricción que deben cumplir las **claves** de un diccionario es ser «**hashables**»<sup>7</sup>. Un objeto es «hashable» si se le puede asignar un valor «hash» que no cambia en ejecución durante toda su vida.

Para encontrar el «hash» de un objeto, Python usa la función `hash()`, que devuelve un número entero y es utilizado para indexar la *tabla «hash»* que se mantiene internamente:

---

<sup>7</sup> Se recomienda [esta ponencia](#) de Víctor Terrón sobre objetos «hashables».

```
>>> hash(999)
999

>>> hash(3.14)
322818021289917443

>>> hash('hello')
-8103770210014465245

>>> hash(('a', 'b', 'c'))
-2157188727417140402
```

Para que un objeto sea «hashable», debe ser **inmutable**:

```
>>> hash(['a', 'b', 'c'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

---

**Nota:** De lo anterior se deduce que las claves de los diccionarios, al tener que ser «hashables», sólo pueden ser objetos inmutables.

---

La función «built-in» `hash()` realmente hace una llamada al método mágico `__hash__()` del objeto en cuestión:

```
>>> hash('spiderman')
-8105710090476541603

>>> 'spiderman'.__hash__()
-8105710090476541603
```

---

## EJERCICIOS DE REPASO

1. Escriba un programa en Python que acepte una lista de palabras y las agrupe por su letra inicial usando un diccionario ([solución](#)).

Entrada: [ “mesa”, “móvil”, “barco”, “coche”, “avión”, “bandeja”, “casa”, “monitor”, “carretera”, “arco”]

Salida: {“m”: [“mesa”, “móvil”, “monitor”], “b”: [“barco”, “bandeja”], “c”: [“coche”, “casa”, “carretera”], “a”: [“avión”, “arco”]}

2. Escriba un programa en Python que acepte un diccionario y determine si todos los valores son iguales o no ([solución](#)).

Entrada: {"Juan": 5, "Antonio": 5, "Inma": 5, "Ana": 5, "Esteban": 5}

Salida: Same values

3. Escriba un programa en Python que acepte una lista de listas con varios elementos y obtenga un diccionario donde las claves serán los primeros elementos de las sublistas y los valores serán los restantes – como listas – (**solución**).

Entrada: [["Episode IV - A New Hope", "May 25", 1977], ["Episode V - The Empire Strikes Back", "May 21", 1980], ["Episode VI - Return of the Jedi", "May 25", 1983]]

Salida: {"Episode IV - A New Hope": ["May 25", 1977], "Episode V - The Empire Strikes Back": ["May 21", 1980], "Episode VI - Return of the Jedi": ["May 25", 1983]}

4. Escriba un programa en Python que acepte un diccionario cuyos valores son listas y borre el contenido de dichas listas (**solución**).

Entrada: {"C1": [10, 20, 30], "C2": [20, 30, 40], "C3": [12, 34]}

Salida: {"C1": [], "C2": [], "C3": []}

5. Escriba un programa en Python que acepte un diccionario y elimine los espacios de sus claves respetando los valores correspondientes (**solución**).

Entrada: {"S 001": ["Math", "Science"], "S 002": ["Math", "English"]}

Salida: {"S001": ["Math", "Science"], "S002": ["Math", "English"]}

## AMPLIAR CONOCIMIENTOS

- Using the Python defaultdict Type for Handling Missing Keys
- Python Dictionary Iteration: Advanced Tips & Tricks
- Python KeyError Exceptions and How to Handle Them
- Dictionaries in Python
- How to Iterate Through a Dictionary in Python
- Shallow vs Deep Copying of Python Objects

## 5.4 Conjuntos



Un **conjunto** en Python representa una serie de **valores únicos y sin orden establecido**, con la única restricción de que sus elementos deben ser «*hashables*». Mantiene muchas similitudes con el concepto matemático de conjunto<sup>1</sup>

### 5.4.1 Creando conjuntos

Para crear un conjunto basta con separar sus valores por *comas* y rodearlos de llaves {}:

```
>>> lottery = {21, 10, 46, 29, 31, 94}

>>> lottery
{10, 21, 29, 31, 46, 94}
```

La excepción la tenemos a la hora de crear un **conjunto vacío**, ya que, siguiendo la lógica de apartados anteriores, deberíamos hacerlo a través de llaves:

```
>>> wrong_empty_set = {}

>>> type(wrong_empty_set)
dict
```

---

<sup>1</sup> Foto original de portada por Duy Pham en Unsplash.

**Advertencia:** Si hacemos esto, lo que obtenemos es un *diccionario vacío*.

La única opción que tenemos es utilizar la función `set()`:

```
>>> empty_set = set()

>>> empty_set
set()

>>> type(empty_set)
set
```

## 5.4.2 Conversión

Para convertir otros tipos de datos en un conjunto podemos usar la función `set()` sobre cualquier iterable:

```
>>> set('aplatanada')
{'a', 'd', 'l', 'n', 'p', 't'}

>>> set([1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5])
{1, 2, 3, 4, 5}

>>> set(('ADENINA', 'TIMINA', 'TIMINA', 'GUANINA', 'ADENINA', 'CITOSINA'))
{'ADENINA', 'CITOSINA', 'GUANINA', 'TIMINA'}

>>> set({'manzana': 'rojo', 'plátano': 'amarillo', 'kiwi': 'verde'})
{'kiwi', 'manzana', 'plátano'}
```

---

**Importante:** Como se ha visto en los ejemplos anteriores, `set()` se suele utilizar en muchas ocasiones como una forma de **extraer los valores únicos** de otros tipos de datos. En el caso de los diccionarios se extraen las claves, que, por definición, son únicas.

---



---

**Nota:** El hecho de que en los ejemplos anteriores los elementos de los conjuntos estén ordenados es únicamente un «detalle de implementación» en el que no se puede confiar.

---

### 5.4.3 Operaciones con conjuntos

#### Obtener un elemento

En un conjunto no existe un orden establecido para sus elementos, por lo tanto **no podemos acceder a un elemento en concreto**.

De este hecho se deriva igualmente que **no podemos modificar un elemento existente**, ya que ni siquiera tenemos acceso al mismo. Python sí nos permite añadir o borrar elementos de un conjunto.

#### Añadir un elemento

Para añadir un elemento a un conjunto debemos utilizar la función `add()`. Como ya hemos indicado, al no importar el orden dentro del conjunto, la inserción no establece a priori la posición donde se realizará.

A modo de ejemplo, vamos a partir de un conjunto que representa a los cuatro integrantes originales de *The Beatles*. Luego añadiremos a un nuevo componente:

```
>>> # John Lennon, Paul McCartney, George Harrison y Ringo Starr
>>> beatles = set(['Lennon', 'McCartney', 'Harrison', 'Starr'])

>>> beatles.add('Best') # Pete Best

>>> beatles
{'Best', 'Harrison', 'Lennon', 'McCartney', 'Starr'}
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://tinyurl.com/9folv2v>

---

#### Ejercicio

Dada una tupla de duplas (2 valores), cree dos conjuntos:

- Uno de ellos con los primeros valores de cada dupla.
- El otro con los segundos valores de cada dupla.

#### Ejemplo

- Entrada: `((4, 3), (8, 2), (7, 5), (8, 2), (9, 1))`
- Salida:

```
{8, 9, 4, 7}
{1, 2, 3, 5}
```

## Borrar elementos

Para borrar un elemento de un conjunto podemos utilizar la función `remove()`. Siguiendo con el ejemplo anterior, vamos a borrar al último «beatle» añadido:

```
>>> beatles
{'Best', 'Harrison', 'Lennon', 'McCartney', 'Starr'}
```

```
>>> beatles.remove('Best')
```

```
>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}
```

## Longitud de un conjunto

Podemos conocer el número de elementos que tiene un conjunto con la función `len()`:

```
>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}
```

```
>>> len(beatles)
```

```
4
```

## Iterar sobre un conjunto

Tal y como hemos visto para otros tipos de datos *iterables*, la forma de recorrer los elementos de un conjunto es utilizar la sentencia `for`:

```
>>> for beatle in beatles:
...     print(beatle)
...
Harrison
McCartney
Starr
Lennon
```

---

**Consejo:** Como en el ejemplo anterior, es muy común utilizar una *variable en singular* para recorrer un iterable (en plural). No es una regla fija ni sirve para todos los casos, pero sí suele ser una *buenas prácticas*.

---

## Pertenencia de elemento

Al igual que con otros tipos de datos, Python nos ofrece el operador `in` para determinar si un elemento pertenece a un conjunto:

```
>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}

>>> 'Lennon' in beatles
True

>>> 'Fari' in beatles
False
```

## 5.4.4 Teoría de conjuntos

Vamos a partir de dos conjuntos  $A = \{1, 2\}$  y  $B = \{2, 3\}$  para exemplificar las distintas operaciones que se pueden hacer entre ellos basadas en los Diagramas de Venn y la Teoría de Conjuntos:

```
>>> A = {1, 2}

>>> B = {2, 3}
```

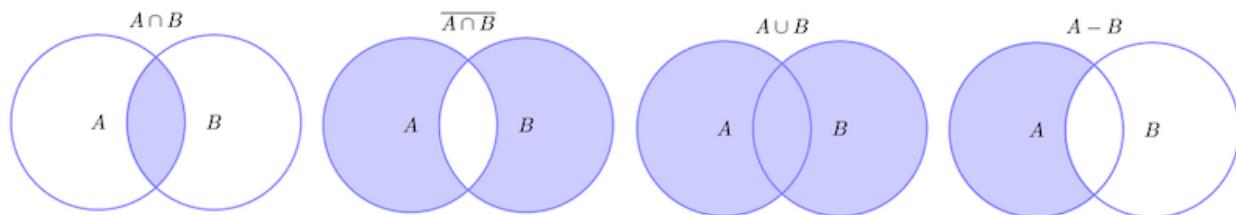


Figura 7: Diagramas de Venn

## Intersección

$A \cap B$  – Elementos que están a la vez en  $A$  y en  $B$ :

```
>>> A & B
{2}

>>> A.intersection(B)
{2}
```

## Unión

$A \cup B$  – Elementos que están tanto en  $A$  como en  $B$ :

```
>>> A | B
{1, 2, 3}

>>> A.union(B)
{1, 2, 3}
```

## Diferencia

$A - B$  – Elementos que están en  $A$  y no están en  $B$ :

```
>>> A - B
{1}

>>> A.difference(B)
{1}
```

## Diferencia simétrica

$\overline{A \cap B}$  – Elementos que están en  $A$  o en  $B$  pero no en ambos conjuntos:

```
>>> A ^ B
{1, 3}

>>> A.symmetric_difference(B)
{1, 3}
```

## 5.4.5 Conjuntos por comprensión

Los conjuntos, al igual que las *listas* y los *diccionarios*, también se pueden crear por comprensión.

Veamos un ejemplo en el que construimos un conjunto por comprensión con los aquellos números enteros múltiplos de 3 en el rango [0, 20]):

```
>>> m3 = {number for number in range(0, 20) if number % 3 == 0}

>>> m3
{0, 3, 6, 9, 12, 15, 18}
```

### Ejercicio

Dadas dos cadenas de texto, obtenga una nueva cadena de texto con las **letras consonantes** que se **repiten en ambas frases**. Ignore los espacios en blanco y muestre la cadena de salida con sus *letras ordenadas*.

*Use conjuntos por comprensión para el ejercicio.*

### Ejemplo

- Entrada: Flat is better than nested y Readability counts
  - Salida: bdlnst
- 

## 5.4.6 Conjuntos inmutables

Python ofrece la posibilidad de crear **conjuntos inmutables** haciendo uso de la función `frozenset()` que recibe cualquier iterable como argumento.

Supongamos que recibimos una serie de calificaciones de exámenes y queremos crear un conjunto inmutable con los posibles niveles (categorías) de calificaciones:

```
>>> marks = [1, 3, 2, 3, 1, 4, 2, 4, 5, 2, 5, 5, 3, 1, 4]

>>> marks_levels = frozenset(marks)

>>> marks_levels
frozenset({1, 2, 3, 4, 5})
```

Veamos qué ocurre si intentamos modificar este conjunto:

```
>>> marks_levels.add(50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

---

**Nota:** Los `frozenset` son a los `sets` lo que las tuplas a las listas: una forma de «congelar» los valores para que no se puedan modificar.

---

---

## AMPLIAR CONOCIMIENTOS

- Sets in Python

## 5.5 Ficheros



Aunque los ficheros encajarían más en un apartado de «*entrada/salida*» ya que representan un **medio de almacenamiento persistente**, también podrían ser vistos como *estructuras de datos*, puesto que nos permiten guardar la información y asignarles un cierto formato.<sup>1</sup>

Un **fichero** es un *conjunto de bytes* almacenados en algún *dispositivo*. El *sistema de ficheros* es la estructura lógica que alberga los ficheros y está jerarquizado a través de *directorios* (o carpetas). Cada fichero se identifica únicamente a través de una *ruta* que nos permite acceder a él.

---

<sup>1</sup> Foto original de portada por Maksym Kaharlytskyi en Unsplash.

### 5.5.1 Lectura de un fichero

Python ofrece la función `open()` para «abrir» un fichero. Esta apertura se puede realizar en 3 modos distintos:

- **Lectura** del contenido de un fichero existente.
- **Escritura** del contenido en un fichero nuevo.
- **Añadido** al contenido de un fichero existente.

Veamos un ejemplo para leer el contenido de un fichero en el que se encuentran las temperaturas máximas y mínimas de cada día de la última semana. El fichero está en la subcarpeta (*ruta relativa*) `files/temps.dat` y tiene el siguiente contenido:

```
29 23  
31 23  
34 26  
33 23  
29 22  
28 22  
28 22
```

Lo primero será abrir el fichero:

```
>>> f = open('files/temps.dat')
```

La función `open()` recibe como primer argumento la **ruta al fichero** que queremos manejar (como un «string») y devuelve el manejador del fichero, que en este caso lo estamos asignando a una variable llamada `f` pero le podríamos haber puesto cualquier otro nombre.

---

**Nota:** Es importante dominar los conceptos de **ruta relativa** y **ruta absoluta** para el trabajo con ficheros. Véase [este artículo de DeNovatoANovato](#).

---

Hay que tener en cuenta que la ruta al fichero que abrimos (*en modo lectura*) **debe existir**, ya que de lo contrario obtendremos un error:

```
>>> f = open('foo.txt')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
FileNotFoundError: [Errno 2] No such file or directory: 'foo.txt'
```

Una vez abierto el fichero ya podemos proceder a leer su contenido. Para ello Python nos ofrece la posibilidad de leer todo el fichero de una vez o bien leerlo línea a línea.

## Lectura completa de un fichero

Siguiendo con nuestro ejemplo de temperaturas, veamos cómo leer todo el contenido del fichero de una sola vez. Para esta operación, Python nos provee, al menos, de dos funciones:

**read()** Devuelve todo el contenido del fichero como una cadena de texto (**str**):

```
>>> f = open('files/temps.dat')
>>> f.read()
'29 23\n31 23\n34 26\n33 23\n29 22\n28 22\n28 22\n'
```

**readlines()** Devuelve todo el contenido del fichero como una lista (**list**) donde cada elemento es una línea:

```
>>> f = open('files/temps.dat')
>>> f.readlines()
['29 23\n', '31 23\n', '34 26\n', '33 23\n', '29 22\n', '28 22\n', '28 22\n']
```

---

**Importante:** Nótese que, en ambos casos, los saltos de línea `\n` siguen apareciendo en los datos leídos, por lo que habría que «limpiar» estos caracteres. Para ello se recomienda utilizar *las funciones ya vistas de cadenas de texto*.

---

## Lectura línea a línea

Hay situaciones en las que interesa leer el contenido del fichero línea a línea. Imaginemos un fichero de tamaño considerable (varios GB). Si intentamos leer completamente este fichero de sola una vez podríamos ocupar demasiada RAM y reducir el rendimiento de nuestra máquina.

Es por ello que Python nos ofrece varias aproximaciones a la lectura de ficheros línea a línea. La más usada es **iterar** sobre el propio *manejador* del fichero:

```
>>> f = open('files/temps.dat')

>>> for line in f:    # that easy!
...     print(line)
...
29 23

31 23

34 26
```

(continuó en la próxima página)

(provine de la página anterior)

```
33 23
```

```
29 22
```

```
28 22
```

```
28 22
```

---

**Truco:** Igual que pasaba anteriormente, la lectura línea por línea también incluye el **salto de línea** `\n` lo que provoca un «doble espacio» entre cada una de las salidas. Bastaría con aplicar `line.split()` para eliminarlo.

---

### 5.5.2 Escritura en un fichero

Para escribir texto en un fichero hay que abrir dicho fichero en **modo escritura**. Para ello utilizamos un *argumento adicional* en la función `open()` que indica esta operación:

```
>>> f = open('files/canary-iata.dat', 'w')
```

---

**Nota:** Si bien el fichero en sí mismo se crea al abrirlo en modo escritura, la **ruta** hasta ese fichero no. Eso quiere decir que debemos asegurarnos que **las carpetas hasta llegar a dicho fichero existen**. En otro caso obtenemos un error de tipo `FileNotFoundException`.

---

Ahora ya podemos hacer uso de la función `write()` para enviar contenido al fichero abierto.

Supongamos que queremos volcar el contenido de una lista en dicho fichero. En este caso partimos de los *códigos IATA* de aeropuertos de las Islas Canarias<sup>2</sup>.

```
1 >>> canary_iata = ("GCFV", "GCHI", "GCLA", "GCLP", "GCGM", "GCRR", "GCTS", "GCXO")
2
3 >>> for code in canary_iata:
4     f.write(code + '\n')
5 ...
6
7 >>> f.close()
```

Nótese:

---

<sup>2</sup> Fuente: Smart Drone

**Línea 4** Escritura de cada código en el fichero. La función `write()` no incluye el salto de línea por defecto, así que lo añadimos de *manera explícita*.

**Línea 7** Cierre del fichero con la función `close()`. Especialmente en el caso de la escritura de ficheros, se recomienda encarecidamente cerrar los ficheros para evitar pérdida de datos.

**Advertencia:** Siempre que se abre un fichero en **modo escritura** utilizando el argumento '`w`', el fichero se inicializa, borrando cualquier contenido que pudiera tener.

### 5.5.3 Añadido a un fichero

La única diferencia entre añadir información a un fichero y *escribir información en un fichero* es el modo de apertura del fichero. En este caso utilizamos '`a`' por «append»:

```
>>> f = open('more-data.txt', 'a')
```

En este caso el fichero `more-data.txt` se abrirá en *modo añadir* con lo que las llamadas a la función `write()` hará que aparezcan nueva información al final del contenido ya existente en dicho fichero.

### 5.5.4 Usando contextos

Python ofrece *gestores de contexto* como una solución para establecer reglas de entrada y salida a un determinado bloque de código.

En el caso que nos ocupa, usaremos la sentencia `with` y el contexto creado se ocupará de cerrar adecuadamente el fichero que hemos abierto, liberando así sus recursos:

```
1  >>> with open('files/temps.dat') as f:
2      ...     for line in f:
3          ...         min_temp, max_temp = line.strip().split()
4          ...         print(min_temp, max_temp)
5
6 29 23
7 31 23
8 34 26
9 33 23
10 29 22
11 28 22
12 28 22
```

**Línea 1** Apertura del fichero en *modo lectura* utilizando el gestor de contexto definido por la palabra reservada `with`.

**Línea 2** Lectura del fichero línea a línea utilizando la iteración sobre el *manejador del fichero*.

**Línea 3** Limpieza de saltos de línea con `strip()` encadenando la función `split()` para separar las dos temperaturas por el carácter *espacio*. Ver [limpiar una cadena](#) y [dividir una cadena](#).

**Línea 4** Imprimir por pantalla la temperatura mínima y la máxima.

---

**Nota:** Es una buena práctica usar `with` cuando se manejan ficheros. La ventaja es que el fichero se cierra adecuadamente en cualquier circunstancia, incluso si se produce cualquier **tipo de error**.

---

Hay que prestar atención a la hora de escribir valores numéricos en un fichero, ya que el método `write()` por defecto espera ver un «string» como argumento:

```
>>> lottery = [43, 21, 99, 18, 37, 99]

>>> with open('files/lottery.dat', 'w') as f:
...     for number in lottery:
...         f.write(number + '\n')

...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

---

**Importante:** Para evitar este tipo de **errores**, se debe convertir a `str` aquellos valores que queramos usar con la función `write()` para escribir información en un fichero de texto.

---

### Ejercicio

Dado el fichero `temperatures.txt` con 12 filas (*meses*) y 31 columnas (temperaturas de cada día), se pide:

1. Leer el fichero de datos.
2. Calcular la temperatura media de cada mes.
3. Escribir un fichero de salida `avgtemps.txt` con 12 filas (*meses*) y la temperatura media de cada mes.

*Guarda el fichero en la misma carpeta en la que vas a escribir tu código. Así evitarás problemas de rutas relativas/absolutas.*

---

## AMPLIAR CONOCIMIENTOS

- Reading and Writing Files in Python
- Python Context Managers and the «with» Statement



# CAPÍTULO 6

---

## Modularidad

---

La **modularidad** es la característica de un sistema que permite que sea estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan solidariamente para alcanzar un objetivo común, realizando cada una de ellas una tarea necesaria para la consecución de dicho objetivo.

Cada una de esas partes en que se encuentre dividido el sistema recibe el nombre de **módulo**. Idealmente un módulo debe poder cumplir las condiciones de caja negra, es decir, ser independiente del resto de los módulos y comunicarse con ellos (con todos o sólo con una parte) a través de entradas y salidas bien definidas.<sup>1</sup>

En este capítulo veremos las facilidades que nos proporciona Python para trabajar en la línea de modularidad del código.

---

<sup>1</sup> Definición de modularidad en [Wikipedia](#)

## 6.1 Funciones



El concepto de **función** es básico en prácticamente cualquier lenguaje de programación. Se trata de una estructura que nos permite agrupar código. Persigue dos objetivos claros:

1. **No repetir** trozos de código durante nuestro programa.
2. **Reutilizar** el código para distintas situaciones.

Una función viene *definida* por su *nombre*, sus *parámetros* y su *valor de retorno*. Esta parametrización de las funciones las convierte en una poderosa herramienta ajustable a las circunstancias que tengamos. Al *invocarla* estaremos solicitando su ejecución y obtendremos unos resultados.<sup>1</sup>

### 6.1.1 Definir una función

Para definir una función utilizamos la palabra reservada `def` seguida del **nombre<sup>6</sup>** de la función. A continuación aparecerán 0 o más **parámetros** separados por comas (entre paréntesis), finalizando la línea con **dos puntos** : En la siguiente línea empezaría el **cuerpo** de la función que puede contener 1 o más **sentencias**, incluyendo (o no) una **sentencia de retorno** con el resultado mediante `return`.

---

<sup>1</sup> Foto original por Nathan Dumlao en Unsplash.

<sup>6</sup> Las *reglas aplicadas a nombres de variables* también se aplican a nombres de funciones.

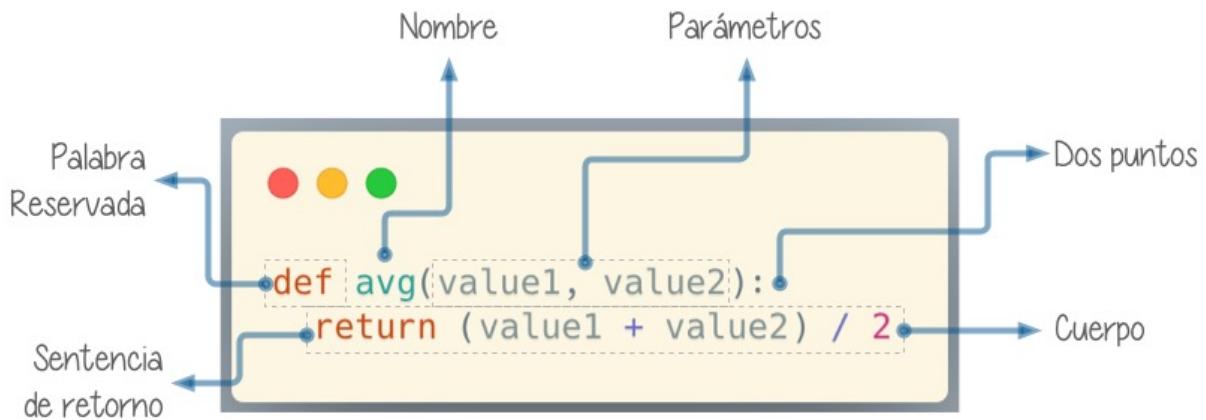


Figura 1: Definición de una función en Python

**Advertencia:** Prestar especial atención a los dos puntos : porque suelen olvidarse en la *definición de la función*.

Hagamos una primera función sencilla que no recibe parámetros:

```
def say_hello():
    print('Hello!')
```

- Nótese la *indentación* (sangrado) del *cuerpo* de la función.
- Los *nombres de las funciones* siguen *las mismas reglas que las variables*.

## Invocar una función

Para invocar (o «llamar») a una función sólo tendremos que escribir su nombre seguido de paréntesis. En el caso de la función sencilla (vista anteriormente) se haría así:

```
>>> def say_hello():
...     print('Hello!')
...
>>> say_hello()
Hello!
```

Como era de esperar, al invocar a esta función obtenemos un mensaje por pantalla, fruto de la ejecución del cuerpo de la función.

## Retornar un valor

Las funciones pueden retornar (o «devolver») un valor. Veamos un ejemplo muy sencillo:

```
>>> def one():
...     return 1
...
>>> one()
1
```

---

**Importante:** No confundir `return` con `print()`. El valor de retorno de una función nos permite usarlo fuera de su contexto. El hecho de añadir `print()` al cuerpo de una función es algo «coyuntural» y no modifica el resultado de la lógica interna.

---

**Nota:** En la sentencia `return` podemos incluir variables y expresiones, no únicamente literales.

---

Pero no sólo podemos invocar a la función directamente, también la podemos integrar en otras expresiones. Por ejemplo en condicionales:

```
>>> if one() == 1:
...     print('It works!')
... else:
...     print('Something is broken')
...
It works!
```

Si una función no incluye un `return` de forma explícita, devolverá `None` de forma implícita:

```
>>> def empty():
...     x = 0
...
>>> print(empty())
None
```

## 6.1.2 Veracidad

### Nivel intermedio

Ya hemos hablado ligeramente sobre la *comprobación de veracidad* en Python.

Vamos a crear una función propia para comprobar la veracidad de distintos objetos del lenguaje, y así hacernos una mejor idea de qué cosas **son evaluadas** a *verdadero* y cuáles a *falso*:

```
>>> def truthiness(obj):
...     if obj:
...         print(f'{obj} is True')
...     else:
...         print(f'{obj} is False')
...
```

### Evaluando a False

Veamos qué «cosas» son evaluadas a *False* en Python:

```
>>> truthiness(False)
False is False

>>> truthiness(None)
None is False

>>> truthiness(0)
0 is False

>>> truthiness(0.0)
0.0 is False

>>> truthiness('')
is False

>>> truthiness([])
[] is False

>>> truthiness(())
() is False

>>> truthiness({})
{} is False
```

(continué en la próxima página)

(provine de la página anterior)

```
>>> truthiness(set())
set() is False
```

---

**Importante:** El resto de objetos son evaluados a `True` en Python.

---

## Evaluando a `True`

Veamos ciertos ejemplos que son evaluados a `True` en Python:

```
>>> truthiness(True)
True is True

>>> truthiness(1e-10)
1e-10 is True

>>> truthiness([0])
[0] is True

>>> truthiness(( ))
() is True

>>> truthiness(' ')
is True

>>> truthiness('duck')
duck is True
```

### 6.1.3 Parámetros y argumentos

Si una función no dispusiera de valores de entrada estaría muy limitada en su actuación. Es por ello que los **parámetros** nos permiten variar los datos que consume una función para obtener distintos resultados. Vamos a empezar a crear funciones que reciben **parámetros**.

En este caso escribiremos una función que recibe un valor numérico y devuelve su raíz cuadrada:

```
>>> def sqrt(value):
...     return value ** (1/2)
... 
```

(continué en la próxima página)

(provine de la página anterior)

```
>>> sqrt(4)
2.0
```

**Nota:** En este caso, el valor 4 es un **argumento** de la función.

Cuando llamamos a una función con *argumentos*, los valores de estos argumentos se copian en los correspondientes *parámetros* dentro de la función:

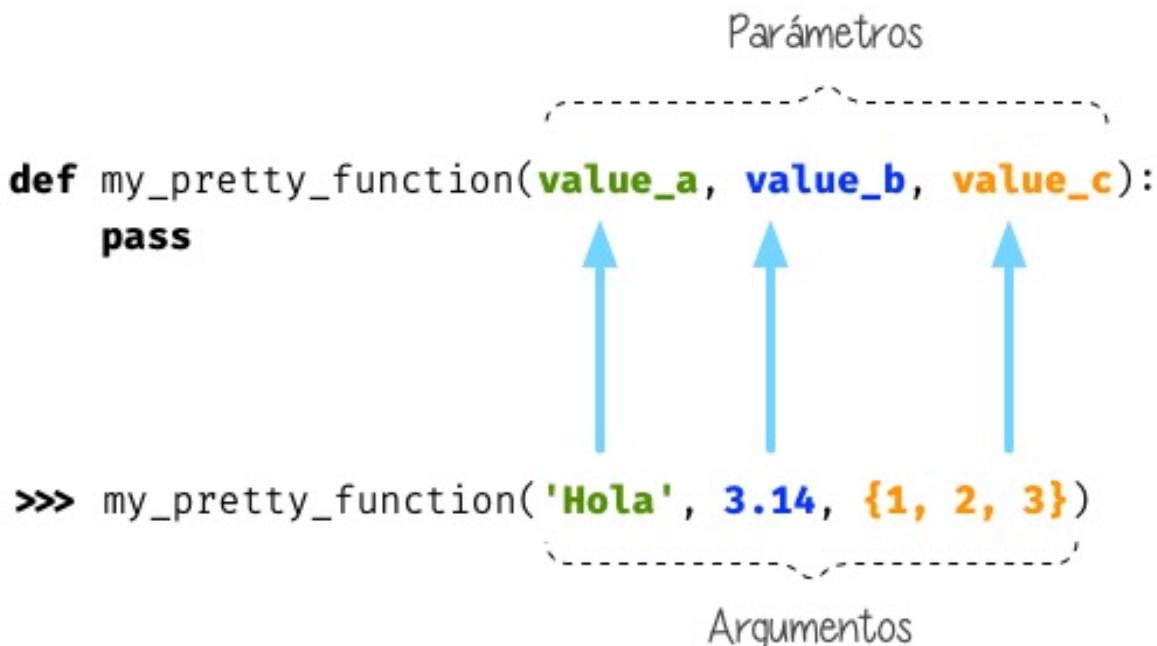


Figura 2: Parámetros y argumentos de una función

**Truco:** La sentencia `pass` permite «no hacer nada». Es una especie de «*placeholder*».

Veamos otra función con dos parámetros y algo más de lógica de negocio:<sup>2</sup>

```
>>> def _min(a, b):
...     if a < b:
...         return a
```

(continué en la próxima página)

<sup>2</sup> Término para identificar el «algoritmo» o secuencia de instrucciones derivadas del procesamiento que corresponda.

(provien de la página anterior)

```
...     else:
...         return b
...
>>> _min(7, 9)
7
```

---

## Ejercicio

Escriba una función en Python que reproduzca lo siguiente:

$$f(x, y) = x^2 + y^2$$

### Ejemplo

---

- Entrada: 3 y 4
  - Salida: 25
- 

## Argumentos posicionales

Los **argumentos posicionales** son aquellos argumentos que se copian en sus correspondientes parámetros **en orden**.

Vamos a mostrar un ejemplo definiendo una función que construye una «cpu» a partir de 3 parámetros:

```
>>> def build_cpu(vendor, num_cores, freq):
...     return dict(
...         vendor=vendor,
...         num_cores=num_cores,
...         freq=freq
...     )
...
```

Una posible llamada a la función con argumentos posicionales sería la siguiente:

```
>>> build_cpu('AMD', 8, 2.7)
{'vendor': 'AMD', 'num_cores': 8, 'freq': 2.7}
```

Lo que ha sucedido es un **mapeo** directo entre argumentos y parámetros en el mismo orden que estaban definidos:

Parámetro	Argumento
vendor	AMD
num_cores	8
freq	2.7

Pero es evidente que una clara desventaja del uso de argumentos posicionales es que se necesita **recordar el orden** de los argumentos. Un error en la posición de los argumentos puede causar resultados indeseados:

```
>>> build_cpu(8, 2.7, 'AMD')
{'vendor': 8, 'num_cores': 2.7, 'freq': 'AMD'}
```

## Argumentos nominales

En esta aproximación los argumentos no son copiados en un orden específico sino que **se asignan por nombre a cada parámetro**. Ello nos permite salvar el problema de conocer cuál es el orden de los parámetros en la definición de la función. Para utilizarlo, basta con realizar una asignación de cada argumento en la propia llamada a la función.

Veamos la misma llamada que hemos hecho en el ejemplo de construcción de la «cpu» pero ahora utilizando paso de argumentos nominales:

```
>>> build_cpu(vendor='AMD', num_cores=8, freq=2.7)
{'vendor': 'AMD', 'num_cores': 8, 'freq': 2.7}
```

Se puede ver claramente que el orden de los argumentos no influye en el resultado final:

```
>>> build_cpu(num_cores=8, freq=2.7, vendor='AMD')
{'vendor': 'AMD', 'num_cores': 8, 'freq': 2.7}
```

## Argumentos posicionales y nominales

Python permite mezclar argumentos posicionales y nominales en la llamada a una función:

```
>>> build_cpu('INTEL', num_cores=4, freq=3.1)
{'vendor': 'INTEL', 'num_cores': 4, 'freq': 3.1}
```

Pero hay que tener en cuenta que, en este escenario, **los argumentos posicionales siempre deben ir antes** que los argumentos nominales. Esto tiene mucho sentido ya que, de hacerlo así, Python no tendría forma de discernir a qué parámetro corresponde cada argumento:

```
>>> build_cpu(num_cores=4, 'INTEL', freq=3.1)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

### Parámetros por defecto

Es posible especificar **valores por defecto** en los parámetros de una función. En el caso de que no se proporcione un valor al argumento en la llamada a la función, el parámetro correspondiente tomará el valor definido por defecto.

Siguiendo con el ejemplo de la «cpu», podemos asignar  $2.0\text{GHz}$  como frecuencia por defecto. La definición de la función cambiaría ligeramente:

```
>>> def build_cpu(vendor, num_cores, freq=2.0):
...     return dict(
...         vendor=vendor,
...         num_cores=num_cores,
...         freq=freq
...     )
... 
```

Llamada a la función sin especificar frecuencia de «cpu»:

```
>>> build_cpu('INTEL', 2)
{'vendor': 'INTEL', 'num_cores': 2, 'freq': 2.0}
```

Llamada a la función indicando una frecuencia concreta de «cpu»:

```
>>> build_cpu('INTEL', 2, 3.4)
{'vendor': 'INTEL', 'num_cores': 2, 'freq': 3.4}
```

---

**Importante:** Los valores por defecto en los parámetros se calculan cuando se **define** la función, no cuando se **ejecuta**.

---

### Ejercicio

Escriba una función **factorial** que reciba un único parámetro **n** y devuelva su factorial.

*El factorial de un número n se define como:*

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

### Ejemplo

- Entrada: 5

- Salida: 120

## Modificando parámetros mutables

### Nivel avanzado

Hay que tener cuidado a la hora de manejar los parámetros que pasamos a una función ya que podemos obtener resultados indeseados, especialmente cuando trabajamos con *tipos de datos mutables*.

Supongamos una función que añade elementos a una lista que pasamos por parámetro. La idea es que si no pasamos la lista, ésta siempre empiece siendo vacía. Hagamos una serie de pruebas pasando alguna lista como segundo argumento:

```
>>> def buggy(arg, result=[]):
...     result.append(arg)
...     print(result)
...
>>> buggy('a', [])
['a']

>>> buggy('b', [])
['b']

>>> buggy('a', ['x', 'y', 'z'])
['x', 'y', 'z', 'a']

>>> buggy('b', ['x', 'y', 'z'])
['x', 'y', 'z', 'b']
```

Aparentemente todo está funcionando de manera correcta, pero veamos qué ocurre en las siguientes llamadas:

```
>>> def buggy(arg, result=[]):
...     result.append(arg)
...     print(result)
...
>>> buggy('a')
['a']

>>> buggy('b') # Se esperaría ['b']
['a', 'b']
```

Obviamente algo no ha funcionado correctamente. Se esperaría que `result` tuviera una lista vacía en cada ejecución. Sin embargo esto no sucede por estas dos razones:

1. El valor por defecto se establece cuando se define la función.
2. La variable `result` apunta a una zona de memoria en la que se modifican sus valores.

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/MgoQGU3>

A riesgo de perder el *parámetro por defecto*, una posible solución sería la siguiente:

```
>>> def works(arg):
...     result = []
...     result.append(arg)
...     return result
...
>>> works('a')
['a']

>>> works('b')
['b']
```

La forma de arreglar el código anterior utilizando un parámetro con valor por defecto sería utilizar un **tipo de dato inmutable** y tener en cuenta cuál es la primera llamada:

```
>>> def nonbuggy(arg, result=None):
...     if result is None:
...         result = []
...     result.append(arg)
...     print(result)
...
>>> nonbuggy('a')
['a']

>>> nonbuggy('b')
['b']

>>> nonbuggy('a', ['x', 'y', 'z'])
['x', 'y', 'z', 'a']

>>> nonbuggy('b', ['x', 'y', 'z'])
['x', 'y', 'z', 'b']
```

## Empaquetar/Desempaquetar argumentos

### Nivel avanzado

Python nos ofrece la posibilidad de empaquetar y desempaquetar argumentos cuando estamos invocando a una función, tanto para **argumentos posicionales** como para **argumentos nominales**.

Y de este hecho se deriva que podamos utilizar un **número variable de argumentos** en una función, algo que puede ser muy interesante según el caso de uso que tengamos.

## Empaquetar/Desempaquetar argumentos posicionales

Si utilizamos el operador `*` delante del nombre de un parámetro posicional, estaremos indicando que los argumentos pasados a la función se empaqueten en una **tupla**:

```
>>> def test_args(*args):
...     print(f'{args=}')
...
>>> test_args()
args=()

>>> test_args(1, 2, 3, 'pescado', 'salado', 'es')
args=(1, 2, 3, 'pescado', 'salado', 'es')
```

---

**Nota:** El hecho de llamar `args` al parámetro es una convención.

---

También podemos utilizar esta estrategia para establecer en una función una serie de parámetros como *requeridos* y recibir el resto de argumentos como *opcionales y empaquetados*:

```
>>> def sum_all(v1, v2, *args):
...     total = 0
...     for value in (v1, v2) + args:
...         total += value
...     return total
...
>>> sum_all()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: sum_all() missing 2 required positional arguments: 'v1' and 'v2'

>>> sum_all(1, 2)
```

(continué en la próxima página)

(provien de la página anterior)

```
3
```

```
>>> sum_all(5, 9, 3, 8, 11, 21)  
57
```

Existe la posibilidad de usar el asterisco \* en la llamada a la función para **desempaquetar** los argumentos posicionales:

```
>>> def test_args(*args):  
...     print(f'args={args}')  
  
>>> my_args = (4, 3, 7, 9)  
  
>>> test_args(my_args)    # No existe desempaquetado!  
args=((4, 3, 7, 9),)  
  
>>> test_args(*my_args)  # Sí existe desempaquetado!  
args=(4, 3, 7, 9)
```

## Empaquetar/Desempaquetar argumentos nominales

Si utilizamos el operador \*\* delante del nombre de un parámetro nominal, estaremos indicando que los argumentos pasados a la función se empaqueten en un **diccionario**:

```
>>> def test_kwargs(**kwargs):  
...     print(f'kwargs={kwargs}')  
  
>>> test_kwargs()  
kwargs={}  
  
>>> test_kwargs(a=4, b=3, c=7, d=9)  
kwargs={'a': 4, 'b': 3, 'c': 7, 'd': 9}
```

---

**Nota:** El hecho de llamar `kwargs` al parámetro es una convención.

---

Al igual que veíamos previamente, existe la posibilidad de usar doble asterisco \*\* en la llamada a la función, para **desempaquetar** los argumentos nominales:

```
>>> def test_kwargs(**kwargs):  
...     print(f'kwargs={kwargs}')
```

(continué en la próxima página)

(provien de la página anterior)

```

...
>>> my_kwargs = {'a': 4, 'b': 3, 'c': 7, 'd': 9}

>>> test_kwargs(my_kwargs)    # No existe desempaquetado!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: test_kwargs() takes 0 positional arguments but 1 was given

>>> test_kwargs(**my_kwargs)  # Sí existe desempaquetado!
kwargs={'a': 4, 'b': 3, 'c': 7, 'd': 9}

```

## Forzando modo de paso de argumentos

Si bien Python nos da flexibilidad para pasar argumentos a nuestras funciones en modo posicional o nominal, existen opciones para forzar a que dicho paso sea obligatorio en una determinada modalidad.

### Argumentos sólo posicionales

#### Nivel avanzado

A partir de [Python 3.8](#) se ofrece la posibilidad de obligar a que determinados parámetros de la función sean pasados sólo por posición.

Para ello, en la definición de los parámetros de la función, tendremos que incluir un parámetro especial / que delimitará el tipo de parámetros. Así, todos los parámetros a la izquierda del delimitador estarán **obligados** a ser posicionales:

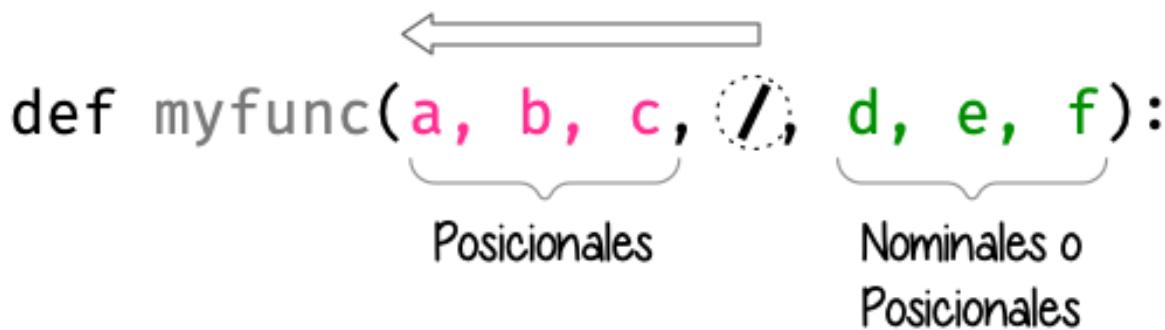


Figura 3: Separador para especificar parámetros sólo posicionales

Ejemplo:

```
>>> def sum_power(a, b, /, power=False):
...     if power:
...         a **= 2
...         b **= 2
...     return a + b
...
...
>>> sum_power(3, 4)
7
>>> sum_power(3, 4, True)
25
>>> sum_power(3, 4, power=True)
25
>>> sum_power(a=3, b=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum_power() got some positional-only arguments passed as keyword_
arguments: 'a, b'
```

## Argumentos sólo nominales

### Nivel avanzado

A partir de Python 3 se ofrece la posibilidad de obligar a que determinados parámetros de la función sean pasados sólo por nombre.

Para ello, en la definición de los parámetros de la función, tendremos que incluir un parámetro especial `*` que delimitará el tipo de parámetros. Así, todos los parámetros a la derecha del separador estarán **obligados** a ser nominales:

Ejemplo:

```
>>> def sum_power(a, b, *, power=False):
...     if power:
...         a **= 2
...         b **= 2
...     return a + b
...
...
>>> sum_power(3, 4)
7
```

(continué en la próxima página)

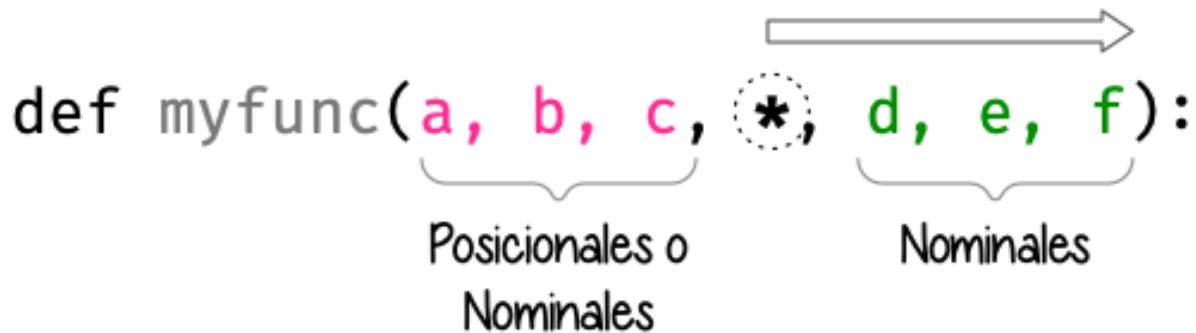


Figura 4: Separador para especificar parámetros sólo nominales

(proviene de la página anterior)

```
>>> sum_power(a=3, b=4)
7

>>> sum_power(3, 4, power=True)
25

>>> sum_power(3, 4, True)
-----
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum_power() takes 2 positional arguments but 3 were given
```

## Fijando argumentos posicionales y nominales

Si mezclamos las dos estrategias anteriores podemos forzar a que una función reciba argumentos de un modo concreto.

Continuando con ejemplo anterior, podríamos hacer lo siguiente:

```
>>> def sum_power(a, b, /, *, power=False):
...     if power:
...         a **= 2
...         b **= 2
...     return a + b
...

>>> sum_power(3, 4, power=True) # Único modo posible de llamada
25
```

## Argumentos mutables e inmutables

### Nivel intermedio

Igual que veímos en la incidencia de *parámetros por defecto con valores mutables*, cuando realizamos modificaciones a los argumentos de una función es importante tener en cuenta si son **mutables** (listas, diccionarios, conjuntos, ...) o **inmutables** (tuplas, enteros, flotantes, cadenas de texto, ...) ya que podríamos obtener efectos colaterales no deseados:

```
>>> fib = [1, 1, 2, 3, 5, 8, 13]

>>> def square_it(values, *, index):
...     values[index] **= 2
...
...
...
>>> fib
[1, 1, 2, 3, 5, 8, 13]

>>> square_it(fib, index=4)

>>> fib # 😱
[1, 1, 2, 3, 25, 8, 13]
```

**Advertencia:** Esto **no es una buena práctica**. O bien documentar que el argumento puede modificarse o bien retornar un nuevo valor.

## Funciones como parámetros

### Nivel avanzado

Las funciones se pueden utilizar en cualquier contexto de nuestro programa. Son objetos que pueden ser asignados a variables, usados en expresiones, devueltos como valores de retorno o pasados como argumentos a otras funciones.

Veamos un primer ejemplo en el que pasamos una función como argumento:

```
>>> def success():
...     print('Yeah! ')
...
...
...
>>> type(success)
function

>>> def doit(f):
```

(continué en la próxima página)

(provine de la página anterior)

```

...
    f()
...
>>> doit(success)
Yeah!

```

Veamos un segundo ejemplo en el que pasamos, no sólo una función como argumento, sino los valores con los que debe operar:

```

>>> def repeat_please(text, times=1):
...     return text * times
...
>>> type(repeat_please)
function

>>> def doit(f, arg1, arg2):
...     return f(arg1, arg2)
...
>>> doit(repeat_please, 'Functions as params', 2)
'Functions as paramsFunctions as params'

```

## 6.1.4 Documentación

Ya hemos visto que en Python podemos incluir *comentarios* para explicar mejor determinadas zonas de nuestro código.

Del mismo modo podemos (y en muchos casos **debemos**) adjuntar **documentación** a la definición de una función incluyendo una cadena de texto (**docstring**) al comienzo de su cuerpo:

```

>>> def sqrt(value):
...     'Returns the square root of the value'
...     return value ** (1/2)
...

```

La forma más ortodoxa de escribir un **docstring** es utilizando *triples comillas*:

```

>>> def closest_int(value):
...     '''Returns the closest integer to the given value.
...     The operation is:
...         1. Compute distance to floor.
...         2. If distance less than a half, return floor.
...

```

(continué en la próxima página)

(provien de la página anterior)

```
...     Otherwise, return ceil.  
...     ...  
...     floor = int(value)  
...     if value - floor < 0.5:  
...         return floor  
...     else:  
...         return floor + 1  
...     ...
```

Para ver el `docstring` de una función, basta con utilizar `help`:

```
>>> help(closest_int)

Help on function closest_int in module __main__:

closest_int(value)
    Returns the closest integer to the given value.
    The operation is:
        1. Compute distance to floor.
        2. If distance less than a half, return floor.
        Otherwise, return ceil.
```

También es posible extraer información usando el símbolo de interrogación:

```
>>> closest_int?
Signature: closest_int(value)
Docstring:
Returns the closest integer to the given value.
The operation is:
    1. Compute distance to floor.
    2. If distance less than a half, return floor.
    Otherwise, return ceil.
File:      ~/aprendepython/<ipython-input-75-5dc166360da1>
Type:      function
```

---

**Importante:** Esto no sólo se aplica a funciones propias, sino a cualquier otra función definida en el lenguaje.

---

---

**Nota:** Si queremos ver el `docstring` de una función en «crudo» (sin formatear), podemos usar `<function>.__doc__`.

---

## Explicación de parámetros

Como ya se ha visto, es posible documentar una función utilizando un **docstring**. Pero la redacción y el formato de esta cadena de texto puede ser muy variada. Existen distintas formas de documentar una función (u otros objetos)<sup>3</sup>:

**Sphinx docstrings** Formato nativo de documentación Sphinx.

**Google docstrings** Formato de documentación recomendado por Google.

**NumPy-SciPy docstrings** Combinación de formatos reStructured y Google (usados por el proyecto NumPy).

**Epytext** Una adaptación a Python de Epydoc(Java).

Aunque cada uno tiene sus particularidades, todos comparten una misma estructura:

- Una primera línea de **descripción de la función**.
- A continuación especificamos las características de los **parámetros** (incluyendo sus tipos).
- Por último, indicamos si la función **retorna un valor** y sus características.

Aunque todos los formatos son válidos, nos centraremos en **Sphinx docstrings** al ser el que viene mejor integrado con la documentación Sphinx. *Google docstrings* y *Numpy docstrings* también son ampliamente utilizados, lo único es que necesitan de un módulo externo denominado **Napoleon** para que se puedan incluir en la documentación *Sphinx*.

## Sphinx

Sphinx es una herramienta para generar documentación e incluye un módulo «built-in» denominado **autodoc** el cual permite la autogeneración de documentación a partir de los «docstrings» definidos en el código.

Veamos el uso de este formato en la documentación de la siguiente función «dummy»:

```
>>> def my_power(x, n):
...     '''Calculate x raised to the power of n.
...
...     :param x: number representing the base of the operation
...     :type x: int
...     :param n: number representing the exponent of the operation
...     :type n: int
...
...     :return: :math:`x^n`
...     :rtype: int
```

(continué en la próxima página)

<sup>3</sup> Véase Docstring Formats.

(provine de la página anterior)

```
...
...
    result = 1
    for _ in range(n):
        result *= x
    return result
...
```

Dentro del «docstring» podemos escribir con sintaxis reStructured Text – véase por ejemplo la expresión matemática en el tag :return: – lo que nos proporciona una gran flexibilidad.

---

**Nota:** La plataforma [Read the Docs](#) aloja la documentación de gran cantidad de proyectos. En muchos de los casos se han usado «docstrings» con el formato Sphinx visto anteriormente.

---

## Anotación de tipos

### Nivel intermedio

Las anotaciones de tipos<sup>5</sup> se introdujeron en [Python 3.5](#) y permiten indicar tipos para los parámetros de una función así como su valor de retorno (aunque también funcionan en creación de variables).

Veamos un ejemplo en el que creamos una función para dividir una cadena de texto por la posición especificada en el parámetro:

```
>>> def ssplit(text: str, split_pos: int) -> tuple:
...     return text[:split_pos], text[split_pos:]
...
>>> ssplit('Always remember us this way', 15)
('Always remember', ' us this way')
```

Como se puede observar, vamos añadiendo los tipos después de cada parámetro utilizando : como separador. En el caso del valor de retorno usamos el símbolo ->

Quizás la siguiente ejecución pueda sorprender:

```
>>> ssplit([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 5)
([1, 2, 3, 4, 5], [6, 7, 8, 9, 10])
```

Efectivamente como habrás visto, **no hemos obtenido ningún error**, a pesar de que estamos pasando como primer argumento una lista en vez de una cadena de texto. Esto ocurre porque lo que hemos definido es una anotación de tipo, no una declaración de tipo. Existen herramientas como [mypy](#) que sí se encargan de chequear estas situaciones.

---

<sup>5</sup> Conocidos como «type hints» en terminología inglesa.

## Valores por defecto

Al igual que ocurre en la definición ordinaria de funciones, cuando usamos anotaciones de tipos también podemos indicar un valor por defecto para los parámetros.

Veamos la forma de hacerlo continuando con el ejemplo anterior:

```
>>> def ssplit(text: str, split_pos: int = -1) -> tuple:
...     if split_pos == -1:
...         split_pos = len(text) // 2
...     return text[:split_pos], text[split_pos:]
...
>>> ssplit('Always remember us this way')
('Always rememb', 'er us this way')
```

Simplemente añadimos el valor por defecto después de indicar el tipo.

---

**Nota:** Las **anotaciones de tipos** son una herramienta muy potente y que, usada de forma adecuada, permite complementar la documentación de nuestro código y aclarar ciertos aspectos, que a priori, pudieran parecer confusos. Su aplicación estará en función de la necesidad detectada por parte del equipo de desarrollo.

---

### 6.1.5 Tipos de funciones

Nivel avanzado

#### Funciones interiores

Está permitido definir una función dentro de otra función:

```
>>> def validation_test(text):
...     def is_valid_char(char):
...         return char in 'xyz'
...     checklist = []
...     for char in text:
...         checklist.append(is_valid_char(char))
...     return sum(checklist) / len(text)
...
>>> validation_test('zxyzxxyz')
1.0
```

(continué en la próxima página)

(provien de la página anterior)

```
>>> validation_test('abzxyabcdz')
0.4

>>> validation_test('abc')
0.0
```

## Clausuras

Una **clausura** (del término inglés «*closure*») establece el uso de una *función interior* que se genera dinámicamente y recuerda los valores de los argumentos con los que fue creada:

```
>>> def make_multiplier_of(n):
...     def multiplier(x):
...         return x * n
...     return multiplier
...

>>> m3 = make_multiplier_of(3)

>>> m5 = make_multiplier_of(5)

>>> type(m3)
function

>>> m3(7)  # 7 * 3
21

>>> type(m5)
function

>>> m5(8)  # 8 * 5
40
```

---

**Importante:** En una clausura retornamos una función, no una llamada a la función.

---

## Funciones anónimas «lambda»

Una función lambda tiene las siguientes propiedades:

1. Se escribe con una única sentencia.
2. No tiene nombre (anónima).
3. Su cuerpo tiene implícito un `return`.
4. Puede recibir cualquier número de parámetros.

Veamos un primer ejemplo de función «lambda» que nos permite contar el número de palabras de una cadena de texto:

```
>>> num_words = lambda t: len(t.strip().split())

>>> type(num_words)
function

>>> num_words
<function __main__.<lambda>(t)

>>> num_words('hola socio vamos a ver')
5
```

Veamos otro ejemplo en el que mostramos una tabla con el resultado de aplicar el «and» lógico mediante una función «lambda» que ahora recibe dos parámetros:

```
>>> logic_and = lambda x, y: x & y

>>> for i in range(2):
...     for j in range(2):
...         print(f'{i} & {j} = {logic_and(i, j)}')
...
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
```

Las funciones «lambda» son bastante utilizadas como argumentos a otras funciones. Un ejemplo claro de ello es la función `sorted` que tiene un parámetro opcional `key` donde se define la clave de ordenación.

Veamos cómo usar una función anónima «lambda» para ordenar una tupla de pares *longitud-latitud*:

```
>>> geoloc = (
... (15.623037, 13.258358),
```

(continué en la próxima página)

(proviene de la página anterior)

```
... (55.147488, -2.667338),
... (54.572062, -73.285171),
... (3.152857, 115.327724),
... (-40.454262, 172.318877)
)

>>> # Ordenación por longitud (primer elemento de la tupla)
>>> sorted(geoloc)
[(-40.454262, 172.318877),
 (3.152857, 115.327724),
 (15.623037, 13.258358),
 (54.572062, -73.285171),
 (55.147488, -2.667338)]

>>> # Ordenación por latitud (segundo elemento de la tupla)
>>> sorted(geoloc, key=lambda t: t[1])
[(54.572062, -73.285171),
 (55.147488, -2.667338),
 (15.623037, 13.258358),
 (3.152857, 115.327724),
 (-40.454262, 172.318877)]
```

## Enfoque funcional

Como se comentó en la [introducción](#), Python es un lenguaje de programación multiparadigma. Uno de los paradigmas menos explotados en este lenguaje es la **programación funcional**<sup>4</sup>.

Python nos ofrece 3 funciones que encajan verdaderamente bien en este enfoque: `map()`, `filter()` y `reduce()`.

### `map()`

Esta función **aplica otra función** sobre cada elemento de un iterable. Supongamos que queremos aplicar la siguiente función:

$$f(x) = \frac{x^2}{2} \quad \forall x \in [1, 10]$$

```
>>> def f(x):
...     return x**2 / 2
```

(continué en la próxima página)

<sup>4</sup> Definición de *Programación funcional* en Wikipedia.

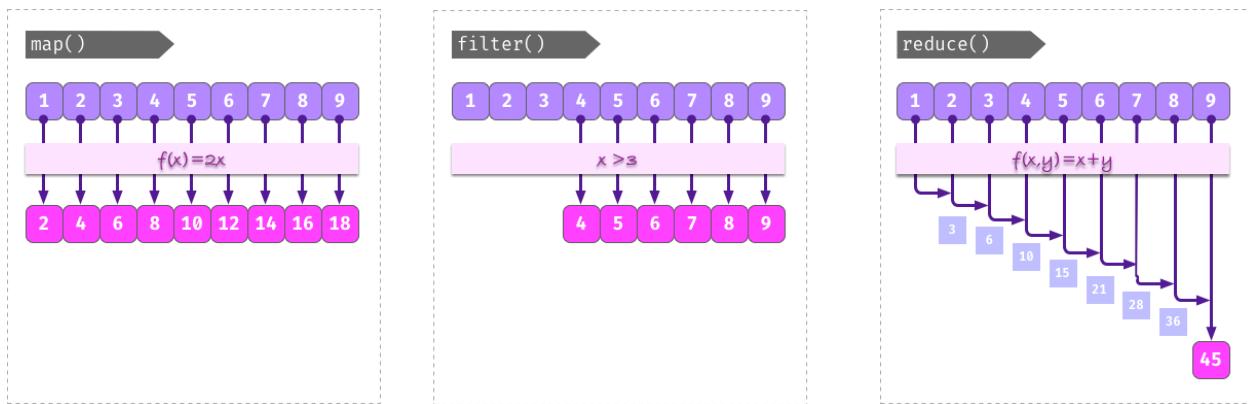


Figura 5: Rutinas muy enfocadas a programación funcional

(proviene de la página anterior)

```
...
>>> data = range(1, 11)
>>> map_gen = map(f, data)
>>> type(map_gen)
map
>>> list(map_gen)
[0.5, 2.0, 4.5, 8.0, 12.5, 18.0, 24.5, 32.0, 40.5, 50.0]
```

Aplicando una *función anónima «lambda»*...

```
>>> list(map(lambda x: x**2 / 2, data))
[0.5, 2.0, 4.5, 8.0, 12.5, 18.0, 24.5, 32.0, 40.5, 50.0]
```

**Importante:** `map()` devuelve un **generador**, no directamente una lista.

### filter()

Esta función **selecciona** aquellos elementos de un iterable que cumplan una determinada condición. Supongamos que queremos seleccionar sólo aquellos números impares dentro de un rango:

```
>>> def odd_number(x):
...     return x % 2 == 1
```

(continué en la próxima página)

(proviene de la página anterior)

```
...  
  
=> data = range(1, 21)  
  
=> filter_gen = filter(odd_number, data)  
  
=> type(filter_gen)  
filter  
  
=> list(filter_gen)  
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Aplicando una *función anónima «lambda»*...

```
>>> list(filter(lambda x: x % 2 == 1, data))  
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

---

**Importante:** `filter()` devuelve un **generador**, no directamente una lista.

---

## reduce()

Para poder usar esta función debemos usar el módulo `functools`. Nos permite aplicar una función dada sobre todos los elementos de un iterable de manera acumulativa. O dicho en otras palabras, nos permite **reducir** una función sobre un conjunto de valores. Supongamos que queremos realizar el producto de una serie de valores aplicando este enfoque:

```
>>> from functools import reduce  
  
>>> def mult_values(a, b):  
...     return a * b  
...  
>>> data = range(1, 6)  
  
>>> reduce(mult_values, data) # (((1 * 2) * 3) * 4) * 5  
120
```

Aplicando una *función anónima «lambda»*...

```
>>> reduce(lambda x, y: x * y, data)  
120
```

---

**Consejo:** Por cuestiones de legibilidad del código, se suelen preferir las **listas por comprensión** a funciones como `map()` o `filter()`, aunque cada problema tiene sus propias características y sus soluciones más adecuadas.

---

## Generadores

Un **generador** es un objeto que nos permite iterar sobre una *secuencia de valores* con la particularidad de no tener que crear explícitamente dicha secuencia. Esta propiedad los hace idóneos para situaciones en las que el tamaño de las secuencias podría tener un impacto negativo en el consumo de memoria.

De hecho ya hemos visto algunos generadores y los hemos usado de forma directa. Un ejemplo es `range()` que ofrece la posibilidad de crear *secuencias de números*.

Básicamente existen dos implementaciones de generadores:

- Funciones generadoras.
- Expresiones generadoras.

---

**Nota:** A diferencia de las funciones ordinarias, los generadores tienen la capacidad de «recordar» su estado para recuperarlo en la siguiente iteración y continuar devolviendo nuevos valores.

---

## Funciones generadoras

Las funciones generadoras se escriben como funciones ordinarias con el matiz de incorporar la sentencia `yield` que sustituye, de alguna manera, a `return`. Esta sentencia devuelve el valor indicado y, a la vez, «congela» el estado de la función para subsiguientes ejecuciones.

Veamos un ejemplo en el que escribimos una función generadora de números pares:

```
>>> def evens(lim):
...     for i in range(0, lim + 1, 2):
...         yield i
...
>>> type(evens)
function

>>> evens_gen = evens(20)  # returns generator
```

(continué en la próxima página)

(provien de la página anterior)

```
>>> type(evens_gen)
generator
```

Una vez creado el generador, ya podemos iterar sobre él:

```
>>> for i in evens_gen:
...     print(i, end=' ')
...
0 2 4 6 8 10 12 14 16 18 20
```

Si queremos «explicitar» la lista de valores que contiene un generador, podemos hacerlo de la siguiente manera:

```
>>> list(evens(20))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

---

**Importante:** Un detalle muy importante sobre los generadores es que «se agotan». Es decir, una vez que ya hemos consumido todos sus elementos ya no obtendremos nuevos valores.

---

## Expresiones generadoras

Una **expresión generadora** es sintácticamente muy similar a una *lista por comprensión*, pero utilizamos **paréntesis** en vez de corchetes. Se podría ver como una versión acortada de una función generadora.

Podemos tratar de reproducir el ejemplo visto en *funciones generadoras* en el que creamos números pares hasta el 20:

```
>>> evens_gen = (i for i in range(0, 20, 2))

>>> type(evens_gen)
generator

>>> for i in evens_gen:
...     print(i, end=' ')
...
0 2 4 6 8 10 12 14 16 18
```

---

**Nota:** Las expresiones generadoras admiten *condiciones* y *anidamiento de bucles*, tal y como se vio con las listas por comprensión.

---

---

## Ejercicio

Escriba una **función generadora** que devuelva los 100 primeros números enteros elevados al cuadrado.

---

## Decoradores

Hay situaciones en las que necesitamos modificar el comportamiento de funciones existentes pero sin alterar su código. Para estos casos es muy útil usar decoradores.

Un **decorador** es una *función* que recibe como parámetro una función y devuelve otra función. Se podría ver como un caso particular de *clausura*.

Veamos un ejemplo en el que documentamos la ejecución de una función:

```
>>> def simple_logger(func):
...     def wrapper(*args, **kwargs):
...         print(f'Running "{func.__name__}"...')
...         return func(*args, **kwargs)
...     return wrapper
...
>>> type(simple_logger)
function
```

Ahora vamos a definir una función ordinaria (que usaremos más adelante):

```
>>> def hi(name):
...     return f'Hello {name}!'
...
>>> hi('Guido')
Hello Guido!
>>> hi('Lovelace')
Hello Lovelace!
```

Ahora aplicaremos el decorador definido previamente `simple_logger()` sobre la función ordinaria `hi()`. Se dice que que `simple_logger()` es la **función decoradora** y que `hi()` es la **función decorada**. De esta forma obtendremos mensajes informativos adicionales. Además el decorador es aplicable a cualquier número y tipo de argumentos e incluso a cualquier otra función ordinaria:

```
>>> decorated_hi = simple_logger(hi)
```

(continué en la próxima página)

(provien de la página anterior)

```
>>> decorated_hi('Guido')
Running "hi"...
'Hello Guido!'

>>> decorated_hi('Lovelace')
Running "hi"...
'Hello Lovelace!'
```

## Usando @ para decorar

Python nos ofrece un «syntactic sugar» para simplificar la aplicación de los decoradores a través del operador @ justo antes de la definición de la función que queremos decorar:

```
>>> @simple_logger
... def hi(name):
...     return f'Hello {name}!'
...
...

>>> hi('Galindo')
Running "hi"...
'Hello Galindo!'

>>> hi('Terrón')
Running "hi"...
'Hello Terrón!'
```

Podemos aplicar más de un decorador a cada función. Para exemplificarlo vamos a crear dos decoradores muy sencillos:

```
>>> def plus5(func):
...     def wrapper(*args, **kwargs):
...         result = func(*args, **kwargs)
...         return result + 5
...     return wrapper
...

>>> def div2(func):
...     def wrapper(*args, **kwargs):
...         result = func(*args, **kwargs)
...         return result // 2
...     return wrapper
...
```

Ahora aplicaremos ambos decoradores sobre una función que realiza el producto de dos números:

```
>>> @plus5
... @div2
... def prod(a, b):
...     return a * b
...
>>> prod(4, 3)
11
>>> ((4 * 3) // 2) + 5
11
```

---

**Importante:** Cuando tenemos varios decoradores aplicados a una función, el orden de ejecución empieza por aquel decorador más «cercano» a la definición de la función.

---

## Ejercicio

Escriba un decorador llamado `fabs()` que convierta a su valor absoluto los dos primeros parámetros de la función que decora y devuelva el resultado de aplicar dicha función a sus dos argumentos. *El valor absoluto de un número se obtiene con la función `abs()`.*

A continuación probar el decorador con una función `fprod()` que devuelva el producto de dos valores, jugando con números negativos y positivos.

*¿Podrías extender el decorador para que tuviera en cuenta un número indeterminado de argumentos posicionales?*

## Ejemplo

- Entrada: -3 y 7
  - Salida: 21
- 

## Funciones recursivas

La **recursividad** es el mecanismo por el cual una función se llama a sí misma:

```
>>> def call_me():
...     return call_me()
...
```

(continué en la próxima página)

(provine de la página anterior)

```
>>> call_me()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in call_me
  File "<stdin>", line 2, in call_me
  File "<stdin>", line 2, in call_me
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

**Advertencia:** Podemos observar que existe un número máximo de llamadas recursivas. Python controla esta situación por nosotros, ya que, de no ser así, podríamos llegar a consumir los recursos del sistema.

Veamos ahora un ejemplo más real en el que computar el enésimo término de la Sucesión de Fibonacci utilizando una función recursiva:

```
>>> def fibonacci(n):
...     if n == 0:
...         return 0
...     if n == 1:
...         return 1
...     return fibonacci(n - 1) + fibonacci(n - 2)
...
...
>>> fibonacci(10)
55
>>> fibonacci(20)
6765
```

## Función generadora recursiva

Si tratamos de extender el ejemplo anterior de Fibonacci para obtener todos los términos de la sucesión hasta un límite, pero con la filosofía recursiva, podríamos plantear el uso de una *función generadora*:

```
>>> def fibonacci():
...     def _fibonacci(n):
...         if n == 0:
...             return 0
...         if n == 1:
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     return 1
...     return _fibonacci(n - 1) + _fibonacci(n - 2)
...
...     n = 0
...     while True:
...         yield _fibonacci(n)
...         n += 1
...
>>> fib = fibonacci()

>>> type(fib)
generator

>>> for _ in range(10):
...     print(next(fib))
...
0
1
1
2
3
5
8
13
21
34
```

---

## Ejercicio

Escriba una función recursiva que calcule el factorial de un número:

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

## Ejemplo

- Entrada: 5
  - Salida: 120
-

### 6.1.6 Espacios de nombres

Como bien indica el *Zen de Python*:

*Namespaces are one honking great idea – let's do more of those!*

Que vendría a traducirse como: «Los espacios de nombres son una gran idea – hagamos más de eso». Los **espacios de nombres** permiten definir **ámbitos** o **contextos** en los que agrupar nombres de objetos.

Los espacios de nombres proporcionan un mecanismo de empaquetamiento, de tal forma que podamos tener incluso nombres iguales que no hacen referencia al mismo objeto (siempre y cuando estén en ámbitos distintos).

Cada *función* define su propio espacio de nombres y es diferente del espacio de nombres global aplicable a todo nuestro programa.

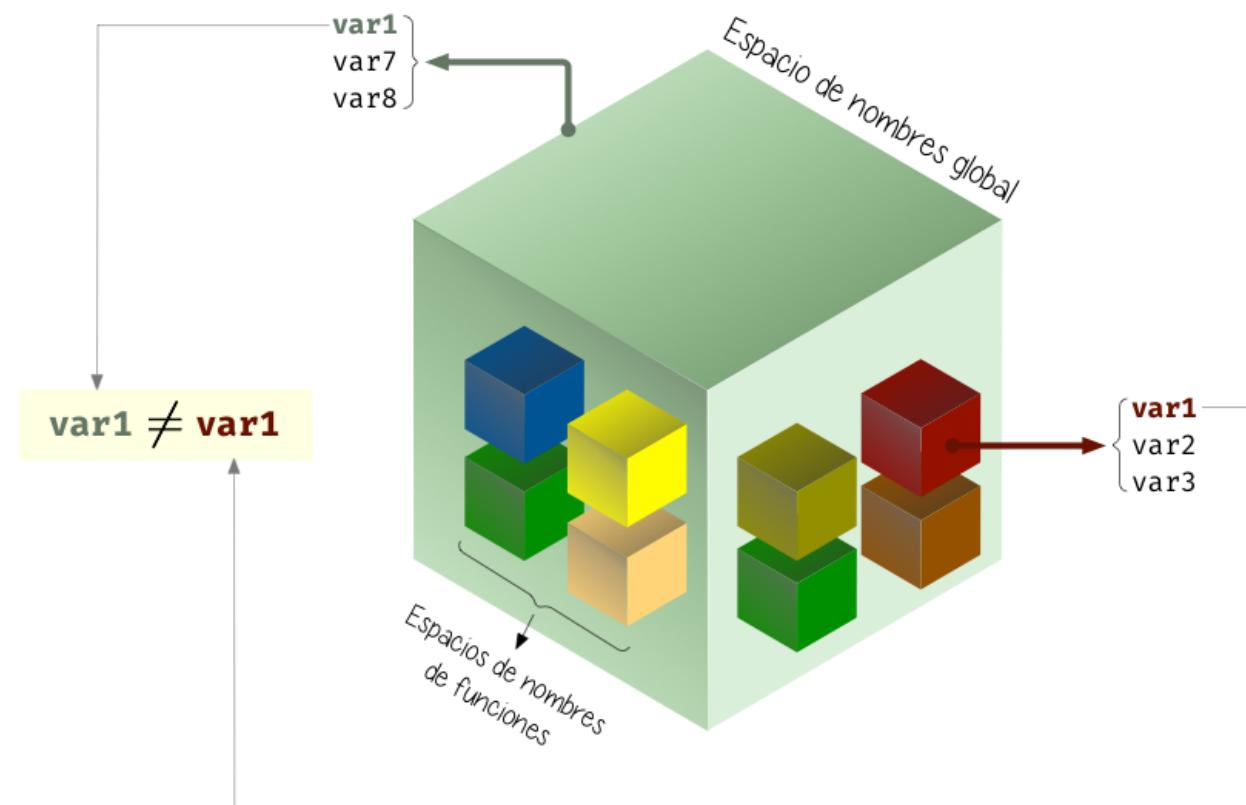


Figura 6: Espacio de nombres global vs espacios de nombres de funciones

## Acceso a variables globales

Cuando una variable se define en el *espacio de nombres global* podemos hacer uso de ella con total transparencia dentro del ámbito de las funciones del programa:

```
>>> language = 'castellano'

>>> def catalonia():
...     print(f'{language=}')

...

>>> language
'castellano'

>>> catalonia()
language='castellano'
```

## Creando variables locales

En el caso de que asignemos un valor a una variable global dentro de una función, no estaremos modificando ese valor. Por el contrario, estaremos creando una *variable en el espacio de nombres local*:

```
>>> language = 'castellano'

>>> def catalonia():
...     language = 'catalan'
...     print(f'{language=}')

...

>>> language
'castellano'

>>> catalonia()
language='catalan'

>>> language
'castellano'
```

## Forzando modificación global

Python nos permite modificar una variable definida en un espacio de nombres global dentro de una función. Para ello debemos usar el modificador `global`:

```
>>> language = 'castellano'

>>> def catalonia():
...     global language
...     language = 'catalan'
...     print(f'{language=}')
...
...
>>> language
'castellano'

>>> catalonia()
language='catalan'

>>> language
'catalan'
```

**Advertencia:** El uso de `global` no se considera una buena práctica ya que puede inducir a confusión y tener efectos colaterales indeseados.

## Contenido de los espacios de nombres

Python proporciona dos funciones para acceder al contenido de los espacios de nombres:

`locals()` Devuelve un diccionario con los contenidos del **espacio de nombres local**.

`globals()` Devuelve un diccionario con los contenidos del **espacio de nombres global**.

```
>>> language = 'castellano'

>>> def catalonia():
...     language = 'catalan'
...     print(f'{locals()=}')
...
...
>>> language
'castellano'

>>> catalonia()
```

(continué en la próxima página)

(provien de la página anterior)

```
locals()={'language': 'catalan'}
```

```
>>> globals()
{'__name__': '__main__',
 '__doc__': 'Automatically created module for IPython interactive environment',
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
 '__builtins__': <module 'builtins' (built-in)>,
 '_ih': [],
 "language = 'castellano'",
 "def catalonia():\n    language = 'catalan'\n    print(f'{locals()=}')\n",
 'language',
 'catalonia()',
 'globals()'],
 '_oh': {3: 'castellano'},
 '_dh': ['/Users/sdelquin'],
 'In': [],
 "language = 'castellano'",
 "def catalonia():\n    language = 'catalan'\n    print(f'{locals()=}')\n",
 'language',
 'catalonia()',
 'globals()'],
 'Out': {3: 'castellano'},
 'get_ipython': <bound method InteractiveShell.get_ipython of <IPython.terminal.
↳interactiveshell.TerminalInteractiveShell object at 0x10e70c2e0>>,
 'exit': <IPython.core.autocall.ExitAutocall at 0x10e761070>,
 'quit': <IPython.core.autocall.ExitAutocall at 0x10e761070>,
 '_': 'castellano',
 '_': '',
 '_': '',
 'Prompts': IPython.terminal.prompts.Prompts,
 'Token': Token,
 'MyPrompt': __main__.MyPrompt,
 'ip': <IPython.terminal.interactiveshell.TerminalInteractiveShell at 0x10e70c2e0>,
 '_i': 'catalonia()',
 '_ii': 'language',
 '_iii': "def catalonia():\n    language = 'catalan'\n    print(f'{locals()=}')\n",
 '_i1': "language = 'castellano'",
 'language': 'castellano',
 '_i2': "def catalonia():\n    language = 'catalan'\n    print(f'{locals()=}')\n",
 '_catalonia': <function __main__.catalonia()>,
```

(continué en la próxima página)

(provien de la página anterior)

```
'_i3': 'language',
'_3': 'castellano',
'_i4': 'catalonia()',  
'_i5': 'globals()'}
```

## EJERCICIOS DE REPASO

1. Escriba una función en Python que indique si un número está en un determinado intervalo ([solución](#)).

Entrada: valor=3; lim\_inferior=2; lim\_superior=5

Salida: True

2. Escriba una función en Python que reciba una lista de valores enteros y devuelva otra lista sólo con aquellos valores pares ([solución](#)).

Entrada: [1, 2, 3, 4, 5, 6, 7, 8, 9]

Salida: [2, 4, 6, 8]

3. Escriba una función en Python que indique si un número es perfecto. *Utilice una función auxiliar que calcule los divisores propios* ([solución](#)).

Entrada: 8128

Salida: True

4. Escriba una función en Python que determine si una cadena de texto es un [palíndromo](#) ([solución](#)).

Entrada: ana lava lana

Salida: True

5. Escriba una función en Python que determine si una cadena de texto es un [pangrama](#) ([solución](#))

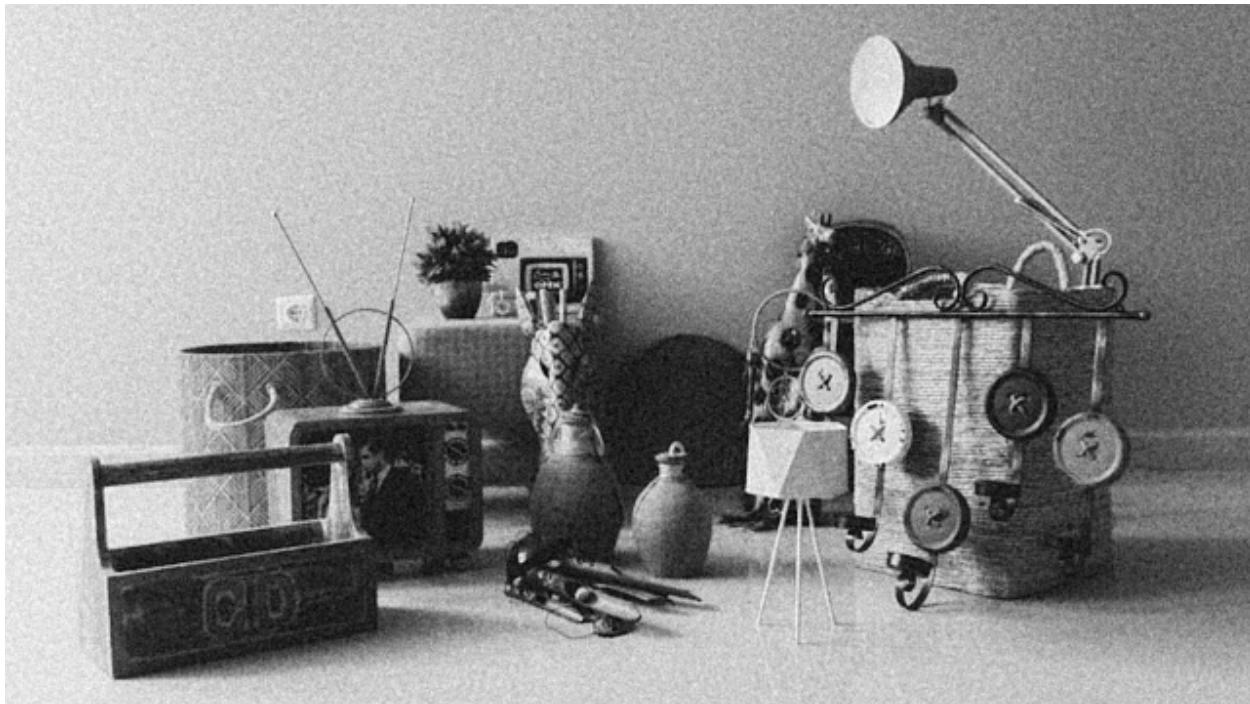
Entrada: The quick brown fox jumps over the lazy dog

Salida: True

## AMPLIAR CONOCIMIENTOS

- Comparing Python Objects the Right Way: «is» vs «==»
- Python Scope & the LEGB Rule: Resolving Names in Your Code
- Defining Your Own Python Function
- Null in Python: Understanding Python's NoneType Object
- Python “!=” Is Not “is not”: Comparing Objects in Python
- Python args and kwargs: Demystified
- Documenting Python Code: A Complete Guide
- Thinking Recursively in Python
- How to Use Generators and yield in Python
- How to Use Python Lambda Functions
- Python Decorators 101
- Writing Comments in Python
- Introduction to Python Exceptions
- Primer on Python Decorators

## 6.2 Objetos y Clases



Hasta ahora hemos estado usando objetos de forma totalmente transparente, casi sin ser conscientes de ello. Pero, en realidad, **todo en Python es un objeto**, desde números a funciones. El lenguaje provee ciertos mecanismos para no tener que usar explícitamente técnicas de orientación a objetos.

Llegados a este punto, investigaremos en profundidad sobre la creación y manipulación de clases y objetos, y todas las operaciones que engloban este paradigma.<sup>1</sup>

### 6.2.1 Programación orientada a objetos

La programación orientada a objetos (POO) o en sus siglas inglesas **OOP** es una manera de programar que permite llevar al código mecanismos usados con entidades de la vida real.

Sus **beneficios** son los siguientes:

**Encapsulamiento** Permite **empaquetar** el código dentro de una unidad (objeto) donde se puede determinar el ámbito de actuación.

**Abstracción** Permite **generalizar** los tipos de objetos a través de las clases y simplificar el programa.

**Herencia** Permite **reutilizar** código al poder heredar atributos y comportamientos de una clase a otra.

---

<sup>1</sup> Foto original por Rabie Madaci en Unsplash.

**Polimorfismo** Permite **crear** múltiples objetos a partir de una misma pieza flexible de código.

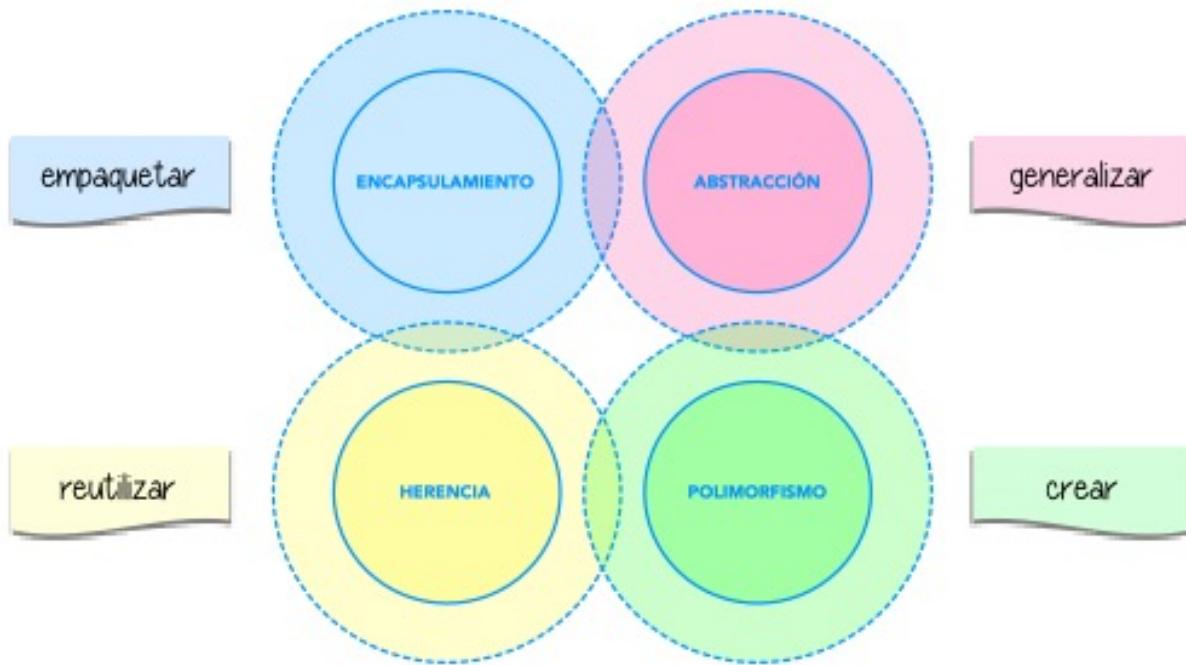


Figura 7: Beneficios de la Programación Orientada a Objetos

### ¿Qué es un objeto?

Un **objeto** es una **estructura de datos personalizada** que contiene **datos y código**:

Elementos	¿Qué son?	¿Cómo se llaman?	¿Cómo se identifican?
Datos	Variables	Atributos	Nombres
Código	Funciones	Métodos	Verbos

Un objeto representa **una instancia única** de alguna entidad a través de los valores de sus atributos e interactúan con otros objetos (o consigo mismos) a través de sus métodos.

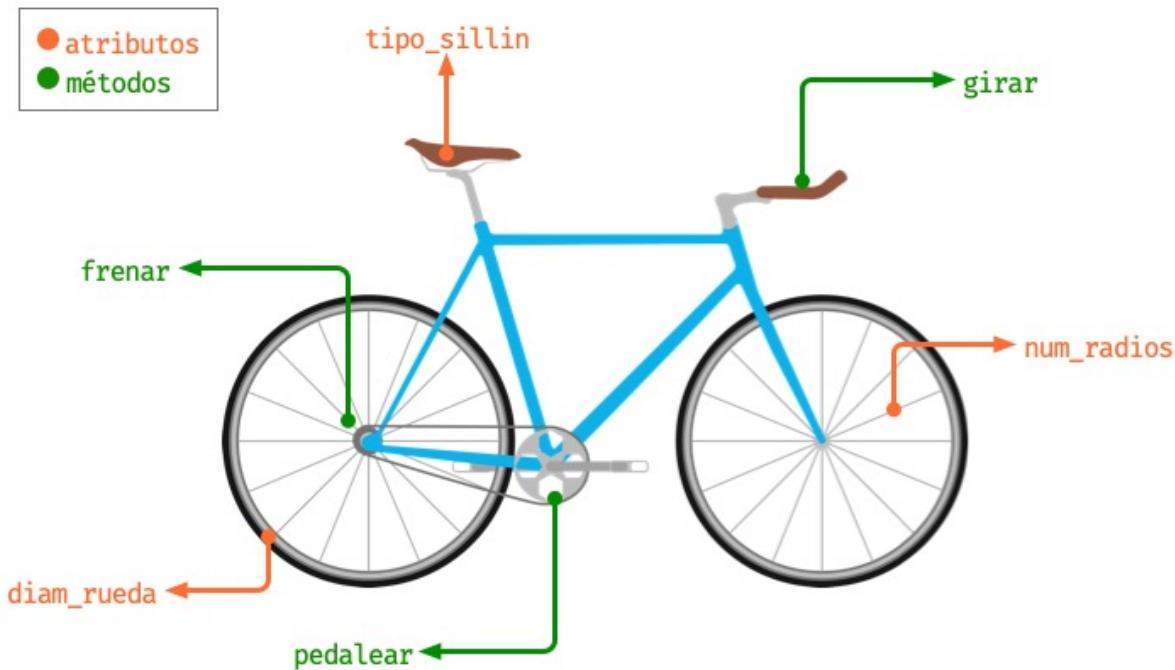


Figura 8: Analogía de atributos y métodos en un objeto «bicicleta»

### ¿Qué es una clase?

Para crear un objeto primero debemos definir la clase que lo contiene. Podemos pensar en la **clase** como el **molde** con el que crear nuevos objetos de ese tipo.

En el **proceso de diseño** de una clase hay que tener en cuenta – entre otros – el **principio de responsabilidad única**<sup>7</sup>, intentando que los atributos y los métodos que contenga estén enfocados a un objetivo único y bien definido.

### 6.2.2 Creando objetos

Empecemos por crear nuestra **primera clase**. En este caso vamos a modelar algunos de los droides de la saga StarWars:

Para ello usaremos la palabra reservada **class** seguido del nombre de la clase:

```
>>> class StarWarsDroid:  
...     pass  
...  
...
```

<sup>7</sup> Principios SOLID

<sup>2</sup> Fuente de la imagen: [Astro Mech Droids](#).

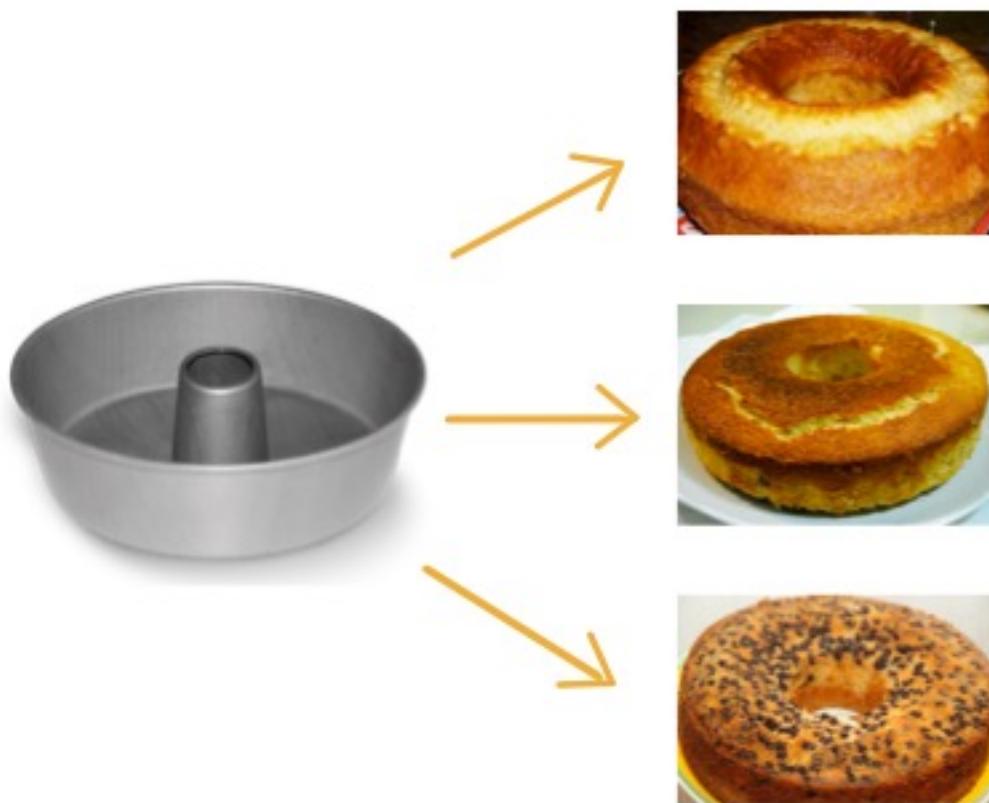


Figura 9: Ejemplificación de creación de objetos a partir de una clase



Figura 10: Droides de la saga StarWars<sup>2</sup>

---

**Consejo:** Los nombres de clases se suelen escribir en formato **CamelCase** y en singular<sup>3</sup>.

---

Existen multitud de droides en el universo StarWars. Una vez que hemos definido la clase genérica podemos crear **instancias/objetos** (droides) concretos:

```
>>> c3po = StarWarsDroid()
>>> r2d2 = StarWarsDroid()
>>> bb8 = StarWarsDroid()

>>> type(c3po)
__main__.StarWarsDroid
>>> type(r2d2)
__main__.StarWarsDroid
>>> type(bb8)
__main__.StarWarsDroid
```

### Añadiendo atributos

Un **atributo** no es más que una variable, un nombre al que asignamos un valor, con la particularidad de vivir dentro de una clase o de un objeto.

Los atributos – por lo general – se suelen asignar durante la creación de un objeto, pero también es posible añadirlos a posteriori:

```
>>> blue_droid = StarWarsDroid()
>>> golden_droid = StarWarsDroid()

>>> golden_droid.name = 'C-3PO'

>>> blue_droid.name = 'R2-D2'
>>> blue_droid.height = 1.09
>>> blue_droid.num_feet = 3
>>> blue_droid.partner_droid = golden_droid # otro droid como atributo
```

Una vez creados, es muy sencillo acceder a los atributos:

```
>>> golden_droid.name
'C-3PO'

>>> blue_droid.num_feet
3
```

---

<sup>3</sup> Guía de estilos [PEP8](#) para convenciones de nombres.

Hemos definido un droide «socio». Veremos a continuación que podemos trabajar con él de una manera totalmente natural:

```
>>> type(blue_droid.partner_droid)
__main__.StarWarsDroid

>>> blue_droid.partner_droid.name # acceso al nombre del droide socio
'C-3PO'

>>> blue_droid.partner_droid.num_feet # aún sin definir!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'StarWarsDroid' object has no attribute 'num_feet'

>>> blue_droid.partner_droid.num_feet = 2
```

## Añadiendo métodos

Un **método** es una función que forma parte de una clase o de un objeto. En su ámbito tiene acceso a otros métodos y atributos de la clase o del objeto al que pertenece.

La definición de un método (de instancia) es análoga a la de una función ordinaria, pero incorporando un primer parámetro `self` que hace referencia a la instancia actual del objeto.

Una de las acciones más sencillas que se pueden hacer sobre un droide es encenderlo o apagarlo. Vamos a implementar estos dos métodos en nuestra clase:

```
>>> class Droid:
...     def switch_on(self):
...         print("Hi! I'm a droid. Can I help you?")
...
...     def switch_off(self):
...         print("Bye! I'm going to sleep")
...

>>> k2so = Droid()

>>> k2so.switch_on()
Hi! I'm a droid. Can I help you?

>>> k2so.switch_off()
Bye! I'm going to sleep
```

## Inicialización

Existe un **método especial** que se ejecuta cuando creamos una instancia de un objeto. Este método es `__init__` y nos permite asignar atributos y realizar operaciones con el objeto en el momento de su creación. También es ampliamente conocido como el **constructor**.

Veamos un ejemplo de este método con nuestros droides en el que únicamente guardaremos el nombre del droide como un atributo del objeto:

```
1 >>> class Droid:  
2 ...     def __init__(self, name):  
3 ...         self.name = name  
4 ...  
5  
6 >>> droid = Droid('BB-8')  
7  
8 >>> droid.name  
9 'BB-8'
```

**Línea 2** Definición del constructor.

**Línea 7** Creación del objeto (y llamada implícita al constructor)

**Línea 9** Acceso al atributo `name` creado previamente en el constructor.

---

## Ejercicio

Escriba una clase `MobilePhone` que represente un teléfono móvil.

Atributos:

- `manufacturer` (cadena de texto)
- `screen_size` (flotante)
- `num_cores` (entero)
- `apps` (lista de cadenas de texto)
- `status` (0: apagado, 1: encendido)

Métodos:

- `__init__(self, manufacturer, screen_size, num_cores)`
- `power_on(self)`
- `power_off(self)`
- `install_app(self, app)`
- `uninstall_app(self, app)`

Crear al menos una instancia (móvil) a partir de la clase creada y «jugar» con los métodos, visualizando cómo cambian sus atributos.

*¿Serías capaz de extender el método `install_app()` para instalar varias aplicaciones a la vez?*

---

### 6.2.3 Atributos

#### Acceso directo

En el siguiente ejemplo vemos que, aunque el atributo `name` se ha creado en el constructor de la clase, también podemos modificarlo desde «fuera» con un acceso directo:

```
>>> class Droid:
...     def __init__(self, name):
...         self.name = name
...
>>> droid = Droid('C-3PO')
>>> droid.name
'C-3PO'
>>> droid.name = 'waka-waka' # esto sería válido!
```

#### Propiedades

Como hemos visto previamente, los atributos definidos en un objeto son accesibles públicamente. Esto puede parecer extraño a personas desarrolladoras de otros lenguajes. En Python existe un cierto «sentido de responsabilidad» a la hora de programar y manejar este tipo de situaciones.

Una posible solución «pitónica» para la privacidad de los atributos es el uso de **propiedades**. La forma más común de aplicar propiedades es mediante el uso de *decoradores*:

- `@property` para leer el valor de un atributo.
- `@name.setter` para escribir el valor de un atributo.

Veamos un ejemplo en el que estamos ofuscando el nombre del droide a través de propiedades:

```
>>> class Droid:
...     def __init__(self, name):
...         self.hidden_name = name
```

(continué en la próxima página)

(provien de la página anterior)

```
...
...     @property
...     def name(self):
...         print('inside the getter')
...         return self.hidden_name
...
...     @name.setter
...     def name(self, name):
...         print('inside the setter')
...         self.hidden_name = name
...
...
>>> droid = Droid('N1-G3L')
>>> droid.name
inside the getter
'N1-G3L'
>>> droid.name = 'Nigel'
inside the setter
>>> droid.name
inside the getter
'Nigel'
```

En cualquier caso, seguimos pudiendo acceder directamente a `.hidden_name`:

```
>>> droid.hidden_name
'Nigel'
```

## Valores calculados

Una propiedad también se puede usar para devolver un **valor calculado** (o computado).

A modo de ejemplo, supongamos que la altura del periscopio de los droides astromecánicos se calcula siempre como un porcentaje de su altura. Veamos cómo implementarlo:

```
>>> class AstromechDroid:
...     def __init__(self, name, height):
...         self.name = name
...         self.height = height
...
...     @property
...     def periscope_height(self):
```

(continué en la próxima página)

(proviene de la página anterior)

```

...
    return 0.3 * self.height
...

>>> droid = AstromechDroid('R2-D2', 1.05)

>>> droid.periscope_height # podemos acceder como atributo
0.315

>>> droid.periscope_height = 10 # no podemos modificarlo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute

```

**Consejo:** La ventaja de usar valores calculados sobre simples atributos es que el cambio de valor en un atributo no asegura que actualicemos otro atributo, y además siempre podremos modificar directamente el valor del atributo, con lo que podríamos obtener efectos colaterales indeseados.

## Ocultando atributos

Python tiene una convención sobre aquellos atributos que queremos hacer «privados» (u ocultos): comenzar el nombre con doble subguion `__`

```

>>> class Droid:
...     def __init__(self, name):
...         self.__name = name
...
>>> droid = Droid('BC-44')

>>> droid.__name # efectivamente no aparece como atributo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Droid' object has no attribute '__name'

```

Lo que realmente ocurre tras el telón se conoce como «*name mangling*» y consiste en modificar el nombre del atributo incorporado la clase como un prefijo. Sabiendo esto podemos acceder al valor del atributo supuestamente privado:

```

>>> droid._Droid__name
'BC-44'

```

## Atributos de clase

Podemos asignar atributos a las clases y serán heredados por todos los objetos instanciados de esa clase.

A modo de ejemplo, en un principio, todos los droides están diseñados para que obedezcan a su dueño. Esto lo conseguiremos a nivel de clase, salvo que ese comportamiento se sobreescriba:

```
>>> class Droid:  
...     obeys_owner = True # obedece a su dueño  
...  
>>> good_droid = Droid()  
>>> good_droid.obeys_owner  
True  
  
>>> t1000 = Droid()  
>>> t1000.obeys_owner = False # T-1000 (Terminator)  
>>> t1000.obeys_owner  
False  
  
>>> Droid.obeys_owner # el cambio no afecta a nivel de clase  
True
```

## 6.2.4 Métodos

### Métodos de instancia

Un **método de instancia** es un método que modifica el comportamiento del objeto al que hace referencia. Recibe `self` como primer parámetro, el cual se convierte en el propio objeto sobre el que estamos trabajando. Python envía este argumento de forma transparente.

Veamos un ejemplo en el que, además del constructor, creamos un método de instancia para desplazar un droide:

```
>>> class Droid:  
...     def __init__(self, name): # método de instancia -> constructor  
...         self.name = name  
...         self.covered_distance = 0  
...  
...     def move_up(self, steps): # método de instancia  
...         self.covered_distance += steps  
...         print(f'Moving {steps} steps')  
...
```

(continué en la próxima página)

(provien de la página anterior)

```
>>> droid = Droid('C1-10P')

>>> droid.move_up(10)
Moving 10 steps
```

## Métodos de clase

Un **método de clase** es un método que modifica el comportamiento de la clase a la que hace referencia. Recibe `cls` como primer parámetro, el cual se convierte en la propia clase sobre la que estamos trabajando. Python envía este argumento de forma transparente. La identificación de estos métodos se completa aplicando el decorador `@classmethod` a la función.

Veamos un ejemplo en el que implementaremos un método de clase que lleva la cuenta de los droides que hemos creado:

```
>>> class Droid:
...     count = 0
...
...     def __init__(self):
...         Droid.count += 1
...
...     @classmethod
...     def total_droids(cls):
...         print(f'{cls.count} droids built so far!')
...
...
>>> droid1 = Droid()
>>> droid2 = Droid()
>>> droid3 = Droid()

>>> Droid.total_droids()
3 droids built so far!
```

## Métodos estáticos

Un **método estático** es un método que no modifica el comportamiento del objeto ni de la clase. No recibe ningún parámetro especial. La identificación de estos métodos se completa aplicando el decorador `@staticmethod` a la función.

Veamos un ejemplo en el que creamos un método estático para devolver las categorías de droides que existen en StarWars:

```
>>> class Droid:  
...     def __init__(self):  
...         pass  
...  
...     @staticmethod  
...     def get_droids_categories():  
...         return ['Messegger', 'Astromech', 'Power', 'Protocol']  
...  
>>> Droid.get_droids_categories()  
['Messegger', 'Astromech', 'Power', 'Protocol']
```

## Métodos mágicos

### Nivel avanzado

Cuando escribimos 'hello world' \* 3 ¿cómo sabe el objeto 'hello world' lo que debe hacer para multiplicarse con el objeto entero 3? O dicho de otra forma, ¿cuál es la implementación del operador \* para «strings» y enteros? En valores numéricos puede parecer evidente (siguiendo los operadores matemáticos), pero no es así para otros objetos. La solución que proporciona Python para estas (y otras) situaciones son los **métodos mágicos**.

Los métodos mágicos empiezan y terminan por doble subguion \_\_ (es por ello que también se les conoce como «dunder-methods»). Uno de los «dunder-methods» más famosos es el constructor de una clase: `__init__()`.

---

**Importante:** Digamos que los métodos mágicos se «disparan» de manera transparente cuando utilizamos ciertas estructuras y expresiones del lenguaje.

---

Para el caso de los operadores, existe un método mágico asociado (que podemos personalizar). Por ejemplo la comparación de dos objetos se realiza con el método `__eq__()`:

Extrapolando esta idea a nuestro universo StarWars, podríamos establecer que dos droides son iguales si su nombre es igual, independientemente de que tengan distintos números de serie:

```
>>> class Droid:  
...     def __init__(self, name, serial_number):  
...         self.serial_number = serial_number  
...         self.name = name  
...  
...     def __eq__(self, droid):  
...         return self.name == droid.name  
...  
...
```

(continuó en la próxima página)

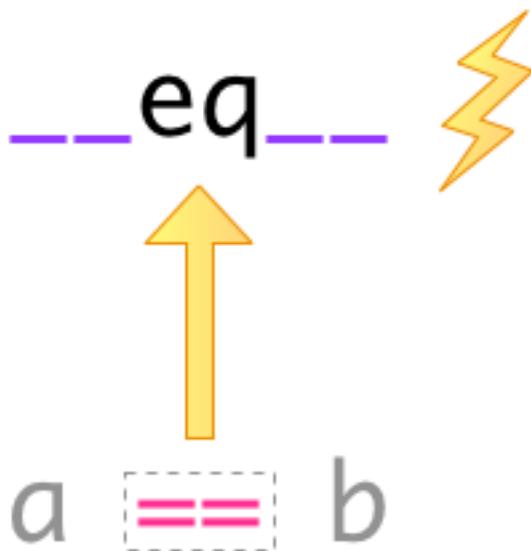


Figura 11: Equivalencia entre operador y método mágico

(proviene de la página anterior)

```
>>> droid1 = Droid('C-3PO', 43974973242)
>>> droid2 = Droid('C-3PO', 85094905984)

>>> droid1 == droid2 # llamada implícita a __eq__
True

>>> droid1.__eq__(droid2)
True
```

**Nota:** Los métodos mágicos no sólo están restringidos a operadores de comparación o matemáticos. Existen muchos otros en la documentación oficial de Python, donde son llamados [métodos especiales](#).

Veamos un ejemplo en el que «sumamos» dos droides. Esto se podría ver como una fusión. Supongamos que la suma de dos droides implica: a) que el nombre del droide resultante es la concatenación de los nombres de los droides; b) que la energía del droide resultante es la suma de la energía de los droides:

```
>>> class Droid:
...     def __init__(self, name, power):
...         self.name = name
...         self.power = power
```

(continué en la próxima página)

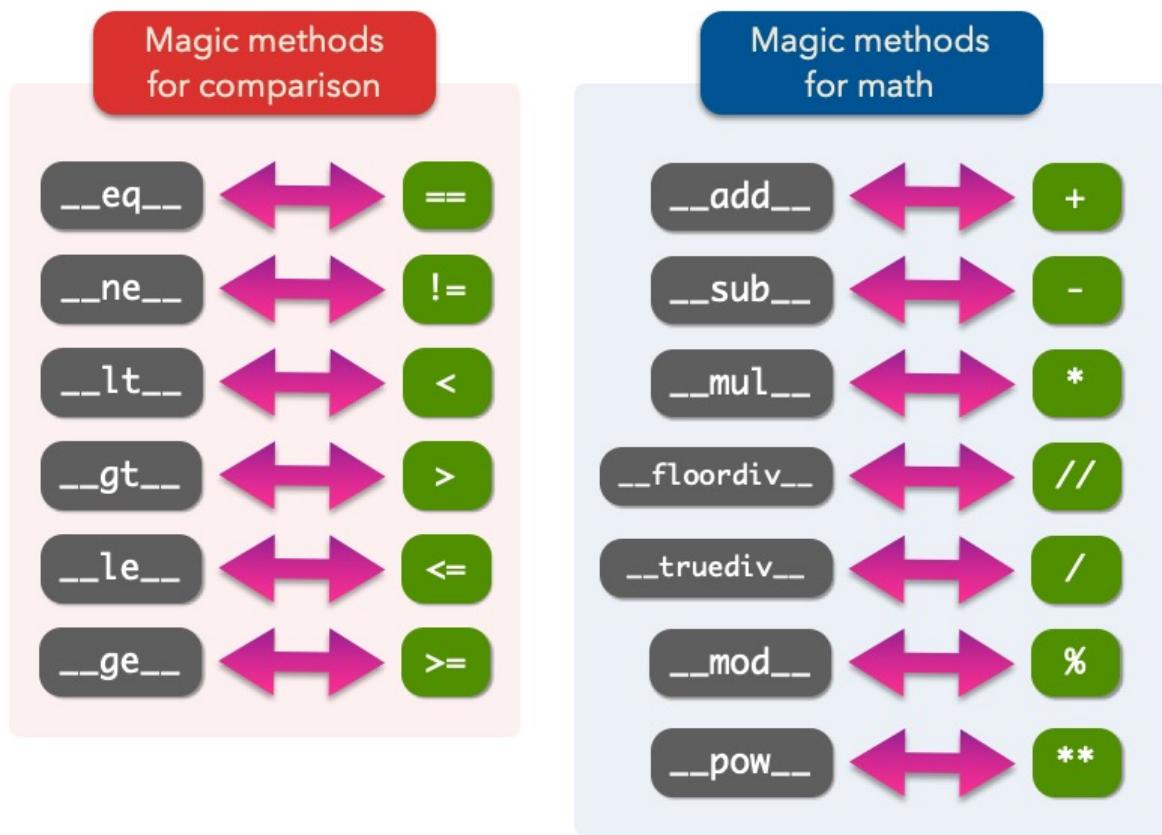


Figura 12: Métodos mágicos para comparaciones y operaciones matemáticas

(provine de la página anterior)

```

...
...     def __add__(self, droid):
...         new_name = self.name + '-' + droid.name
...         new_power = self.power + droid.power
...         return Droid(new_name, new_power) # Hay que devolver un objeto de tipo_
Droid
...

>>> droid1 = Droid('C3PO', 45)
>>> droid2 = Droid('R2D2', 91)

>>> droid3 = droid1 + droid2

>>> print(f'Fusion droid:\n{droid3.name} with power {droid3.power}')
Fusion droid:
C3PO-R2D2 with power 136

```

### \_\_str\_\_

Uno de los métodos mágicos más utilizados es `__str__` que permite establecer la forma en la que un objeto es representado como *cadena de texto*:

```

>>> class Droid:
...     def __init__(self, name, serial_number):
...         self.serial_number = serial_number
...         self.name = name
...
...     def __str__(self):
...         return f'🤖 Droid "{self.name}" serial-no {self.serial_number}'
...

>>> droid = Droid('K-2SO', 8403898409432)

>>> print(droid) # llamada a droid.__str__()
🤖 Droid "K-2SO" serial-no 8403898409432

>>> str(droid)
'🤖 Droid "K-2SO" serial-no 8403898409432'

>>> f'Droid -> {droid}'
'Droid -> 🤖 Droid "K-2SO" serial-no 8403898409432'

```

---

### Ejercicio

Defina una clase `Fraction` que represente una fracción con numerador y denominador enteros y utilice los métodos mágicos para poder sumar, restar, multiplicar y dividir estas fracciones.

Además de esto, necesitaremos:

- `gcd(a, b)` como **método estático** siguiendo el **algoritmo de Euclides** para calcular el máximo común divisor entre `a` y `b`.
- `__init__(self, num, den)` para construir una fracción (incluyendo simplificación de sus términos mediante el método `gcd()`).
- `__str__(self)` para representar una fracción.

Compruebe que se cumplen las siguientes igualdades:

$$\left[ \frac{25}{30} + \frac{40}{45} = \frac{31}{18} \right] \quad \left[ \frac{25}{30} - \frac{40}{45} = \frac{-1}{18} \right] \quad \left[ \frac{25}{30} * \frac{40}{45} = \frac{20}{27} \right] \quad \left[ \frac{25}{30} / \frac{40}{45} = \frac{15}{16} \right]$$

---

## Gestores de contexto

Otra de las aplicaciones de los métodos mágicos (especiales) que puede ser interesante es la de **gestores de contexto**. Se trata de un bloque de código en Python que engloba una serie de acciones a la entrada y a la salida del mismo.

Hay dos métodos que son utilizados para implementar los gestores de contexto:

`__enter__()` Acciones que se llevan a cabo al entrar al contexto.

`__exit__(self, exc_type, exc_value, exc_traceback)` Acciones que se llevan a cabo al salir del contexto.

Veamos un ejemplo en el que implementamos un gestor de contexto que mide tiempos de ejecución:

```
>>> from time import time

>>> class Timer():
...     def __enter__(self):
...         self.start = time()
...
...     def __exit__(self, exc_type, exc_value, exc_traceback):
...         # Omit exception handling
...         self.end = time()
...         exec_time = self.end - self.start
...         print(f'Execution time (seconds): {exec_time:.5f}')
... 
```

Ahora podemos probar nuestro gestor de contexto con un ejemplo concreto. La forma de «activar» el contexto es usar la sentencia `with` seguida del símbolo que lo gestiona:

```

>>> with Timer():
...     for _ in range(1_000_000):
...         x = 2 ** 20
...
Execution time (seconds): 0.05283

>>> with Timer():
...     x = 0
...     for _ in range(1_000_000):
...         x += 2 ** 20
...
Execution time (seconds): 0.08749

```

## 6.2.5 Herencia

### Nivel intermedio

La **herencia** consiste en **crear una nueva clase partiendo de una clase existente**, pero que añade o modifica ciertos aspectos. Se considera una buena práctica tanto para *reutilizar código* como para *realizar generalizaciones*.

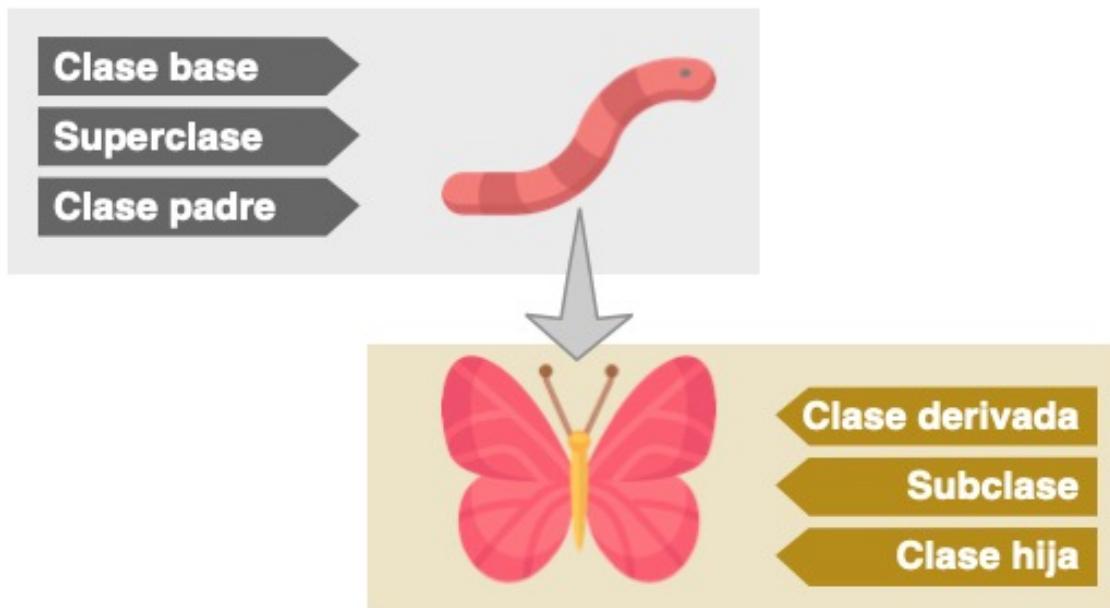


Figura 13: Nomenclatura de clases en la herencia<sup>6</sup>

<sup>6</sup> Iconos por Freepik.

---

**Nota:** Cuando se utiliza herencia, la clase derivada, de forma automática, puede usar todo el código de la clase base sin necesidad de copiar nada explícitamente.

---

## Heredar desde una clase base

Para que una clase «herede» de otra, basta con indicar la clase base entre paréntesis en la definición de la clase derivada.

Sigamos con el ejemplo. Una de las grandes categorías de droides en StarWars es la de [droides de protocolo](#). Vamos a crear una herencia sobre esta idea:

```
>>> class Droid:  
...     ''' Clase Base '''  
...     pass  
...  
  
>>> class ProtocolDroid(Droid):  
...     ''' Clase Derivada '''  
...     pass  
...  
  
>>> issubclass(ProtocolDroid, Droid)  # comprobación de herencia  
True  
  
>>> r2d2 = Droid()  
>>> c3po = ProtocolDroid()
```

Vamos a añadir un par de métodos a la clase base, y analizar su comportamiento:

```
>>> class Droid:  
...     def switch_on(self):  
...         print("Hi! I'm a droid. Can I help you?")  
...  
...     def switch_off(self):  
...         print("Bye! I'm going to sleep")  
...  
  
>>> class ProtocolDroid(Droid):  
...     pass  
...  
  
>>> r2d2 = Droid()  
>>> c3po = ProtocolDroid()
```

(continuó en la próxima página)

(proviene de la página anterior)

```
>>> r2d2.switch_on()
Hi! I'm a droid. Can I help you?

>>> c3po.switch_on() # método heredado de Droid
Hi! I'm a droid. Can I help you?

>>> r2d2.switch_off()
Bye! I'm going to sleep
```

## Sobreescribir un método

Como hemos visto, una clase derivada hereda todo lo que tiene su clase base. Pero en muchas ocasiones nos interesa modificar el comportamiento de esta herencia.

En el ejemplo vamos a modificar el comportamiento del método `switch_on()` para la clase derivada:

```
>>> class Droid:
...     def switch_on(self):
...         print("Hi! I'm a droid. Can I help you?")
...
...     def switch_off(self):
...         print("Bye! I'm going to sleep")
...

>>> class ProtocolDroid(Droid):
...     def switch_on(self):
...         print("Hi! I'm a PROTOCOL droid. Can I help you?")
...
...

>>> r2d2 = Droid()
>>> c3po = ProtocolDroid()

>>> r2d2.switch_on()
Hi! I'm a droid. Can I help you?

>>> c3po.switch_on() # método heredado pero sobreescrito
Hi! I'm a PROTOCOL droid. Can I help you?
```

## Añadir un método

La clase derivada también puede añadir métodos que no estaban presentes en su clase base. En el siguiente ejemplo vamos a añadir un método `translate()` que permita a los *droides de protocolo* traducir cualquier mensaje:

```
>>> class Droid:
...     def switch_on(self):
...         print("Hi! I'm a droid. Can I help you?")
...
...     def switch_off(self):
...         print("Bye! I'm going to sleep")
...
...
>>> class ProtocolDroid(Droid):
...     def switch_on(self):
...         print("Hi! I'm a PROTOCOL droid. Can I help you?")
...
...     def translate(self, msg, from_language):
...         ''' Translate from language to Human understanding '''
...         print(f'{msg} means "ZASCA" in {from_language}')
...
>>> r2d2 = Droid()
>>> c3po = ProtocolDroid()
...
>>> c3po.translate('kiitos', 'Huttese') # idioma de Watto
kiitos means "ZASCA" in Huttese
...
>>> r2d2.translate('kiitos', 'Huttese') # droide genérico no puede traducir
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Droid' object has no attribute 'translate'
```

Con esto ya hemos aportado una personalidad diferente a los droides de protocolo, a pesar de que heredan de la clase genérica de droides de StarWars.

## Accediendo a la clase base

Puede darse la situación en la que tengamos que acceder desde la clase derivada a métodos o atributos de la clase base. Python ofrece `super()` como mecanismo para ello.

Veamos un ejemplo más elaborado con nuestros droides:

```
>>> class Droid:
...     def __init__(self, name):
...         self.name = name
```

(continué en la próxima página)

(provine de la página anterior)

```

...
>>> class ProtocolDroid(Droid):
...     def __init__(self, name, languages):
...         super().__init__(name) # llamada al constructor de la clase base
...         self.languages = languages
...
>>> droid = ProtocolDroid('C-3PO', ['Ewokese', 'Huttese', 'Jawaese'])

>>> droid.name # fijado en el constructor de la clase base
'C-3PO'

>>> droid.languages # fijado en el constructor de la clase derivada
['Ewokese', 'Huttese', 'Jawaese']

```

## Herencia múltiple

### Nivel avanzado

Aunque no está disponible en todos los lenguajes de programación, Python sí permite que los objetos pueden heredar de **múltiples clases base**.

Si en una clase se hace referencia a un método o atributo que no existe, Python lo buscará en todas sus clases base. Es posible que exista una *colisión* en caso de que el método o el atributo buscado esté, a la vez, en varias clases base. En este caso, Python resuelve el conflicto a través del **orden de resolución de métodos**<sup>4</sup>.

Supongamos que queremos modelar la siguiente estructura de clases con *herencia múltiple*:

```

>>> class Droid:
...     def greet(self):
...         return 'Here a droid'
...
>>> class ProtocolDroid(Droid):
...     def greet(self):
...         return 'Here a protocol droid'
...
>>> class AstromechDroid(Droid):
...     def greet(self):

```

(continué en la próxima página)

<sup>4</sup> Viene del inglés «method resolution order» o mro.

<sup>5</sup> Imágenes de los droides por StarWars Fandom.

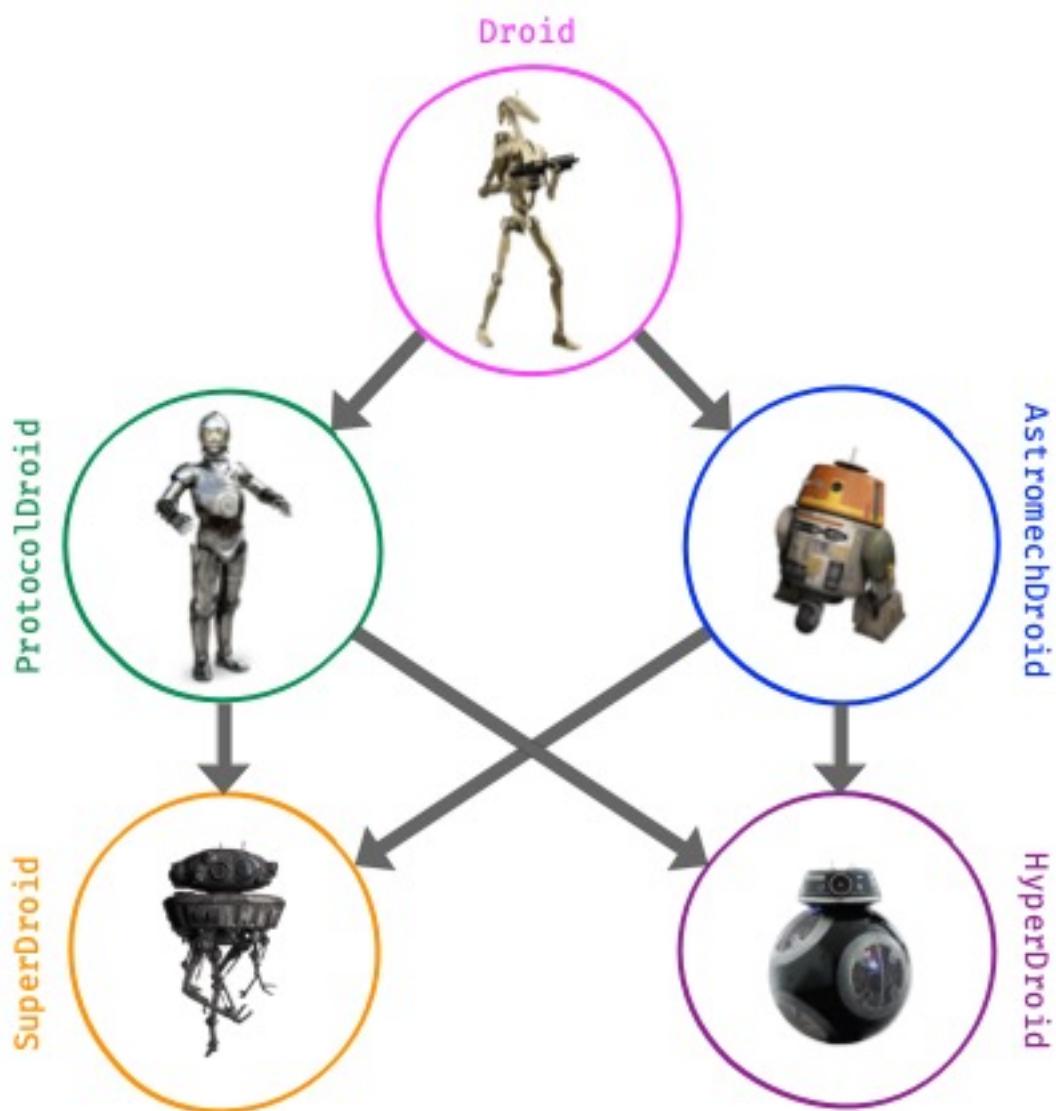


Figura 14: Ejemplo de herencia múltiple<sup>5</sup>

(provine de la página anterior)

```

...
    return 'Here an astromech droid'

...
>>> class SuperDroid(ProtocolDroid, AstromechDroid):
...     pass
...
>>> class HyperDroid(AstromechDroid, ProtocolDroid):
...     pass

```

Todas las clases en Python disponen de un método especial llamado `mro()` que devuelve una lista de las clases que se visitarían en caso de acceder a un método o un atributo. También existe el atributo `__mro__` como una tupla de esas clases:

```

>>> SuperDroid.mro()
[__main__.SuperDroid,
 __main__.ProtocolDroid,
 __main__.AstromechDroid,
 __main__.Droid,
 object]

>>> HyperDroid.__mro__
(__main__.HyperDroid,
 __main__.AstromechDroid,
 __main__.ProtocolDroid,
 __main__.Droid,
 object)

```

Veamos el resultado de la llamada a los métodos definidos:

```

>>> super_droid = SuperDroid()
>>> hyper_droid = HyperDroid()

>>> super_droid.greet()
'Here a protocol droid'

>>> hyper_droid.greet()
'Here an astromech droid'

```

---

**Nota:** Todos los objetos en Python heredan, en primera instancia, de `object`. Esto se puede comprobar con el `mro()` correspondiente:

```

>>> int.mro()
[int, object]

```

(continué en la próxima página)

(provien de la página anterior)

```
>>> str.mro()
[str, object]

>>> float.mro()
[float, object]

>>> tuple.mro()
[tuple, object]

>>> list.mro()
[list, object]

>>> bool.mro()
[bool, int, object]
```

---

## Mixins

Hay situaciones en la que nos interesa incorporar una clase base «independiente» de la jerarquía establecida, y sólo a efectos de **tareas auxiliares**. Esta aproximación podría ayudar a evitar *colisiones* en métodos o atributos reduciendo la ambigüedad que añade la herencia múltiple. Estas clases auxiliares reciben el nombre de «**mixins**».

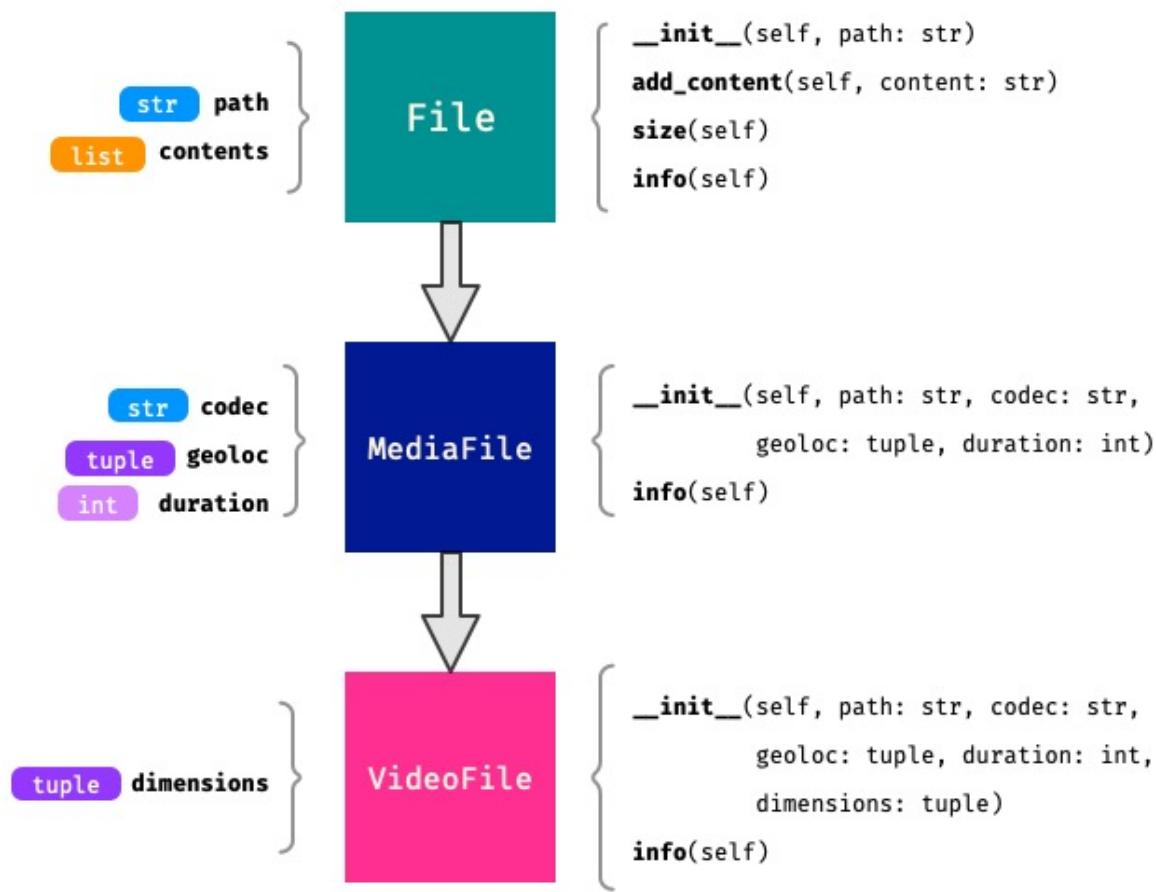
Veamos un ejemplo en el que usamos un «mixin» para mostrar las variables de un objeto:

```
>>> class Instrospection:
...     def dig(self):
...         print(vars(self)) # vars devuelve las variables del argumento
...
...     class Droid(Instrospection):
...         pass
...
...
>>> droid = Droid()
>>> droid.code = 'DN-LD'
>>> droid.num_feet = 2
>>> droid.type = 'Power Droid'
>>> droid.dig()
{'code': 'DN-LD', 'num_feet': 2, 'type': 'Power Droid'}
```

---

## Ejercicio

Dada la siguiente estructura/herencia que representa diferentes clases de ficheros:



Se pide lo siguiente:

1. Cree las **3 clases** de la imagen anterior con la herencia señalada.
2. Cree un objeto de tipo **VideoFile** con las siguientes características:
  - path: /home/python/vanrossum.mp4
  - codec: h264
  - geoloc: (23.5454, 31.4343)
  - duration: 487
  - dimensions: (1920, 1080)
3. Añada el contenido 'audio/ogg' al fichero.
4. Añada el contenido 'video/webm' al fichero.
5. Imprima por pantalla la **info()** de este objeto (el método **info()** debería retornar str y debería hacer uso de los métodos **info()** de las clases base).

Salida esperada:

```
/home/python/vanrossum.mp4 [size=19B]      # self.info() de File
Codec: h264                                # □
Geolocalization: (23.5454, 31.4343)        # □ self.info() de MediaFile
Duration: 487s                               # □
Dimensions: (1920, 1080)                     # self.info() de VideoFile
```

El método `size()` debe devolver el número total de caracteres sumando las longitudes de los elementos del atributo `contents`.

## Agregación y composición

Aunque la herencia de clases nos permite modelar una gran cantidad de casos de uso en términos de «**is-a**» (*es un*), existen muchas otras situaciones en las que la agregación o la composición son una mejor opción. En este caso una clase se compone de otras clases: hablamos de una relación «**has-a**» (*tiene un*).

Hay una sutil diferencia entre agregación y composición:

- La **composición** implica que el objeto utilizado no puede «funcionar» sin la presencia de su propietario.
- La **agregación** implica que el objeto utilizado puede funcionar por sí mismo.

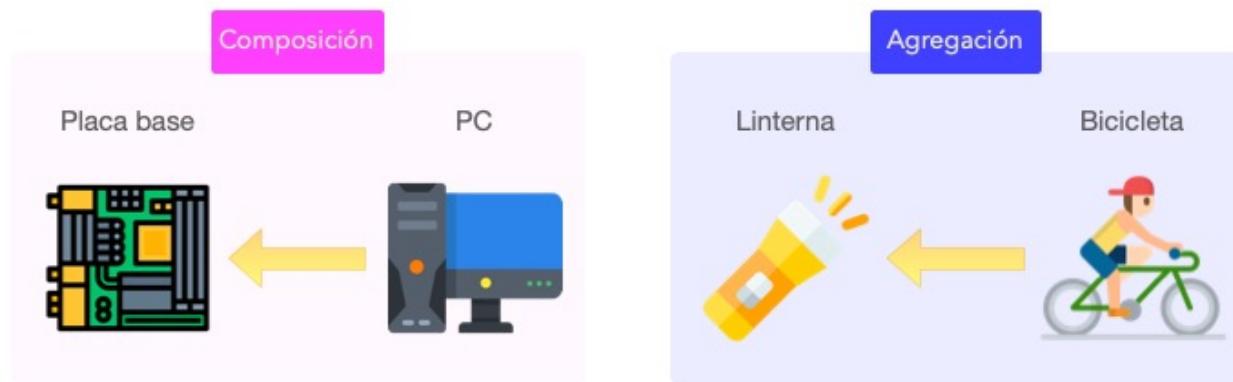


Figura 15: Agregación vs. Composición<sup>6</sup>

Veamos un ejemplo de **agregación** en el que añadimos una herramienta a un droide:

```
>>> class Tool:
...     def __init__(self, name):
...         self.name = name
...
...     def __str__(self):
```

(continué en la próxima página)

(provine de la página anterior)

```
...     return self.name.upper()
...
...
... class Droid:
...     def __init__(self, name, serial_number, tool):
...         self.name = name
...         self.serial_number = serial_number
...         self.tool = tool  # agregación
...
...
...     def __str__(self):
...         return f'Droid {self.name} armed with a {self.tool}'
...
...
>>> lighter = Tool('lighter')
>>> bb8 = Droid('BB-8', 48050989085439, lighter)
>>> print(bb8)
Droid BB-8 armed with a LIGHTER
```

## EJERCICIOS DE REPASO

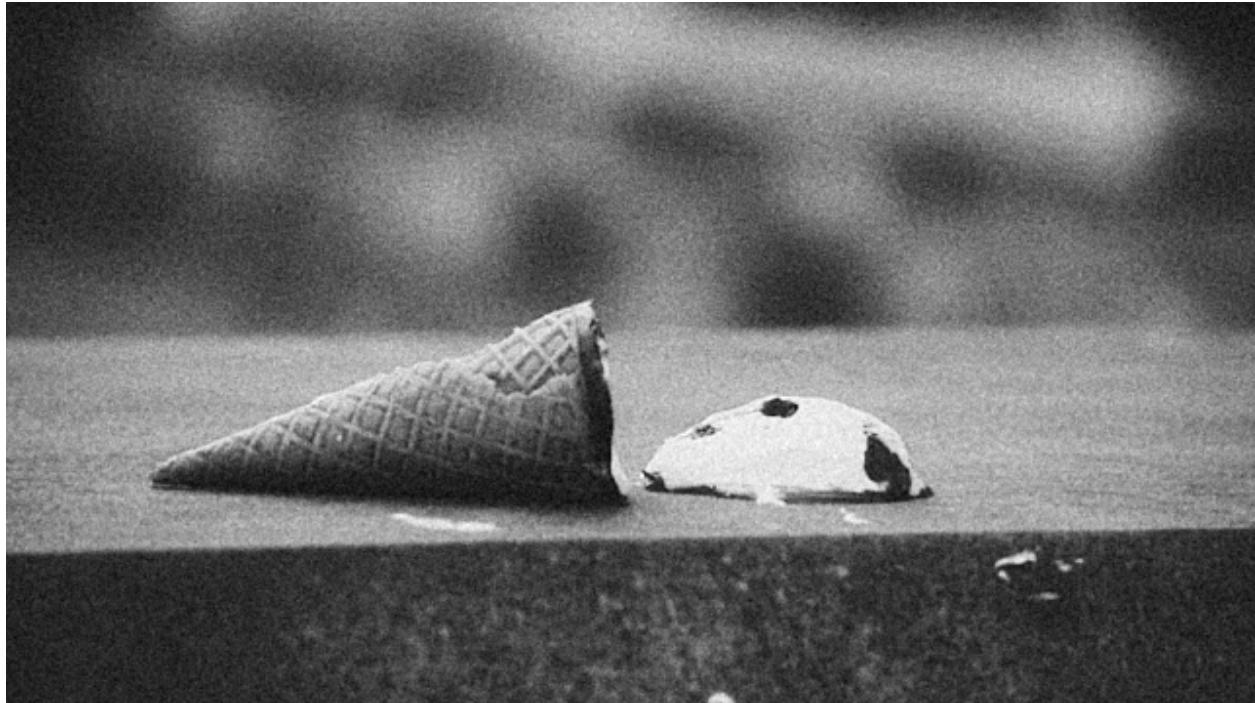
1. Escriba una clase en Python para representar una secuencia de ADN. De momento, la clase sólo contendrá los siguientes elementos:
  - 4 atributos de clase, cada uno representando una base nitrogenada con su valor como un carácter.
  - Constructor que recibe una secuencia de caracteres (bases).
  - Método para representar el objeto en formato «string».
2. Continúe con el ejercicio anterior, y añada a la clase 4 propiedades que calculen el número total de cada una de las bases presentes en la secuencia.
3. Continúe con el ejercicio anterior, y añada a la clase un método de instancia para sumar dos secuencias de ADN. La suma se hará base a base y el resultado será el máximo de cada letra(base).
4. Continúe con el ejercicio anterior, y añada a la clase un método de instancia para obtener el porcentaje de aparición de cada base (usando las propiedades definidas en ejercicios anteriores).
5. Continúe con el ejercicio anterior, y añada a la clase un método de instancia para multiplicar dos secuencias de ADN. La multiplicación consiste en dar como salida una nueva secuencia que contenga sólo aquellas bases que coincidan en posición en ambas secuencias de entrada.

→ Solución a todos los ejercicios

### AMPLIAR CONOCIMIENTOS

- Supercharge Your Classes With Python super()
- Inheritance and Composition: A Python OOP Guide
- OOP Method Types in Python: @classmethod vs @staticmethod vs Instance Methods
- Intro to Object-Oriented Programming (OOP) in Python
- Pythonic OOP String Conversion: \_\_repr\_\_ vs \_\_str\_\_
- @staticmethod vs @classmethod in Python
- Modeling Polymorphism in Django With Python
- Operator and Function Overloading in Custom Python Classes
- Object-Oriented Programming (OOP) in Python 3
- Why Bother Using Property Decorators in Python?

### 6.3 Excepciones



Una **excepción** es el bloque de código que se lanza cuando se produce un **error** en la ejecución de un programa Python.<sup>1</sup>

De hecho ya hemos visto algunas de estas excepciones: accesos fuera de rango a listas o tuplas, accesos a claves inexistentes en diccionarios, etc. Cuando ejecutamos código que podría fallar bajo ciertas circunstancias, necesitamos también manejar, de manera adecuada, las excepciones que se generan.

### 6.3.1 Manejando errores

Si una excepción ocurre en algún lugar de nuestro programa y no es capturada en ese punto, va subiendo (burbujeando) hasta que es capturada en alguna función que ha hecho la llamada. Si en toda la «pila» de llamadas no existe un control de la excepción, Python muestra un mensaje de error con información adicional:

```
>>> def intdiv(a, b):
...     return a // b
...
>>> intdiv(3, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in intdiv
ZeroDivisionError: integer division or modulo by zero
```

Para manejar (capturar) las excepciones podemos usar un bloque de código con las palabras reservadas **try** and **except**:

```
>>> def intdiv(a, b):
...     try:
...         return a // b
...     except:
...         print('Please do not divide by zero...')
...
>>> intdiv(3, 0)
Please do not divide by zero...
```

Aquel código que se encuentre dentro del bloque **try** se ejecutará normalmente siempre y cuando no haya un error. Si se produce una excepción, ésta será capturada por el bloque **except**, ejecutándose el código que contiene.

---

**Consejo:** No es una buena práctica usar un bloque **except** sin indicar el **tipo de excepción** que estamos gestionando, no sólo porque puedan existir varias excepciones que capturar sino

---

<sup>1</sup> Foto original por Sarah Kilian en Unsplash.

porque, como dice el *Zen de Python*: «explícito» es mejor que «implícito».

---

## Especificando excepciones

En el siguiente ejemplo mejoraremos el código anterior, capturando distintos tipos de excepciones:

- `TypeError` por si los operandos no permiten la división.
- `ZeroDivisionError` por si el denominador es cero.
- `Exception` para cualquier otro error que se pueda producir.

Veamos su implementación:

```
>>> def intdiv(a, b):
...     try:
...         result = a // b
...     except TypeError:
...         print('Check operands. Some of them seems strange...')
...     except ZeroDivisionError:
...         print('Please do not divide by zero...')
...     except Exception:
...         print('Ups. Something went wrong...')
...
...
>>> intdiv(3, 0)
Please do not divide by zero...
>>> intdiv(3, '0')
Check operands. Some of them seems strange...
```

## Cubriendo más casos

Python proporciona la cláusula `else` para saber que todo ha ido bien y que no se ha lanzado ninguna excepción. Esto es relevante a la hora de manejar los errores.

De igual modo, tenemos a nuestra disposición la cláusula `finally` que se ejecuta siempre, independientemente de si ha habido o no ha habido error.

Veamos un ejemplo de ambos:

```
>>> values = [4, 2, 7]
>>> user_index = 3
```

(continué en la próxima página)

(provine de la página anterior)

```
>>> try:  
...     r = values[user_index]  
... except IndexError:  
...     print('Error: Index not in list')  
... else:  
...     print(f'Your wishes are my command: {r}')  
... finally:  
...     print('Have a good day!')  
...  
Error: Index not in list  
Have a good day!  
  
>>> user_index = 2  
  
>>> try:  
...     r = values[user_index]  
... except IndexError:  
...     print('Error: Index not in list')  
... else:  
...     print(f'Your wishes are my command: {r}')  
... finally:  
...     print('Have a good day!')  
...  
Your wishes are my command: 7  
Have a good day!
```

---

## Ejercicio

Sabiendo que `ValueError` es la excepción que se lanza cuando no podemos convertir una cadena de texto en su valor numérico, escriba una función `get_int()` que lea un valor entero del usuario y lo devuelva, iterando mientras el valor no sea correcto.

Ejecución a modo de ejemplo:

```
Give me an integer number: ten  
Not a valid integer. Try it again!  
Give me an integer number: diez  
Not a valid integer. Try it again!  
Give me an integer number: 10
```

---

*Trate de implementar tanto la versión recursiva como la versión iterativa.*

---

### 6.3.2 Excepciones propias

## Nivel avanzado

Python ofrece una gran cantidad de excepciones predefinidas. Hasta ahora hemos visto cómo gestionar y manejar este tipo de excepciones. Pero hay ocasiones en las que nos puede interesar crear nuestras propias excepciones. Para ello tendremos que crear una clase *heredando* de `Exception`, la clase base para todas las excepciones.

Veamos un ejemplo en el que creamos una excepción propia controlando que el valor sea un número entero:

```
>>> class NotIntError(Exception):
...     pass
...
...
>>> values = (4, 7, 2.11, 9)

>>> for value in values:
...     if not isinstance(value, int):
...         raise NotIntError(value)
...
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
__main__.NotIntError: 2.11
```

Hemos usado la sentencia `raise` para «elevar» esta excepción, que podría ser controlada en un nivel superior mediante un bloque `try - except`.

**Nota:** Para crear una excepción propia basta con crear una clase vacía. No es necesario incluir código más allá de un `pass`.

## Mensaje personalizado

Podemos personalizar la excepción añadiendo un mensaje más informativo. Siguiendo el ejemplo anterior, veamos cómo introducimos esta información:

```
>>> class NotIntError(Exception):
...     def __init__(self, message='This module only works with integers. Sorry!'):
...         super().__init__(message)
...
>>> raise NotIntError()
Traceback (most recent call last):
```

(continué en la próxima página)

(proviene de la página anterior)

```
File "<stdin>", line 1, in <module>
__main__.NotIntError: This module only works with integers. Sorry!
```

Podemos ir un paso más allá e incorporar en el mensaje el propio valor que está generando el error:

```
>>> class NotIntError(Exception):
...     def __init__(self, value, message='This module only works with integers. ↴Sorry!'):
...         self.value = value
...         self.message = message
...         super().__init__(self.message)
...
...     def __str__(self):
...         return f'{self.value} -> {self.message}'
...
...
>>> raise NotIntError(2.11)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.NotIntError: 2.11 -> This module only works with integers. Sorry!
```

## AMPLIAR CONOCIMIENTOS

- Python Exceptions: An introduction
- Python KeyError Exceptions and How to Handle Them
- Understanding the Python Traceback

## 6.4 Módulos



Escribir pequeños trozos de código puede resultar interesante para realizar determinadas pruebas. Pero a la larga, nuestros programas tenderán a crecer y será necesario agrupar el código en unidades manejables.

Los **módulos** son simplemente ficheros de texto que contienen código Python y representan unidades con las que *evitar la repetición y favorecer la reutilización*.<sup>1</sup>

### 6.4.1 Importar un módulo

Para hacer uso del código de otros módulos usaremos la sentencia `import`. Esto permite importar el código y las variables de dicho módulo para que estén disponibles en nuestro programa.

La forma más sencilla de importar un módulo es `import <module>` donde `module` es el nombre de otro fichero Python, sin la extensión `.py`.

Supongamos que partimos del siguiente fichero (*módulo*):

`arith.py`

```
1 def addere(a, b):
2     '''Sum of input values'''
```

(continué en la próxima página)

<sup>1</sup> Foto original por Xavi Cabrera en Unsplash.

(proviene de la página anterior)

```

3   return a + b
4
5
6 def minuas(a, b):
7     '''Subtract of input values'''
8     return a - b
9
10
11 def pullulate(a, b):
12     '''Product of input values'''
13     return a * b
14
15
16 def partitus(a, b):
17     '''Division of input values'''
18     return a / b

```

Desde otro fichero - en principio en la misma carpeta - podríamos hacer uso de las funciones definidas en `arith.py`.

## Importar módulo completo

Desde otro fichero haríamos lo siguiente para importar todo el contenido del módulo `arith.py`:

```

1 >>> import arith
2
3 >>> arith.addere(3, 7)
4 10

```

---

**Nota:** Nótese que en la **Línea 3** debemos anteponer a la función `addere()` el *espacio de nombres* que define el módulo `arith`.

---

## Ruta de búsqueda de módulos

Python tiene 2 formas de encontrar un módulo:

1. En la carpeta actual de trabajo.
2. En las rutas definidas en la variable de entorno `PYTHONPATH`.

Para ver las rutas de búsqueda establecidas, podemos ejecutar lo siguiente en un intérprete de Python:

```
>>> import sys  
  
>>> sys.path  
['/path/to/.pyenv/versions/3.9.1/envs/aprendepython/bin',  
 '/path/to/.pyenv/versions/3.9.1/lib/python3.9',  
 '/path/to/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-packages',  
 '']
```

La cadena vacía que existe al final de la lista hace referencia a la **carpeta actual**.

## Modificando la ruta de búsqueda

Si queremos modificar la ruta de búsqueda, existen dos opciones:

**Modificando directamente la variable PYTHONPATH** Para ello exportamos dicha variable de entorno desde una terminal:

```
$ export PYTHONPATH=/tmp
```

Y comprobamos que se ha modificado en `sys.path`:

```
>>> sys.path  
['/path/to/.pyenv/versions/3.9.1/envs/aprendepython/bin',  
 '/tmp',  
 '/path/to/.pyenv/versions/3.9.1/lib/python3.9',  
 '/path/to/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-packages',  
 '']
```

**Modificando directamente la lista `sys.path`** Para ello accedemos a la lista que está en el módulo `sys` de la librería estandar:

```
>>> sys.path.append('/tmp') # añadimos al final  
  
>>> sys.path  
['/path/to/.pyenv/versions/3.9.1/envs/aprendepython/bin',  
 '/path/to/.pyenv/versions/3.9.1/lib/python3.9',  
 '/path/to/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-packages',  
 '',  
 '/tmp']
```

```
>>> sys.path.insert(0, '/tmp') # insertamos por el principio  
  
>>> sys.path  
['/tmp',  
 '/path/to/.pyenv/versions/3.9.1/envs/aprendepython/bin',
```

(continué en la próxima página)

(provine de la página anterior)

```
'/path/to/.pyenv/versions/3.9.1/lib/python3.9',
'/path/to/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-packages',
'']
```

---

**Truco:** El hecho de poner nuestra ruta al principio o al final de `sys.path` influye en la búsqueda, ya que si existen dos (o más módulos) que se llaman igual en nuestra ruta de búsqueda, Python usará el primero que encuentre.

---

## Importar partes de un módulo

Es posible que no necesitemos todo aquello que está definido en `arith.py`. Supongamos que sólo vamos a realizar divisiones. Para ello haremos lo siguiente:

```
1 >>> from arith import partitus
2
3 >>> partitus(5, 2)
4 2.5
```

---

**Nota:** Nótese que en la **Línea 3** ya podemos hacer uso directamente de la función `partitus()` porque la hemos importado directamente. Este esquema tiene el inconveniente de la posible **colisión de nombres**, en aquellos casos en los que tuviéramos algún objeto con el mismo nombre que el objeto que estamos importando.

---

## Importar usando un alias

Hay ocasiones en las que interesa, por colisión de otros nombres o por mejorar la legibilidad, usar un nombre diferente del módulo (u objeto) que estamos importando. Python nos ofrece esta posibilidad a través de la sentencia `as`.

Supongamos que queremos importar la función del ejemplo anterior pero con otro nombre:

```
>>> from arith import partitus as mydivision
>>> mydivision(5, 2)
2.5
```

### 6.4.2 Paquetes

Un **paquete** es simplemente una **carpeta** que contiene ficheros .py. Además permite tener una jerarquía con más de un nivel de subcarpetas anidadas.

Para ejemplificar este modelo vamos a crear un paquete llamado `mymath` que contendrá 2 módulos:

- `arith.py` para operaciones aritméticas (ya visto *anteriormente*).
- `logic.py` para operaciones lógicas.

El código del módulo de operaciones lógicas es el siguiente:

`logic.py`

```
1 def et(a, b):
2     '''Logic "and" of input values'''
3     return a & b
4
5
6 def uel(a, b):
7     '''Logic "or" of input values'''
8     return a | b
9
10
11 def vel(a, b):
12     '''Logic "xor" of input values'''
13     return a ^ b
```

Si nuestro código principal va a estar en un fichero `main.py` (*a primer nivel*), la estructura de ficheros nos quedaría tal que así:

```
1 .
2   main.py
3   mymath
4     arith.py
5     logic.py
6
7 1 directory, 3 files
```

**Línea 2** Punto de entrada de nuestro programa a partir del fichero `main.py`

**Línea 3** Carpeta que define el paquete `mymath`.

**Línea 4** Módulo para operaciones aritméticas.

**Línea 5** Módulo para operaciones lógicas.

## Importar desde un paquete

Si ya estamos en el fichero `main.py` (o a ese nivel) podremos hacer uso de nuestro paquete de la siguiente forma:

```

1 >>> from mymath import arith, logic
2
3 >>> arith.pullulate(4, 7)
4 28
5
6 >>> logic.et(1, 0)
7 0

```

**Línea 1** Importar los módulos `arith` y `logic` del paquete `mymath`

**Línea 3** Uso de la función `pullulate` que está definida en el módulo `arith`

**Línea 5** Uso de la función `et` que está definida en el módulo `logic`

### 6.4.3 Programa principal

Cuando decidimos desarrollar una pieza de software en Python, normalmente usamos distintos ficheros para ello. Algunos de esos ficheros se convertirán en *módulos*, otros se englobarán en *paquetes* y existirá uno en concreto que será nuestro **punto de entrada**, también llamado **programa principal**.

---

**Consejo:** Suele ser una buena práctica llamar `main.py` al fichero que contiene nuestro programa principal.

---

La estructura que suele tener este *programa principal* es la siguiente:

```

# imports de la librería estándar
# imports de librerías de terceros
# imports de módulos propios

# CÓDIGO PROPIO
# ...
# CÓDIGO PROPIO

if __name__ == '__main__':
    # punto de entrada real

```

---

**Importante:** Si queremos ejecutar este fichero `main.py` desde línea de comandos, tendríamos que hacer:

```
$ python3 main.py
```

```
if __name__ == '__main__'
```

Esta condición permite, en el programa principal, diferenciar qué código se lanzará cuando el fichero se ejecuta directamente o cuando el fichero se importa desde otro lugar.

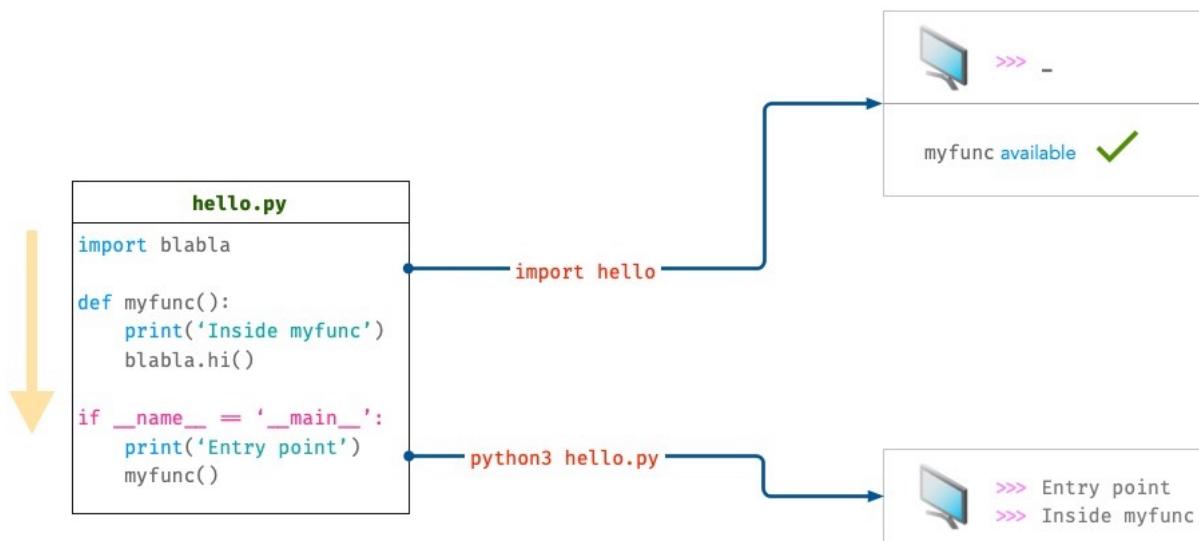


Figura 16: Comportamiento de un programa principal al importarlo o ejecutarlo

`hello.py`

```
1 import blabla
2
3
4 def myfunc():
5     print('Inside myfunc')
6     blabla.hi()
7
8
9 if __name__ == '__main__':
10    print('Entry point')
11    myfunc()
```

**import hello** El código se ejecuta siempre desde la primera instrucción a la última:

- Línea 1: se importa el módulo `blabla`.
- Línea 4: se define la función `myfunc()` y estará disponible para usarse.

- **Línea 9:** esta condición **no** se cumple, ya que estamos importando y la variable especial `__name__` no toma ese valor. Con lo cual finaliza la ejecución.
- *No hay salida por pantalla.*

\$ **python3 hello.py** El código se ejecuta siempre desde la primera instrucción a la última:

- **Línea 1:** se importa el módulo `blabla`.
- **Línea 4:** se define la función `myfunc()` y estará disponible para usarse.
- **Línea 9:** esta condición **sí** se cumple, ya que estamos ejecutando directamente el fichero (*como programa principal*) y la variable especial `__name__` toma el valor `__main__`.
- **Línea 10:** salida por pantalla de la cadena de texto `Entry point`.
- **Línea 11:** llamada a la función `myfunc()` que muestra por pantalla `Inside myfunc`, además de invocar a la función `hi()` del módulo `blabla`.

## AMPLIAR CONOCIMIENTOS

- Defining Main Functions in Python
- Python Modules and Packages: An Introduction
- Absolute vs Relative Imports in Python
- Running Python Scripts
- Writing Beautiful Pythonic Code With PEP 8
- Python Imports 101



# CAPÍTULO 7

---

## Procesamiento de texto

---

Además de las herramientas que se han visto en *cadenas de texto*, la librería estándar nos ofrece una serie de módulos para procesamiento de texto que nos harán la vida más fácil a la hora de gestionar este tipo de datos.

## 7.1 string



El módulo `string` proporciona una serie de constantes muy útiles para manejo de «strings», además de distintas estrategias de formateado de cadenas.<sup>1</sup>

### 7.1.1 Constantes

Las constantes definidas en este módulo son las siguientes:

```
>>> import string

>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'

>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

>>> string.digits
'0123456789'
```

(continuó en la próxima página)

---

<sup>1</sup> Foto original de portada por Steve Johnson en Unsplash.

(proviene de la página anterior)

```
>>> string.hexdigits
'0123456789abcdefABCDEF'

>>> string.octdigits
'01234567'

>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\"]^_`{|}~'

>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?
@[\\"]^_`{|}~ \t\n\r\x0b\x0c'

>>> string.whitespace
' \t\n\r\x0b\x0c'
```

## Ejercicio

Dada una cadena de texto, compruebe si todos sus caracteres son dígitos ASCII. *Ignore los espacios en blanco.*

### Ejemplo

- Entrada: This is it
- Salida: True

## 7.1.2 Plantillas

El módulo `string` también nos permite usar plantillas con interpolación de variables. Algo similar a los *f-strings* pero con otro tipo de sintaxis.

Lo primero es definir la plantilla. Las variables que queramos interporlar deben ir precedidas del signo dólar \$:

```
from string import Template

tmpl = Template('$lang is the best programming language in the $place!')
```

Ahora podemos realizar la sustitución con los valores que nos interesen:

```
>>> tmpl.substitute(lang='Python', place='World')
'Python is the best programming language in the World!'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> tmpl.substitute({'lang': 'Python', 'place': 'World'})  
'Python is the best programming language in the World!'
```

Hay que prestar atención cuando el identificador de variable está seguido por algún carácter que, a su vez, puede formar parte del identificador. En este caso hay que utilizar llaves para evitar la ambigüedad:

```
>>> tmpl = Template('You won several ${price}s')  
  
>>> tmpl.substitute(price='phone')  
'You won several phones'
```

### Sustitución segura

En el caso de que alguna de las variables que estamos interpolando no exista o no tenga ningún valor, obtendremos un error al sustituir:

```
>>> tmpl = Template('$lang is the best programming language in the $place!')  
  
>>> tmpl.substitute(lang='Python')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'place'
```

Para ello Python nos ofrece el método `safe_substitute()` que no emite error si alguna variable no es especificada:

```
>>> tmpl.safe_substitute(lang='Python')  
'Python is the best programming language in the $place!'
```

### Casos de uso

A primera vista podría parecer que este sistema de plantillas no aporta gran ventaja sobre los *f-strings* que ya hemos visto. Sin embargo hay ocasiones en los que puede resultar muy útil.

La mayoría de estas situaciones tienen que ver con **la oportunidad** de definir el «string». Si en el momento de crear la plantilla aún no están disponibles las variables de sustitución, podría interesar utilizar la estrategia que nos proporciona este módulo.

Supongamos un ejemplo en el que tenemos una estructura de «url» y queremos únicamente sustituir una parte de ella. Para no tener que repetir la cadena de texto completa en un «f-string», podríamos seguir este enfoque:

```
>>> urlbase = Template('https://python.org/3/library/$module.html')

>>> for module in ('string', 're', 'difflib'):
...     url = urlbase.substitute(module=module)
...     print(url)
...
https://python.org/3/library/string.html
https://python.org/3/library/re.html
https://python.org/3/library/difflib.html
```



# CAPÍTULO 8

---

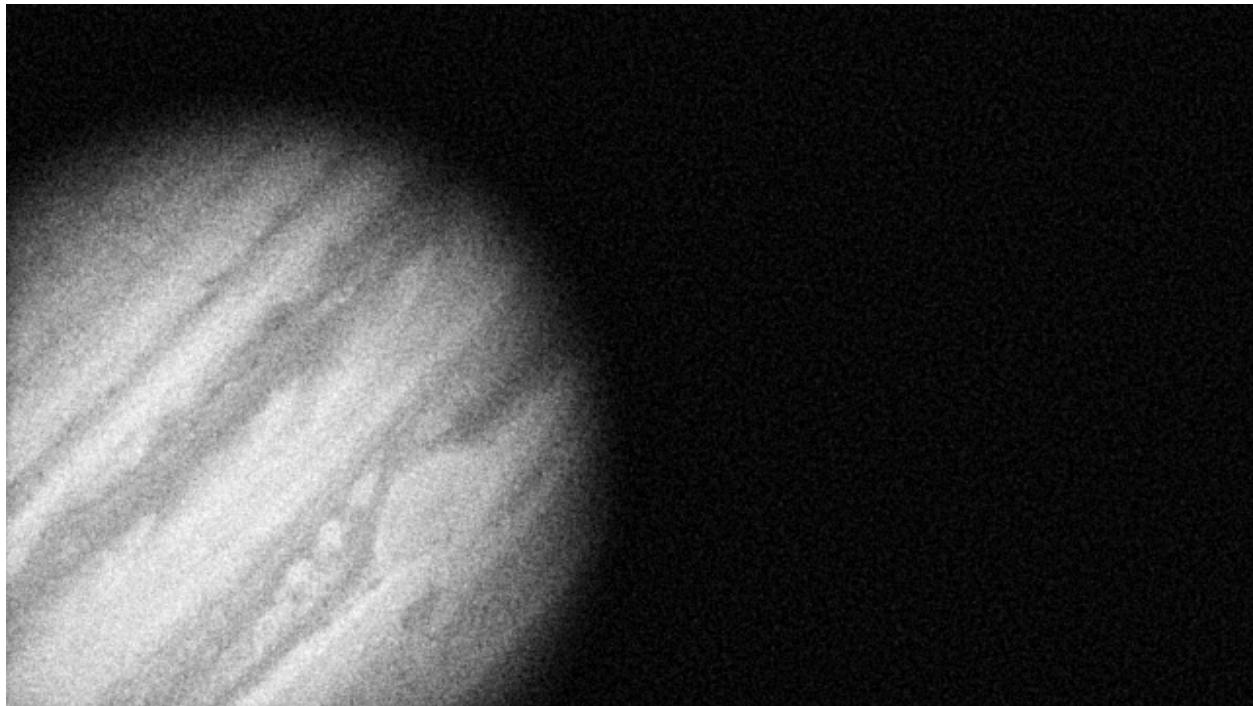
## Ciencia de datos

---

La **Ciencia de Datos** representa uno de los ámbitos de aplicación más importantes dentro del mundo Python. De hecho, en la encuesta a desarrolladores/as de JetBrains del año 2020 se puede observar que **análisis de datos y aprendizaje automático** están en primera y tercera opción en la pregunta *¿para qué utiliza Python?*.

Muchos de los paquetes que veremos en esta sección también vienen incluidos por defecto en *Anaconda* una plataforma para desarrollo de ciencia de datos que dispone de instaladores para todos los sistemas operativos.

## 8.1 jupyter



El módulo `jupyter` proporciona un entorno de desarrollo integrado para ciencia de datos, que no es exclusivo de Python, sino que además admite otros lenguajes en su «backend».<sup>1</sup>

```
$ pip install jupyter
```

Para lanzar el servidor de «notebooks»<sup>2</sup>:

```
$ jupyter notebook
```

---

**Nota:** Este comando nos debería abrir una ventana en el navegador web por defecto del sistema, apuntando a la dirección <http://localhost:8888>

---

<sup>1</sup> Foto original de portada por [NASA](#) en Unsplash.

<sup>2</sup> Un «notebook» es el concepto de cuaderno (documento) científico que se maneja en Jupyter

## 8.1.1 Notebooks

Un «notebook» es un documento que está compuesto por **celdas** en las que podemos incluir:

- Texto en formato **markdown** (incluyendo fórmulas).
- Elementos multimedia.
- Código Python *ejecutable*.

Los «notebooks» son archivos con extensión **.ipynb** (proviene de «IPython Notebook»).

## 8.1.2 Interfaz

Jupyter se presenta como una aplicación web en cuya interfaz podemos encontrar distintos elementos que nos permitirán desarrollar nuestras tareas de programación de una forma más cómoda.

### Explorador de archivos

Lo primero que veremos al arrancar el servidor de «notebooks» será el **explorador de archivos** con un diseño muy similar al de cualquier sistema operativo.

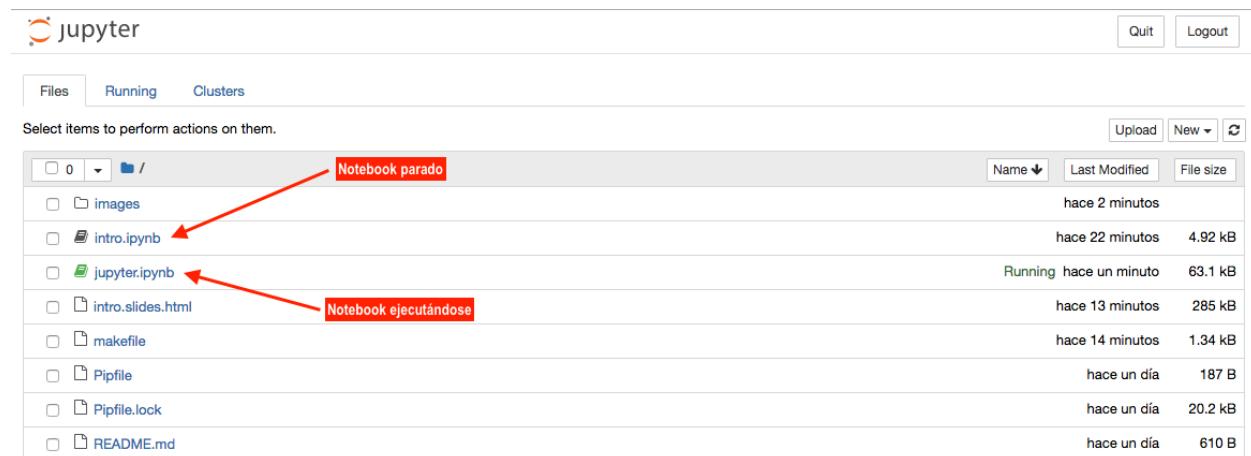


Figura 1: Explorador de archivos de Jupyter Notebook

**Nota:** Los «notebooks» que se están ejecutando suelen tener un color verde en el ícono, mientras que los que están parados aparecen en gris.

## Barra de menú

### Menú Fichero

Del estilo de los menús tradicionales de aplicaciones, aquí podemos encontrar las principales funciones sobre ficheros.

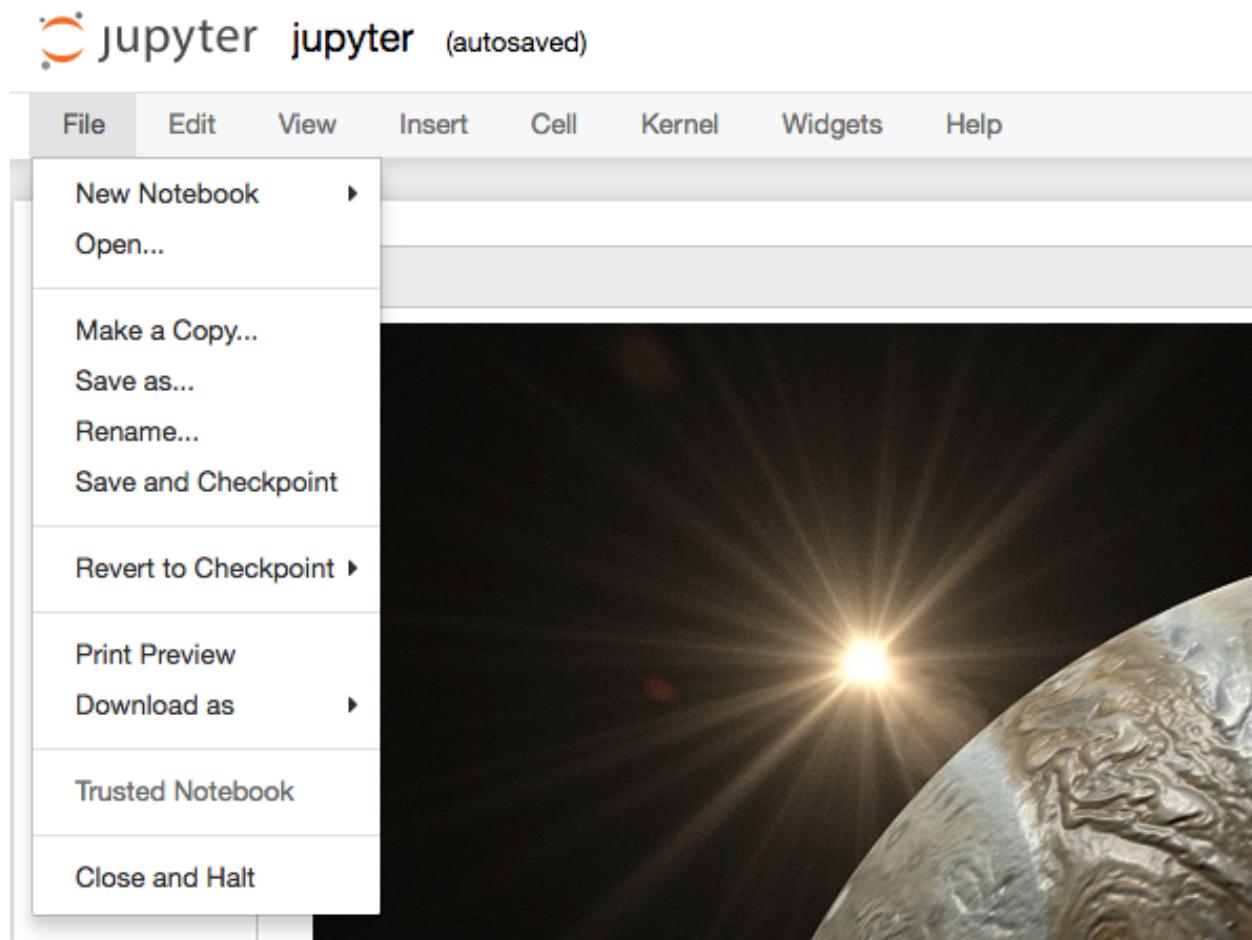


Figura 2: Menú Fichero de Jupyter Notebook

**Checkpoints:** Permiten guardar el estado del «notebook» en un momento determinado para luego poder revertirlo a ese momento del tiempo.

**Exportar notebooks:** Es posible exportar «notebooks» a una gran variedad de formatos:

- Python (.py)
- HTML (.html)
- Reveal.js «slides» (.html)
- Markdown (.md)

- reST (.rst)
- PDF vía LaTeX (.pdf)
- asciidoc (.asciidoc)
- custom (.txt)
- LaTeX (.tex)

---

## Ejercicio

Cree un «notebook» de prueba y descárgelo en formato **HTML** y **Markdown**.

---

## Menú Edición

Este menú contiene las acciones que podemos realizar sobre una o varias celdas.

Las funciones las podríamos agrupar en **gestión de celdas** (cortar, pegar, borrar, dividir, unir, mover, etc.) e **inserción de imágenes** seleccionando desde un cuadro de diálogo.

## Menú Vista

Permite modificar el aspecto visual de determinados elementos de la aplicación.

**Números de línea:** Puede resultar interesante mostrar los números de línea en celdas que contengan código.

**Modo presentación (Cell Toolbar ➔ Slideshow)** : Jupyter Notebook ofrece la posibilidad de crear una presentación sobre el documento en el que estamos trabajando. Cada celda se puede configurar con alguno de los siguientes tipos:

- Slide.
- Subslide.
- Fragment.
- Skip.
- Notes.

**Etiquetas (Cell Toolbar ➔ Tags):** Es interesante – entre otras – el uso de la etiqueta `raises-exception` ya que nos permite ejecutar todas las celdas de un «notebook» sin que el sistema se detenga por errores en la celda etiquetada, ya que estamos informando que lanzará una *excepción*.

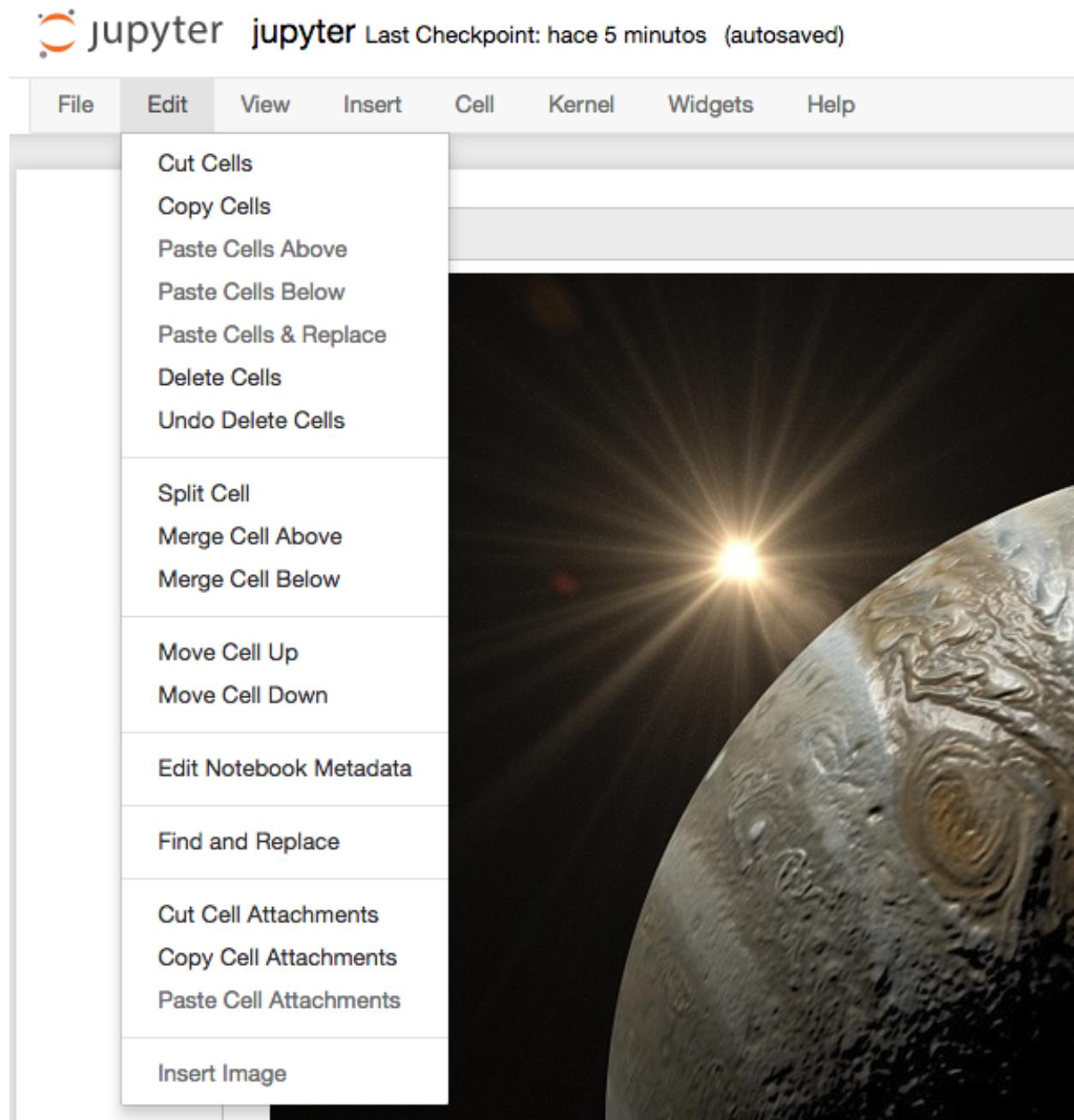


Figura 3: Menú Edición de Jupyter Notebook

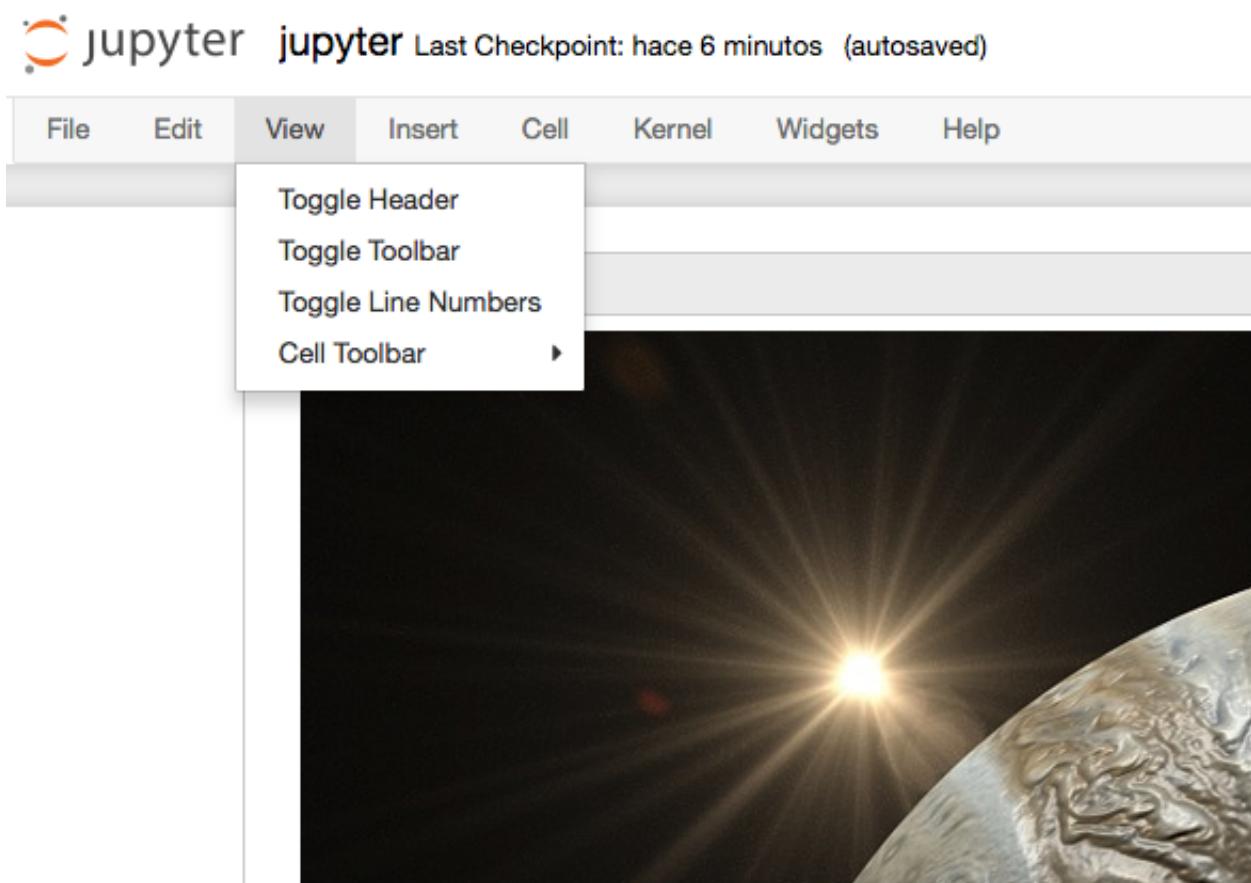


Figura 4: Menú Vista de Jupyter Notebook

## Menú Insertar

Insertar celda antes o después de la actual.

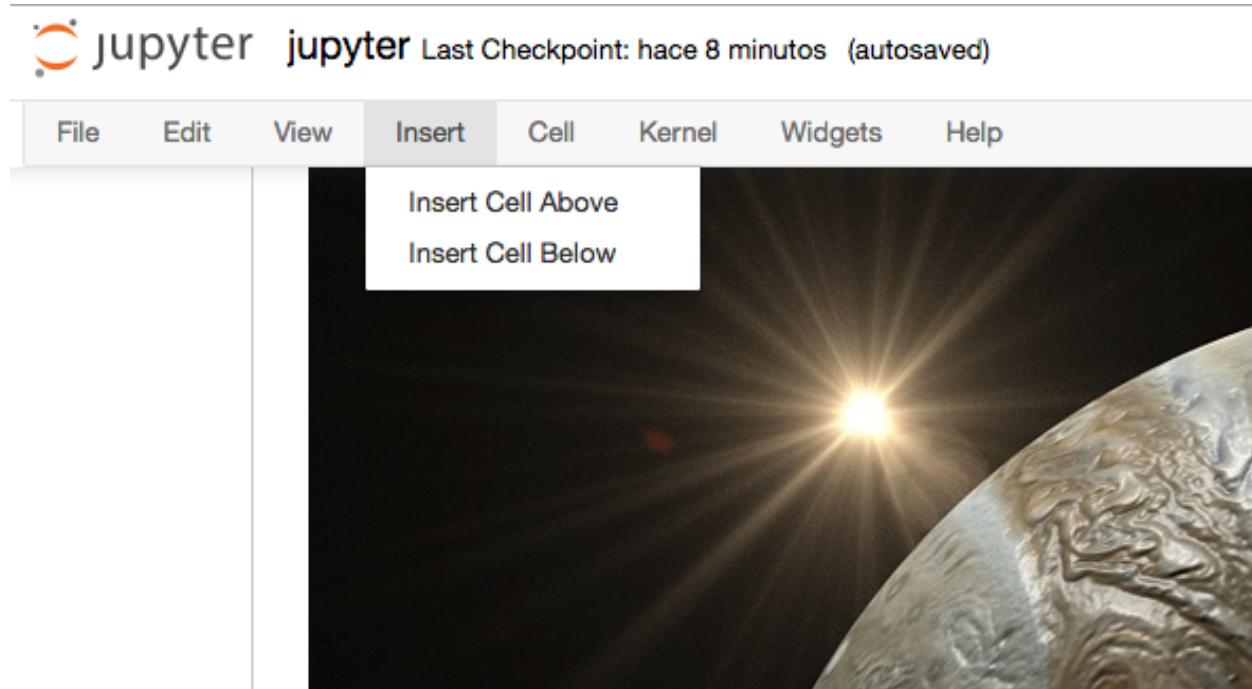


Figura 5: Menú Insertar de Jupyter Notebook

## Menú Celda

Principalmente enfocado a la ejecución de las celdas que componen el «notebook».

**Ejecución de celdas:** La ejecución de celdas se puede hacer de forma individual o grupal así como indicando el punto de partida (celda actual).

### Tipo de celdas:

- **Code:** para incluir código (se podrá ejecutar el lenguaje de programación según el «kernel» instalado).
- **Markdown:** para escribir texto utilizando sintaxis markdown.
- **Raw:** estas celdas no serán formateadas.

**Salida de celdas:** La ejecución de las celdas de código tiene (suele tener) una salida. Esta salida se puede ocultar (si interesa). Incluso tenemos control sobre activar o desactivar el «scroll» en caso de que la salida sea muy larga.

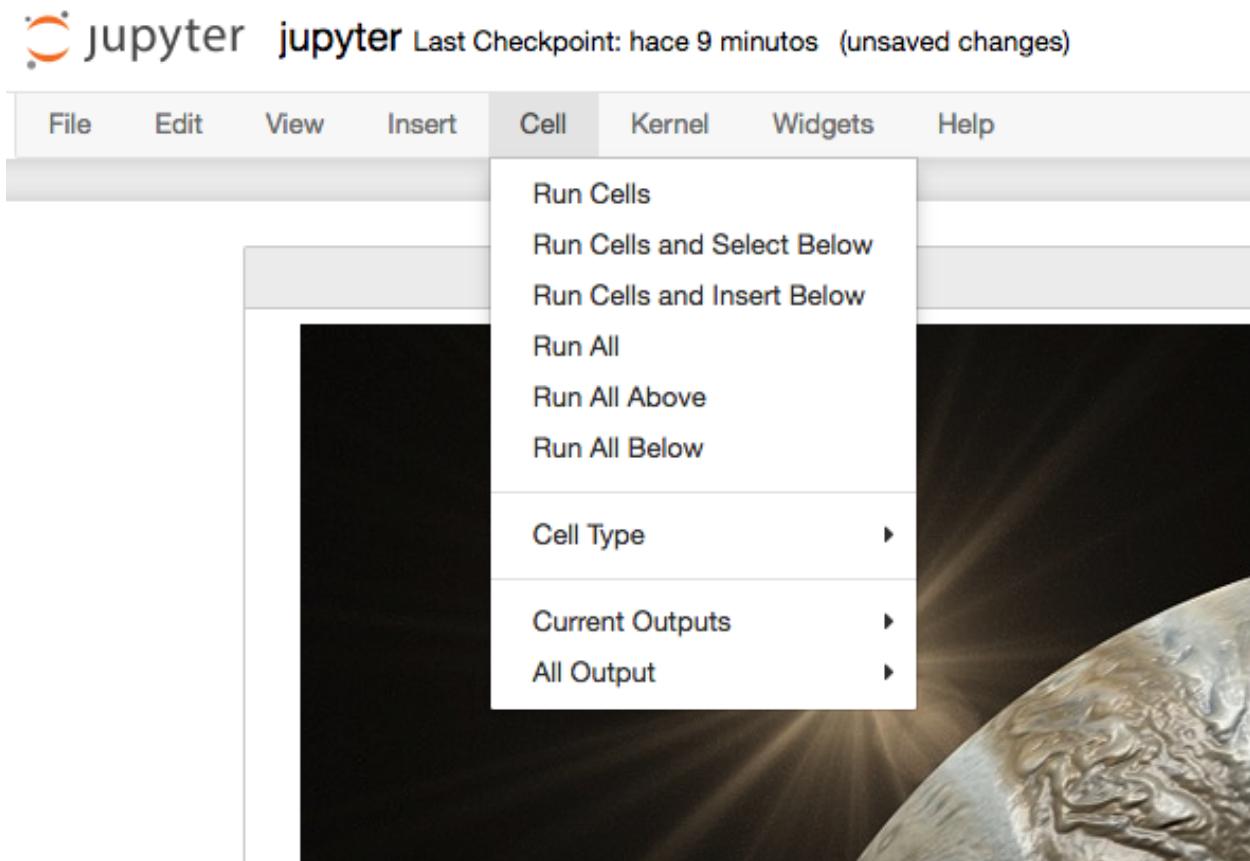


Figura 6: Menú Celda de Jupyter Notebook

## Menú Kernel

Permite gestionar el servicio que se encarga de lanzar los «notebooks».

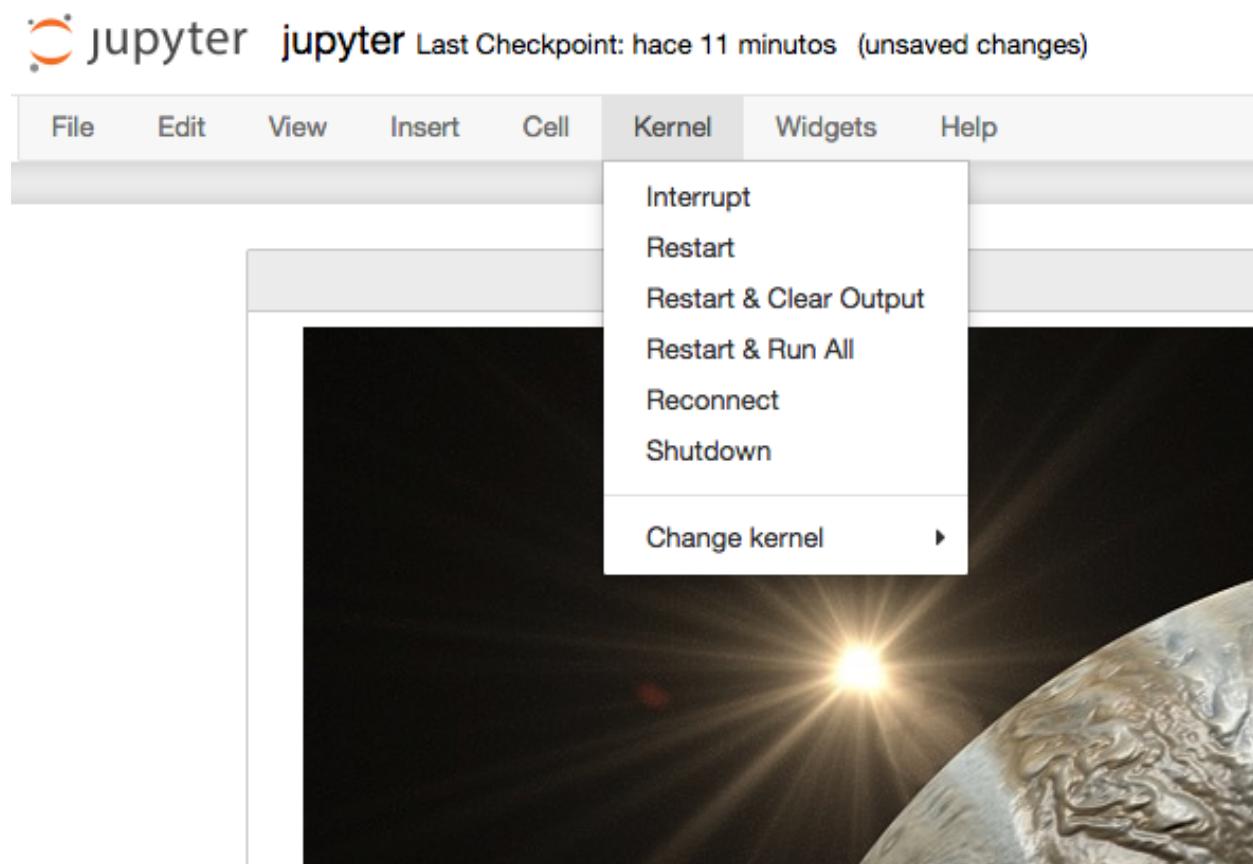


Figura 7: Menú Kernel de Jupyter Notebook

El **kernel** es la capa de software que se encarga de ejecutar las celdas de nuestro «notebook» que contienen código. Podemos tener instalados distintos «kernels» para un mismo Jupyter Notebook. El kernel se puede interrumpir o reiniciar.

Hay veces, que debido a un error de programación o a procesos muy largos, podemos encontrarnos con el «kernel» bloqueado durante un largo período de tiempo. En estas ocasiones es útil reiniciarlo para salvar esa situación.

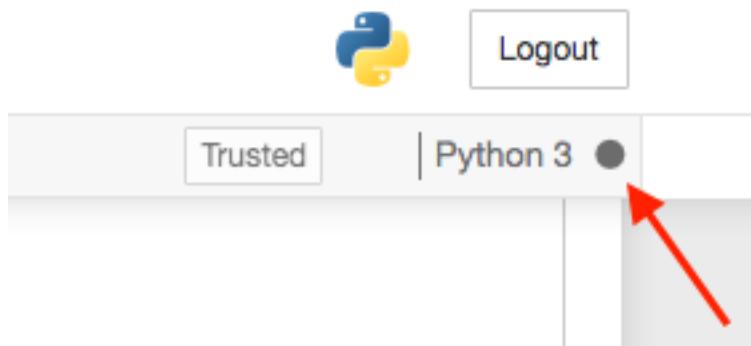


Figura 8: Kernel ocupado

## Menú Ayuda

Como cualquier aplicación, existe un menú de ayuda en el que se pueden encontrar enlaces a referencias y manuales.

Uno de los elementos más interesantes de la ayuda es el uso de los «shortcuts»<sup>3</sup>. Aunque hay muchos, dejamos aquí algunos de los más útiles:

Shortcut	Acción
SHIFT + ENTER	Ejecutar la celda actual
ALT + ENTER	Ejecutar la celda actual y «abrir» una celda debajo
a	Abrir una celda encima de la actual («above»)
b	Abrir una celda debajo de la actual («below»)
m	Convertir la celda actual a Markdown
y	Convertir la celda actual a código
dd	Borrar la celda actual

### 8.1.3 MathJax

MathJax es una biblioteca javascript que permite visualizar fórmulas matemáticas en navegadores web, utilizando (entre otros) el lenguajes de marcado LaTeX. Para escribir fórmulas matemáticas la celda debe ser de tipo Markdown y tendremos que usar delimitadores especiales.

**Fórmulas «en línea»:** Se debe usar el delimitador dólar antes y después de la expresión \$  
... \$

Por ejemplo: \$ \sum\_{x=1}^n \sin(x) + \cos(x) \$ produce : $\sum_{x=1}^n \sin(x) + \cos(x)$

**Fórmulas «de bloque»:** Se debe usar el delimitador doble dólar antes y después de la expresión \$\$ ... \$\$

<sup>3</sup> Un «shortcut» es un «atajo de teclado» (combinación de teclas) para lanzar una determinada acción.

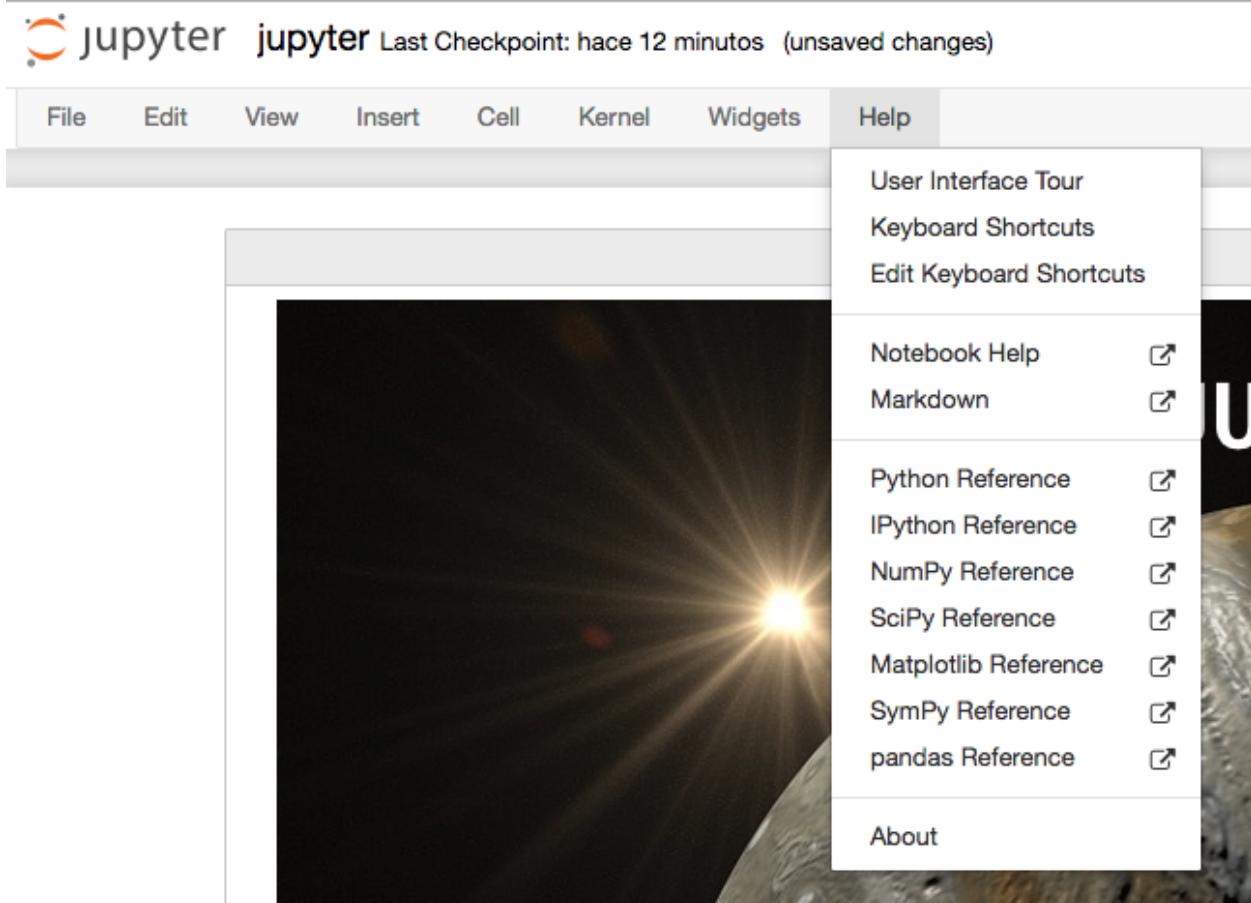


Figura 9: Menú Ayuda de Jupyter Notebook

Por ejemplo: \$\$ \sum\_{x=1}^n \sin(x) + \cos(x) \$\$ produce:

$$\sum_{x=1}^n \sin(x) + \cos(x)$$

## Ejemplos de fórmulas

A continuación veremos distintas fórmulas inspiradas en Motivating Examples de la documentación oficial de Jupyter Notebook. Nótese que aunque no se estén indicando los delimitadores \$\$ sí habría que ponerlos para conseguir el efecto deseado.

Ecuaciones en varias líneas:

```
\dot{x} = \sigma(y-x) \\
\dot{y} = \rho x - y - xz \\
\dot{z} = -\beta z + xy
```

$$\begin{aligned}\dot{x} &= \sigma(y - x) \\ \dot{y} &= \rho x - y - xz \\ \dot{z} &= -\beta z + xy\end{aligned}$$

Ecuaciones en varias líneas (con alineación):

```
\begin{aligned}
\dot{x} &= \sigma(y-x) \\
\dot{y} &= \rho x - y - xz \\
\dot{z} &= -\beta z + xy
\end{aligned}
```

$$\begin{aligned}\dot{x} &= \sigma(y - x) \\ \dot{y} &= \rho x - y - xz \\ \dot{z} &= -\beta z + xy\end{aligned}$$

Usando paréntesis:

```
\left( \sum_{k=1}^n a_k b_k \right)^2 \leq
\left( \sum_{k=1}^n a_k^2 \right) \left( \sum_{k=1}^n b_k^2 \right)
```

$$\left( \sum_{k=1}^n a_k b_k \right)^2 \leq \left( \sum_{k=1}^n a_k^2 \right) \left( \sum_{k=1}^n b_k^2 \right)$$

Trabajando con matrices:

```
\mathbf{V}_1 \times \mathbf{V}_2 =
\begin{vmatrix}
\mathbf{i} & \mathbf{j} & \mathbf{k} \\

```

(continué en la próxima página)

(provien de la página anterior)

```
\frac{\partial X}{\partial u} & \frac{\partial Y}{\partial u} & 0 \\
\frac{\partial X}{\partial v} & \frac{\partial Y}{\partial v} & 0
\end{vmatrix}
```

$$\mathbf{V}_1 \times \mathbf{V}_2 = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \frac{\partial X}{\partial u} & \frac{\partial Y}{\partial u} & 0 \\ \frac{\partial X}{\partial v} & \frac{\partial Y}{\partial v} & 0 \end{vmatrix}$$

Algo de probabilidad:

```
P(E) = {n \choose k} p^k (1-p)^{n-k}
```

$$P(E) = \binom{n}{k} p^k (1 - p)^{n-k}$$

Algunos ejemplos con fracciones:

```
\frac{1}{\Bigl(\sqrt{\phi}\sqrt{5}-\phi\Bigr)e^{\frac{25}{5}\pi}} =
1+\frac{e^{-2\pi}}{1+\frac{e^{-4\pi}}{1+\frac{e^{-6\pi}}{1+\frac{e^{-8\pi}}{1+\dots}}}}
```

$$\frac{1}{\left(\sqrt{\phi}\sqrt{5}-\phi\right)e^{\frac{2}{5}\pi}} = 1 + \frac{e^{-2\pi}}{1 + \frac{e^{-4\pi}}{1 + \frac{e^{-6\pi}}{1 + \frac{e^{-8\pi}}{1 + \dots}}}}$$

```
1 + \frac{q^2}{(1-q)} + \frac{q^6}{(1-q)(1-q^2)} + \dots =
\prod_{j=0}^{\infty} \frac{1}{(1-q^{5j+2})(1-q^{5j+3})}, \\
\quad \quad \quad \text{for } |q| < 1.
```

Múltiples puntos de alineación:

```
\begin{eqnarray}
x' &=& x \sin \phi + z \cos \phi \\
z' &=& -x \cos \phi + z \sin \phi
\end{eqnarray}
```

$$\begin{aligned} x' &= x \sin \phi + z \cos \phi \\ z' &= -x \cos \phi + z \sin \phi \end{aligned}$$

---

## Ejercicio

Escriba en MathJax las siguientes ecuaciones:

### Ecuación 1

$$\int_a^b f'(x)dx = f(b) - f(a)$$

**Ecuación 2**

$$t' = t \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

**Ecuación 3**

$$\left[ M \frac{\partial}{\partial M} + \beta(g) \frac{\partial}{\partial g} + \eta\gamma \right] G^n(x_1, x_2, \dots, x_n; M, g) = 0$$

**Ecuación 4**

$$R_{00} \approx -\frac{1}{2} \sum_i \frac{\partial^2 h_{00}}{\partial (x^i)^2} = \frac{4\pi G}{c^2} (\rho c^2) \Rightarrow \nabla^2 \phi_g = 4\pi G \rho$$

**Truco:** Puede encontrar símbolos matemáticos para Latex en este enlace así como dibujar directamente un símbolo y obtener su referencia a través de la herramienta Detexify.

### 8.1.4 Comandos especiales

Jupyter Notebook ofrece una gama de comandos especiales que cubren gran variedad de funcionalidades.

#### Comandos de shell

Podemos ejecutar comandos de «shell» usando el prefijo exclamación !

```
>>> !date
martes, 15 de junio de 2021, 09:13:25 WEST
```

```
>>> !whoami
sdelquin
```

#### Ejercicio

Ejecute los siguientes comandos del sistema y obtenga la salida en una celda del Notebook:

Windows	Linux & macOS
time	date
dir	ls
mem	free

## Obteniendo ayuda

Una de las formas más sencillas de obtener información de librerías, funciones o módulos es utilizar el sufijo interrogación ?

```
>>> import random

>>> random.randint?
Signature: random.randint(a, b)
Docstring:
Return random integer in range [a, b], including both end points.

File:      ~/.pyenv/versions/3.9.1/lib/python3.9/random.py
Type:      method
```

---

## Ejercicio

Obtenga la documentación de las siguientes funciones:

- `os.path.dirname`
  - `re.match`
  - `datetime.timedelta`
- 

## Comandos mágicos

Jupyter Notebook, o mejor expresado IPython, admite un conjunto de [comandos mágicos](#) que permiten realizar distintas tareas, en muchos casos, no necesariamente relacionadas con Python:

```
>>> %lsmagic
Available line magic:
%import %alias %alias_magic %autoawait %autocall %autoindent %automagic
%autoreload %bookmark %cat %cd %clear %colors %conda %config %cp %cpaste
%debug %dhist %dirs %doctest_mode %ed %edit %env %gui %hist %history
%killbgscripts %ldir %less %lf %lk %ll %load %load_ext %loadpy %logoff
%logon %logstart %logstate %logstop %ls %lsmagic %lx %macro %magic %man
%matplotlib %mkdir %more %mv %notebook %page %paste %pastebin %pdb %pdef
%pdoc %pfile %pinfo %pinfo2 %pip %popd %pprint %precision %prun %psearch
%psource %pushd %pycat %pylab %quickref %recall %rehashx %reload_ext
%rep %rerun %reset %reset_selective %rm %rmdir %run %save %sc %set_env
%store %sx %system %tb %time %timeit %unalias %unload_ext %who %who_ls
%whos %xdel %xmode
```

(continué en la próxima página)

(provien de la página anterior)

Available cell magics:

```
%%! %%HTML %%SVG %%bash %%capture %%debug %%file %%html %%javascript %%js %
→ %%latex %%markdown %%perl %%prun %%pypy %%python %%python2 %%python3 %%ruby_
→ %%script %%sh %%svg %%sx %%system %%time %%timeit %%writefile
```

Automagic is ON, % prefix IS NOT needed for line magics.

Si nos fijamos en el último mensaje, al estar habilitado el modo «automagic», no es estrictamente necesario que usemos el prefijo % para hacer uso de estos comandos. Por ejemplo, si quisiéramos conocer la *historia de comandos* en el intérprete:

```
>>> hist # equivalente a %hist
!date
import random
random.randint?
%lsmagic
pwd
hist
```

## Representando gráficas

Otra de las grandes ventajas que ofrece Jupyter Notebook es poder graficar directamente sobre el cuaderno. Para ello utilizamos código Python (en este caso) y una directiva de comando mágico para indicar que se renderice en línea:

```
>>> %matplotlib inline

>>> from matplotlib import pyplot as plt

>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> y = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

>>> plt.plot(x, y)
[<matplotlib.lines.Line2D at 0x106414e50>]
<Figure size 432x288 with 1 Axes>
```

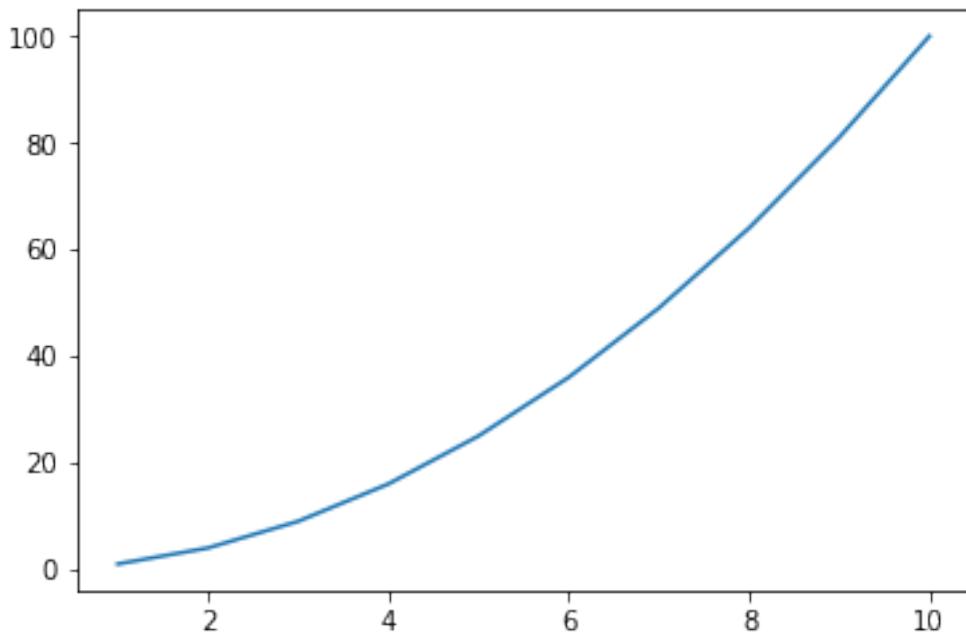


Figura 10: Gráfica sencilla hecha en Jupyter Notebook

## Manejando ficheros

**Cargando un fichero en la celda actual:** Para ello utilizamos el comando `%load "ruta/al/fichero"`

**Ejecutando un fichero en la celda actual:** Para ello utilizamos el comando `%run "ruta/al/fichero"`

**Escribiendo el contenido de la celda actual a fichero:** Para ello utilizamos el comando `%writefile "ruta/al/fichero"` como primera línea de la celda y después vendría el código que queremos escribir.

---

## Ejercicio

- En una celda del «notebook», escriba código Python para crear una lista de 100 números pares.
  - Guarde el contenido de esa celda un fichero Python usando `%%writefile`
  - Carge este fichero en una celda con `%load`
  - Ejecútelo con `%run`
-

## Tiempos de ejecución

Para medir el tiempo de ejecución de una determinada instrucción Python podemos utilizar el comando `%timeit` que calcula un promedio tras correr repetidas veces el código indicado:

```
>>> import numpy

>>> %timeit numpy.random.normal(size=100)
3.03 µs ± 6.77 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

De igual forma, existe un mecanismo para medir el tiempo de ejecución de una celda completa. En este caso se utiliza el comando `%%timeit` (nótese la diferencia del doble porcentaje como prefijo):

```
%%timeit

numpy.random.poisson(size=100)
numpy.random.uniform(size=100)
numpy.random.logistic(size=100)

8.88 µs ± 25.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

## Ejercicio

Mida si hay diferencias significativas en tiempos de ejecución en la creación de distribuciones aleatorias atendiendo a:

- Tipo de distribución (*Poisson, Uniform, Logistic*).
- Tamaño de la muestra (100, 10000, 1000000).

## Incluyendo otros lenguajes

**Celdas con HTML:** Si necesitamos insertar código HTML en una celda, podemos usar el comando `%%html` al comienzo de la misma:

```
%%html

<iframe src="https://www.google.com/maps/embed?pb=!1m18!1m12!1m3!1d3592984.
˓→8538165656!2d-18.096789575396794!3d28.426067294993228!2m3!1f0!2f0!3f0!3m2!
˓→1i1024!2i768!4f13.1!3m3!1m2!1s0xc41aa86ef755363%3A0x10340f3be4bc8c0!
˓→2sCanarias!5e0!3m2!1ses!2ses!4v1623755509663!5m2!1ses!2ses" width="400"
˓→height="300" style="border:0;" allowfullscreen="" loading="lazy"></iframe>
```



**Celdas con «shell script»:** Hay ocasiones en las que un código en shell script suele ser útil. Para incluirlo recurrimos al comando `%%bash` al principio de la celda:

```
%%bash

!tree -d -L 2
.
└── __pycache__
    ├── _build
    └── __init__.py
        ├── html
        ├── static
        ├── css
        ├── img
        ├── js
        └── core
            ├── controlflow
            ├── datastructures
            ├── datatypes
            ├── deenv
            ├── introduction
            ├── modularity
            └── miniprojects
```

(continué en la próxima página)

(proviene de la página anterior)

```

□  ⋮  ⋮  ⋮  spotify
⋮  ⋮  ⋮  pypi
□  ⋮  ⋮  datascience
⋮  ⋮  stdlib
    ⋮  ⋮  text_processing

20 directories

```

**Celdas con perl:** No hay que subestimar el poder del lenguaje de programación perl. Si fuera necesario, lo podemos incluir en una celda del «notebook» con `%%perl` al comienzo de la misma:

```

%%perl

my $email = 'sdelquin@gmail.com';

if ($email =~ /^[^@]+\@[.]+$/) {
    print "Username is: $1\n";
    print "Hostname is: $2\n";
}

...

Username is: sdelquin
Hostname is: gmail.com

```

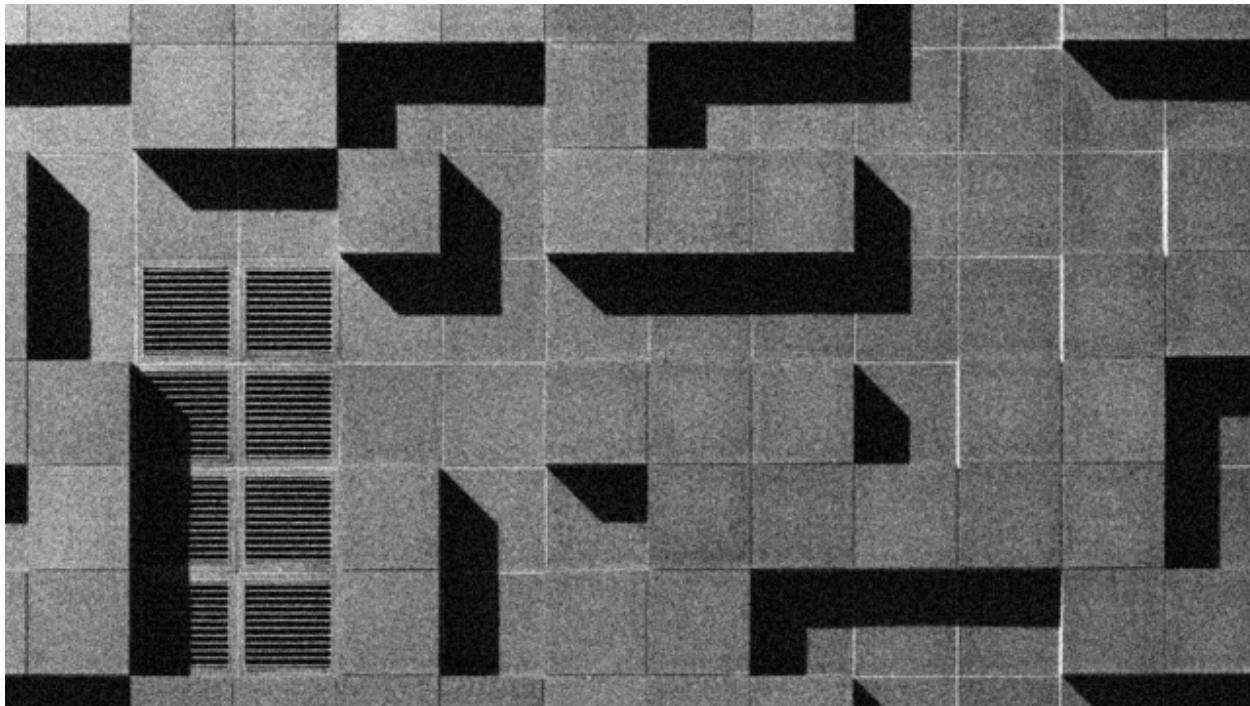
### 8.1.5 Extensiones

El ecosistema de Jupyter Notebook es muy amplio y ofrece una gran variedad de extensiones que se pueden incluir en la instalación que tengamos: [Unofficial Jupyter Notebook Extensions](#).

Su instalación es tan sencilla como:

```
$ pip install jupyter_contrib_nbextensions
```

## 8.2 numpy



NumPy es el paquete fundamental para computación científica en Python y manejo de arrays numéricos multi-dimensionales.<sup>1</sup>

```
$ pip install numpy
```

### 8.2.1 ndarray

En el núcleo de NumPy está el `ndarray`, donde «nd» es por n-dimensional. Un ndarray es un array multidimensional de **elementos del mismo tipo**.

Aquí tenemos una diferencia fundamental con las *listas* en Python que pueden mantener objetos heterogéneos. Y esta característica propicia que el rendimiento de un ndarray sea bastante mejor que el de una lista convencional.

Para crear un array podemos usar:

```
>>> import numpy as np  
  
>>> x = np.array([1, 2, 3, 4, 5])  
  
>>> x
```

(continué en la próxima página)

<sup>1</sup> Foto original de portada por Vlado Paunovic en Unsplash.

(proviene de la página anterior)

```
array([1, 2, 3, 4, 5])
```

```
>>> type(x)
numpy.ndarray
```

Si queremos obtener información sobre el array creado, podemos acceder a distintos atributos del mismo:

```
>>> x.ndim # dimensión
1

>>> x.size # tamaño total del array
5

>>> x.shape # forma
(5,)

>>> x.dtype # tipo de sus elementos
dtype('int64')
```

## Datos heterogéneos

Hemos dicho que los ndarray son estructuras de datos que almacenan un único tipo de datos. A pesar de esto, es posible crear un array con los siguientes valores:

```
>>> x = np.array([4, 'Einstein', 1e-7])
```

Aunque, a priori, puede parecer que estamos mezclando tipos enteros, flotantes y cadenas de texto, lo que realmente se produce (de forma implícita) es una coerción<sup>2</sup> de tipos a **Unicode**:

```
>>> x
array(['4', 'Einstein', '1e-07'], dtype='<U32')

>>> x.dtype
dtype('<U32')
```

<sup>2</sup> Característica de los lenguajes de programación que permite, implícita o explícitamente, convertir un elemento de un tipo de datos en otro, sin tener en cuenta la comprobación de tipos.

## Tipos de datos

NumPy maneja gran cantidad de tipos de datos. A diferencia de los *tipos de datos numéricos en Python* que no establecen un tamaño de bytes de almacenamiento, aquí sí hay una diferencia clara.

Algunos de los tipos de datos numéricos en NumPy se presentan en la siguiente tabla:

Tabla 1: Tipos de datos numéricos en NumPy

dtype	Descripción	Rango
np.int32	Integer	De -2147483648 a 2147483647
np.int64	Integer	De -9223372036854775808 a 9223372036854775807
np.uint32	Unsigned integer	De 0 a 4294967295
np.uint64	Unsigned integer	De 0 a 18446744073709551615
np.float32	Float	De -3.4028235e+38 a 3.4028235e+38
np.float64	Float	De -1.7976931348623157e+308 a 1.7976931348623157e+308

---

**Truco:** NumPy entiende por defecto que `int` hace referencia a `np.int64` y que `float` hace referencia a `np.float64`. Son «alias» bastante utilizados.

---

Si creamos un array de **números enteros**, el tipo de datos por defecto será `int64`:

```
>>> a = np.array(range(10))

>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> a.dtype
dtype('int64')
```

Sin embargo podemos especificar el tipo de datos que nos interese:

```
>>> b = np.array(range(10), dtype='int32') # 'int32' hace referencia a np.int32

>>> b
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int32)

>>> b.dtype
dtype('int32')
```

Lo mismo ocurre con **valores flotantes**, donde `float64` es el tipo de datos por defecto.

Es posible convertir el tipo de datos que almacena un array mediante el método `astype`. Por

ejemplo:

```
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> c = a.astype(float)

>>> c.dtype
dtype('float64')
```

## ndarray vs list

Como ya se ha comentado en la introducción de esta sección, el uso de `ndarray` frente a `list` está justificado por cuestiones de rendimiento. Pero veamos un ejemplo clarificador en el que sumamos 10 millones de valores enteros:

```
>>> array_as_list = list(range(int(10e6)))
>>> array_as_ndarray = np.array(array_as_list)

>>> %timeit sum(array_as_list)
48 ms ± 203 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

>>> %timeit array_as_ndarray.sum()
3.83 ms ± 4.84 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

---

**Nota:** El cómputo es casi 12 veces más rápido utilizando ndarray frente a listas clásicas.

---

En cualquier caso, existe la posibilidad de **convertir a lista** cualquier ndarray mediante el método `tolist()`:

```
>>> a = np.array([10, 20, 30])

>>> a
array([10, 20, 30])

>>> b = a.tolist()

>>> b
[10, 20, 30]

>>> type(b)
list
```

## Matrices

Una matriz no es más que un array bidimensional. Como ya se ha comentado, NumPy provee `ndarray` que se comporta como un array multidimensional con lo que podríamos crear una matriz sin mayor problema.

Veamos un ejemplo en el que tratamos de construir la siguiente matriz:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

Nos apoyamos en una *lista de listas* para la creación de la matriz:

```
>>> M = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])  
  
>>> M  
array([[ 1,  2,  3],  
       [ 4,  5,  6],  
       [ 7,  8,  9],  
       [10, 11, 12]])  
  
>>> M.ndim # bidimensional  
2  
  
>>> M.size  
12  
  
>>> M.shape # 4 filas x 3 columnas  
(4, 3)  
  
>>> M.dtype  
dtype('int64')
```

---

## Ejercicio

Cree los siguientes arrays en NumPy:

$$\begin{aligned} \text{array1} &= [88 \ 23 \ 39 \ 41] \\ \text{array2} &= \begin{bmatrix} 76.4 & 21.7 & 38.4 \\ 41.2 & 52.8 & 68.9 \end{bmatrix} \\ \text{array3} &= \begin{bmatrix} 12 \\ 4 \\ 9 \\ 8 \end{bmatrix} \end{aligned}$$

Obtenga igualmente las siguientes características de cada uno de ellos: dimensión, tamaño, forma y tipo de sus elementos.

## Cambiando la forma

Dado un array, podemos cambiar su forma mediante la función `np.reshape()`:

```
>>> a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])  
  
>>> np.reshape(a, (3, 4)) # 3 x 4  
array([[1, 2, 3, 4],  
       [5, 6, 7, 8],  
       [9, 10, 11, 12]])
```

Si sólo queremos especificar un número determinado de filas o columnas, podemos dejar la otra dimensión a -1:

```
>>> np.reshape(a, (6, -1)) # 6 filas  
array([[1, 2],  
       [3, 4],  
       [5, 6],  
       [7, 8],  
       [9, 10],  
       [11, 12]])  
  
>>> np.reshape(a, (-1, 3)) # 3 columnas  
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9],  
       [10, 11, 12]])
```

**Advertencia:** En el caso de que no exista posibilidad de cambiar la forma del array por el número de filas y/o columnas especificado, obtendremos un error de tipo `ValueError: cannot reshape array`.

## Almacenando arrays

Es posible que nos interese almacenar (de forma persistente) los arrays que hemos ido creando. Para ello NumPy nos provee, al menos, de dos mecanismos:

**Almacenamiento en formato binario propio:** Mediante el método `save()` podemos guardar la estructura de datos en [ficheros .npy](#). Veamos un ejemplo:

```
>>> M
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

>>> np.save('my_matrix', M)

>>> !ls my_matrix.npy
my_matrix.npy

>>> M_reloaded = np.load('my_matrix.npy')

>>> M_reloaded
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

**Almacenamiento en formato de texto plano:** NumPy proporciona el método `savetxt()` con el que podremos volcar la estructura de datos a un fichero de texto csv. Veamos un ejemplo:

```
>>> M
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

>>> np.savetxt('my_matrix.csv', M, fmt='%d')

>>> !cat my_matrix.csv
1 2 3
4 5 6
7 8 9
10 11 12

>>> M_reloaded = np.loadtxt('my_matrix.csv', dtype=int)
```

(continué en la próxima página)

(provien de la página anterior)

```
>>> M_reloaded  
array([[ 1,  2,  3],  
       [ 4,  5,  6],  
       [ 7,  8,  9],  
       [10, 11, 12]])
```

---

**Truco:** Por defecto el almacenamiento y la carga de arrays en formato texto usa tipos de **datos flotantes**. Es por ello que hemos usado el parámetro `fmt` en el almacenamiento y el parámetro `dtype` en la carga.

---

Es posible cargar un array **desempaquetando sus valores** a través del parámetro `unpack`. En el siguiente ejemplo sepáramos las columnas en tres variables diferentes:

```
>>> M  
array([[ 1,  2,  3],  
       [ 4,  5,  6],  
       [ 7,  8,  9],  
       [10, 11, 12]])  
  
>>> col1, col2, col3 = np.loadtxt('my_matrix.csv', unpack=True, dtype=int)  
  
>>> col1  
array([ 1,  4,  7, 10])  
>>> col2  
array([ 2,  5,  8, 11])  
>>> col3  
array([ 3,  6,  9, 12])
```

## 8.2.2 Funciones predefinidas para creación de arrays

NumPy ofrece una gran variedad de funciones predefinidas para creación de arrays que nos permiten simplificar el proceso de construcción de este tipo de estructuras de datos.

### Valores fijos

A continuación veremos una serie de funciones para crear arrays con valores fijos.

## Ceros

```
>>> np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

Por defecto, ésta y otras funciones del estilo, devuelven **valores flotantes**. Si quisieramos trabajar con valores enteros basta con usar el parámetro `dtype`:

```
>>> np.zeros((3, 4), dtype=int)
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

Existe la posibilidad de crear un array de **ceros con las mismas dimensiones** (y forma) que otro array:

```
>>> M = np.array([[1, 2, 3], [4, 5, 6]])
>>> M
array([[1, 2, 3],
       [4, 5, 6]])

>>> np.zeros_like(M)
array([[0, 0, 0],
       [0, 0, 0]])
```

Lo cual sería equivalente a pasar la «forma» del array a la función predefinida de creación de ceros:

```
>>> np.zeros(M.shape, dtype=int)
array([[0, 0, 0],
       [0, 0, 0]])
```

## Unos

```
>>> np.ones((3, 4)) # también existe np.ones_like()
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

## Mismo valor

```
>>> np.full((3, 4), 7) # también existe np.full_like()
array([[7, 7, 7, 7],
       [7, 7, 7, 7],
       [7, 7, 7, 7]])
```

## Matriz identidad

```
>>> np.eye(5)
array([[1.,  0.,  0.,  0.,  0.],
       [0.,  1.,  0.,  0.,  0.],
       [0.,  0.,  1.,  0.,  0.],
       [0.,  0.,  0.,  1.,  0.],
       [0.,  0.,  0.,  0.,  1.]])
```

## Matriz diagonal

```
>>> np.diag([5, 4, 3, 2, 1])
array([[5,  0,  0,  0,  0],
       [0,  4,  0,  0,  0],
       [0,  0,  3,  0,  0],
       [0,  0,  0,  2,  0],
       [0,  0,  0,  0,  1]])
```

---

## Ejercicio

Cree la siguiente matriz mediante código Python:

$$\text{diagonal} = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 2 & \dots & 0 \\ \vdots & \vdots & 0 & \ddots & 0 \\ 0 & 0 & 0 & \dots & 49 \end{bmatrix}$$

Obtenga igualmente las siguientes características de cada uno de ellos: dimensión, tamaño, forma y tipo de sus elementos.

---

## Valores equiespaciados

A continuación veremos una serie de funciones para crear arrays con valores equiespaciados o en intervalos definidos.

### Valores enteros equiespaciados

La función que usamos para este propósito es `np.arange()` cuyo comportamiento es totalmente análogo a la función «built-in» `range()`.

Especificando límite superior:

```
>>> np.arange(21)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20])
```

Especificando límite inferior y superior:

```
>>> np.arange(6, 60)
array([ 6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
       23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
       40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56,
       57, 58, 59])
```

Especificando límite inferior, superior y paso:

```
>>> np.arange(6, 60, 3)
array([ 6,  9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54,
       57])
```

Es posible especificar un **paso flotante** en la función `arange()`:

```
>>> np.arange(6, 16, .3)
array([ 6. ,  6.3,  6.6,  6.9,  7.2,  7.5,  7.8,  8.1,  8.4,  8.7,  9. ,
       9.3,  9.6,  9.9, 10.2, 10.5, 10.8, 11.1, 11.4, 11.7, 12. , 12.3,
       12.6, 12.9, 13.2, 13.5, 13.8, 14.1, 14.4, 14.7, 15. , 15.3, 15.6,
       15.9])
```

## Valores flotantes equiespaciados

La función que usamos para este propósito es `np.linspace()` cuyo comportamiento es «similar» a `np.arange()` pero para valores flotantes.

### Especificando límite inferior y superior:

```
>>> np.linspace(6, 60) # [6, 60] con 50 valores
array([ 6.          ,  7.10204082,  8.20408163,  9.30612245, 10.40816327,
       11.51020408, 12.6122449 , 13.71428571, 14.81632653, 15.91836735,
       17.02040816, 18.12244898, 19.2244898 , 20.32653061, 21.42857143,
       22.53061224, 23.63265306, 24.73469388, 25.83673469, 26.93877551,
       28.04081633, 29.14285714, 30.24489796, 31.34693878, 32.44897959,
       33.55102041, 34.65306122, 35.75510204, 36.85714286, 37.95918367,
       39.06122449, 40.16326531, 41.26530612, 42.36734694, 43.46938776,
       44.57142857, 45.67346939, 46.7755102 , 47.87755102, 48.97959184,
       50.08163265, 51.18367347, 52.28571429, 53.3877551 , 54.48979592,
       55.59183673, 56.69387755, 57.79591837, 58.89795918, 60.        ])
```

---

**Nota:** Por defecto `np.linspace()` genera 50 elementos.

---

### Especificando límite inferior, superior y total de elementos:

```
>>> np.linspace(6, 60, 20) # [6, 60] con 20 valores
array([ 6.          ,  8.84210526, 11.68421053, 14.52631579, 17.36842105,
       20.21052632, 23.05263158, 25.89473684, 28.73684211, 31.57894737,
       34.42105263, 37.26315789, 40.10526316, 42.94736842, 45.78947368,
       48.63157895, 51.47368421, 54.31578947, 57.15789474, 60.        ])
```

---

**Importante:** A diferencia de `np.arange()`, la función `np.linspace()` incluye siempre entre los valores generados el límite superior especificado.

---

### Especificando un intervalo abierto $[a, b)$ :

```
>>> np.linspace(6, 60, 20, endpoint=False) # [6, 60) con 20 elementos
array([ 6. ,  8.7, 11.4, 14.1, 16.8, 19.5, 22.2, 24.9, 27.6, 30.3, 33. ,
       35.7, 38.4, 41.1, 43.8, 46.5, 49.2, 51.9, 54.6, 57.3])
```

## Valores aleatorios

A continuación veremos una serie de funciones para crear arrays con valores aleatorios y distribuciones de probabilidad.

### Valores aleatorios enteros

Valores aleatorios enteros en  $[a, b]$ :

```
>>> np.random.randint(3, 30) # escalar  
4
```

```
>>> np.random.randint(3, 30, size=9) # vector  
array([29, 7, 8, 21, 27, 23, 29, 15, 28])
```

```
>>> np.random.randint(3, 30, size=(3, 3)) # matriz  
array([[24, 4, 29],  
       [10, 22, 27],  
       [27, 7, 20]])
```

### Valores aleatorios flotantes

Por simplicidad, en el resto de ejemplos vamos a obviar la salida *escalar* y *matriz*.

Valores aleatorios flotantes en  $[0, 1]$ :

```
>>> np.random.random(9)  
array([0.53836208, 0.78315275, 0.6931254 , 0.97194325, 0.01523289,  
     0.47692141, 0.27653964, 0.82297655, 0.70502383])
```

Valores aleatorios flotantes en  $[a, b]$ :

```
>>> np.random.uniform(1, 100, size=9)  
array([17.00450378, 67.08416159, 56.99930273, 9.19685998, 35.27334323,  
     97.34651516, 25.89283558, 53.59685476, 72.74943888])
```

## Distribuciones de probabilidad

**Distribución normal:** Ejemplo en el que generamos un millón de valores usando como parámetros de la distribución  $\mu = 0, \sigma = 5$

```
>>> dist = np.random.normal(0, 5, size=1_000_000)

>>> dist[:20]
array([ 2.5290643 ,  1.46577658,  1.65170437, -1.36970819, -2.24547757,
       7.19905613, -4.4666239 , -1.05505116,  2.42351298, -4.45314272,
      1.13604077, -2.85054948,  4.34589478, -2.81235743, -0.8215143 ,
      0.57796411, -2.56594122, -7.14899388,  3.49197644,  1.80691996])

>>> dist.mean()
0.004992046432131982

>>> dist.std()
4.998583810032169
```

**Muestra aleatoria:** Ejemplo en el que generamos una muestra aleatoria de un millón de lanzamientos de una moneda:

```
>>> coins = np.random.choice(['head', 'tail'], size=1_000_000)

>>> coins
array(['tail', 'head', 'tail', ..., 'tail', 'head', 'tail'], dtype='<U4')

>>> sum(coins == 'head')
499874
>>> sum(coins == 'tail')
500126
```

**Muestra aleatoria con probabilidades no uniformes:** Ejemplo en el que generamos una muestra aleatoria de un millón de lanzamientos con un dado «trucado»:

```
>>> # La cara del "1" tiene un 50% de probabilidades de salir
>>> dices = np.random.choice(range(1, 7), size=1_000_000, p=[.5, .1, .1, .1, .1,
   > .1])

>>> dices
array([6, 5, 4, ..., 1, 6, 1])

>>> sum(dices == 1)
500290
>>> sum(dices == 6)
99550
```

**Muestra aleatoria sin reemplazo:** Ejemplo en el que seleccionamos 5 principios aleatorios del *Zen de Python* sin reemplazo:

```
>>> import this
>>> import codecs

>>> zen = codecs.decode(this.s, 'rot-13').splitlines()[3:] # https://bit.ly/
↪ 3xhsucQ

>>> np.random.choice(zen, size=5, replace=False)
array(['Unless explicitly silenced.',
       "Although that way may not be obvious at first unless you're Dutch.",
       'Sparse is better than dense.',
       'If the implementation is easy to explain, it may be a good idea.',
       'Complex is better than complicated.'], dtype='|<U69')
```

---

## Ejercicio

Cree:

- Una matriz de 20 filas y 5 columnas con valores flotantes equiespaciados en el intervalo cerrado  $[1, 10]$ .
  - Un array unidimensional con 128 valores aleatorios de una distribución normal  $\mu = 1, \sigma = 2$ .
  - Un array unidimensional con 15 valores aleatorios de una muestra 1,  $X_2$  donde la probabilidad de que gane el equipo local es del 50%, la probabilidad de que empaten es del 30% y la probabilidad de que gane el visitante es del 20%.
- 

### 8.2.3 Manipulando elementos

Los arrays multidimensionales de NumPy están indexados por unos ejes que establecen la forma en la que debemos acceder a sus elementos. Véase el siguiente diagrama:

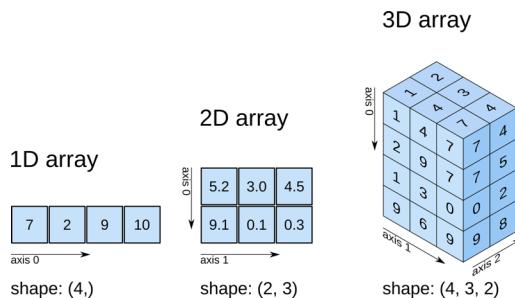


Figura 11: Esquema de ejes sobre los arrays de NumPy<sup>3</sup>

## Arrays unidimensionales

### Acceso a arrays unidimensionales

```
>>> values
array([10, 11, 12, 13, 14, 15])

>>> values[2]
12
>>> values[-3]
13
```

### Modificación a arrays unidimensionales

```
>>> values
array([10, 11, 12, 13, 14, 15])

>>> values[0] = values[1] + values[5]

>>> values
array([26, 11, 12, 13, 14, 15])
```

### Borrado en arrays unidimensionales

```
>>> values
array([10, 11, 12, 13, 14, 15])

>>> np.delete(values, 2) # índice (como escalar)
array([10, 11, 13, 14, 15])

>>> np.delete(values, (2, 3, 4)) # indices (como tupla)
array([10, 11, 15])
```

---

**Nota:** La función `np.delete()` no es destructiva. Devuelve una copia modificada del array.

---

<sup>3</sup> Imagen de Harriet Dashnow, Stéfan van der Walt y Juan Núñez-Iglesias en O'Reilly.

## Inserción en arrays unidimensionales

```
>>> values
array([10, 11, 12, 13, 14, 15])

>>> np.append(values, 16) # añade elementos al final
array([10, 11, 12, 13, 14, 15, 16])

>>> np.insert(values, 1, 101) # añade elementos en una posición
array([10, 101, 11, 12, 13, 14, 15])
```

Para ambas funciones también es posible añadir varios elementos de una sola vez:

```
>>> values
array([10, 11, 12, 13, 14, 15])

>>> np.append(values, [16, 17, 18])
array([10, 11, 12, 13, 14, 15, 16, 17, 18])
```

---

**Nota:** La funciones `np.append()` y `np.insert()` no son destructivas. Devuelven una copia modificada del array.

---

## Arrays multidimensionales

Partimos del siguiente array bidimensional (matriz) para ejemplificar las distintas operaciones:

```
>>> values = np.arange(1, 13).reshape(3, 4)
>>> values
array([[1, 2, 3, 4],
       [5, 6, 7, 8],
       [9, 10, 11, 12]])
```

## Acceso a arrays multidimensionales

```
>>> values
array([[1, 2, 3, 4],
       [5, 6, 7, 8],
       [9, 10, 11, 12]])
```

Acceso a elementos individuales::

```
>>> values[0, 0]
1
>>> values[-1, -1]
12
>>> values[1, 2]
7
```

### Acceso a múltiples elementos::

```
>>> values[[0, 2], [1, 2]] # Elementos [0, 1] y [2, 2]
array([ 2, 11])
```

### Acceso a filas o columnas completas:

```
>>> values[2] # tercera fila
array([ 9, 10, 11, 12])

>>> values[:, 1] # segunda columna
array([ 2,  6, 10])
```

### Acceso a zonas parciales del array:

```
>>> values[0:2, 0:2]
array([[1, 2],
       [5, 6]])

>>> values[0:2, [1, 3]]
array([[2, 4],
       [6, 8]])
```

---

**Importante:** Todos estos accesos crean una copia (vista) del array original. Esto significa que, si modificamos un valor en el array copia, se ve reflejado en el original. Para evitar esta situación podemos usar la función `np.copy()` y desvincular la vista de su fuente.

---

## Modificación de arrays multidimensionales

```
>>> values
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])

>>> values[0, 0] = 100
```

(continué en la próxima página)

(provien de la página anterior)

```
>>> values[1] = [55, 66, 77, 88]

>>> values[:,2] = [30, 70, 110]

>>> values
array([[100,    2,   30,    4],
       [ 55,   66,   70,   88],
       [  9,   10,  110,   12]])
```

## Borrado en arrays multidimensionales

```
>>> values
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])

>>> np.delete(values, 0, axis=0) # Borrado de la primera fila
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])

>>> np.delete(values, (1, 3), axis=1) # Borrado de la segunda y cuarta columna
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11]])
```

---

**Truco:** Tener en cuenta que `axis=0` hace referencia a **filas** y `axis=1` hace referencia a **columnas** tal y como describe el *diagrama* del comienzo de la sección.

---

## Inserción en arrays multidimensionales

Añadir elementos al final del array:

```
>>> values
array([[1, 2],
       [3, 4]])

>>> np.append(values, [[5, 6]], axis=0)
array([[1, 2],
       [3, 4],
```

(continué en la próxima página)

(provien de la página anterior)

```
[5, 6]])

>>> np.append(values, [[5], [6]], axis=1)
array([[1, 2, 5],
       [3, 4, 6]])
```

**Insertar elementos en posiciones arbitrarias del array:**

```
>>> values
array([[1, 2],
       [3, 4]])

>>> np.insert(values, 0, [0, 0], axis=0)
array([[0, 0],
       [1, 2],
       [3, 4]])

>>> np.insert(values, 1, [0, 0], axis=1)
array([[1, 0, 2],
       [3, 0, 4]])
```

**Ejercicio**

Utilizando las operaciones de modificación, borrado e inserción, convierta la siguiente matriz:

$$\begin{bmatrix} 17 & 12 & 31 \\ 49 & 11 & 51 \\ 21 & 31 & 62 \\ 63 & 75 & 22 \end{bmatrix}$$

en esta:

$$\begin{bmatrix} 17 & 12 & 31 & 63 \\ 49 & 11 & 51 & 75 \\ 21 & 31 & 62 & 22 \end{bmatrix}$$

y luego en esta:

$$\begin{bmatrix} 17 & 12 & 31 & 63 \\ 49 & 49 & 49 & 63 \\ 21 & 31 & 62 & 63 \end{bmatrix}$$

## Apilando matrices

Hay veces que nos interesa combinar dos matrices (arrays en general). Una de los mecanismos que nos proporciona NumPy es el **apilado**.

### Apilado vertical:

```
>>> m1 = np.random.randint(1, 100, size=(3, 2))
>>> m2 = np.random.randint(1, 100, size=(1, 2))

>>> m1
array([[68, 68],
       [10, 50],
       [87, 92]])

>>> m2
array([[63, 80]])

>>> np.vstack((m1, m2))
array([[68, 68],
       [10, 50],
       [87, 92],
       [63, 80]])
```

### Apilado horizontal:

```
>>> m1 = np.random.randint(1, 100, size=(3, 2))
>>> m2 = np.random.randint(1, 100, size=(3, 1))

>>> m1
array([[51, 50],
       [52, 15],
       [14, 21]])

>>> m2
array([[18],
       [52],
       [1]])

>>> np.hstack((m1, m2))
array([[51, 50, 18],
       [52, 15, 52],
       [14, 21, 1]])
```

## Repitiendo elementos

**Repetición por ejes:** El parámetro de repetición indica el número de veces que repetimos el array completo por cada eje:

```
>>> values
array([[1, 2],
       [3, 4],
       [5, 6]])

>>> np.tile(values, 3) # x3 en filas y columnas
array([[1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4],
       [5, 6, 5, 6, 5, 6]])

>>> np.tile(values, (2, 3)) # x2 en filas; x3 en columnas
array([[1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4],
       [5, 6, 5, 6, 5, 6],
       [1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4],
       [5, 6, 5, 6, 5, 6]])
```

**Repetición por elementos:** El parámetro de repetición indica el número de veces que repetimos cada elemento del array:

```
>>> values
array([[1, 2],
       [3, 4],
       [5, 6]])

>>> np.repeat(values, 2)
array([1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6])

>>> np.repeat(values, 2, axis=0) # x2 en filas
array([[1, 2],
       [1, 2],
       [3, 4],
       [3, 4],
       [5, 6],
       [5, 6]])

>>> np.repeat(values, 3, axis=1) # x3 en columnas
array([[1, 1, 1, 2, 2, 2],
       [3, 3, 3, 4, 4, 4],
       [5, 5, 5, 6, 6, 6]])
```

## Acceso por diagonal

Es bastante común acceder a elementos de una matriz (array en general) tomando como referencia su diagonal. Para ello, NumPy nos provee de ciertos mecanismos que veremos a continuación.

Para ejemplificarlo, partiremos del siguiente array:

```
>>> values
array([[73, 86, 90, 20],
       [96, 55, 15, 48],
       [38, 63, 96, 95],
       [13, 87, 32, 96]])
```

## Extracción de elementos por diagonal

La función `np.diag()` permite acceder a los elementos de un array especificando un parámetro `k` que indica la «distancia» con la diagonal principal:

Veamos cómo variando el parámetro `k` obtenemos distintos resultados:

```
>>> np.diag(values) # k = 0
array([73, 55, 96, 96])

>>> for k in range(1, values.shape[0]):
...     print(f'k={k}', np.diag(values, k=k))
...
k=1 [86 15 95]
k=2 [90 48]
k=3 [20]

>>> for k in range(1, values.shape[0]):
...     print(f'k={-k}', np.diag(values, k=-k))
...
k=-1 [96 63 32]
k=-2 [38 87]
k=-3 [13]
```

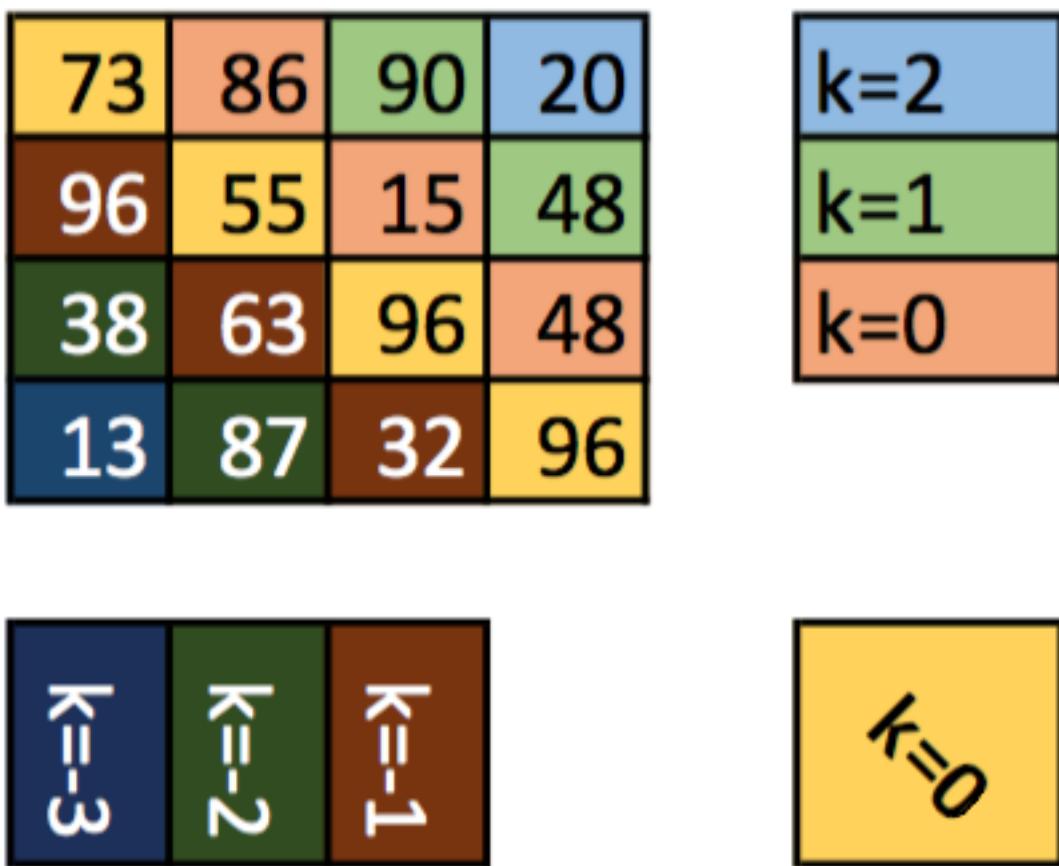


Figura 12: Acceso a elementos de un array por su diagonal

### Modificación de elementos por diagonal

NumPy también provee un método `np.diag_indices()` que retorna los índices de los elementos de la diagonal principal, con lo que podemos modificar sus valores directamente:

```
>>> values
array([[73, 86, 90, 20],
       [96, 55, 15, 48],
       [38, 63, 96, 95],
       [13, 87, 32, 96]])

>>> di = np.diag_indices(values.shape[0])

>>> di
(array([0, 1, 2, 3]), array([0, 1, 2, 3]))

>>> values[di] = 0

>>> values
array([[ 0, 86, 90, 20],
       [96,  0, 15, 48],
       [38, 63,  0, 95],
       [13, 87, 32,  0]])
```

---

**Truco:** Existen igualmente las funciones `np.triu_indices()` y `np.tril_indices()` para obtener los índices de la diagonal superior e inferior de una matriz.

---

## 8.2.4 Operaciones sobre arrays

### Operaciones lógicas

#### Indexado booleano

El indexado booleano es una operación que permite conocer (a nivel de elemento) si un array cumple o no con una determinada condición:

```
>>> values
array([[60, 47, 34, 38],
       [43, 63, 37, 68],
       [58, 28, 31, 43],
       [32, 65, 32, 96]])
```

(continué en la próxima página)

(provine de la página anterior)

```
>>> values > 50 # indexado booleano
array([[ True, False, False, False],
       [False,  True, False,  True],
       [ True, False, False, False],
       [False,  True, False,  True]])

>>> values[values > 50] # uso de máscara
array([60, 63, 68, 58, 65, 96])

>>> values[values > 50] = -1 # modificación de valores

>>> values
array([[ -1,  47,  34,  38],
       [43, -1,  37, -1],
       [-1, 28,  31,  43],
       [32, -1,  32, -1]])
```

Las condiciones pueden ser más complejas e incorporar operadores lógicos | (or) y & (and):

```
>>> values
array([[60, 47, 34, 38],
       [43, 63, 37, 68],
       [58, 28, 31, 43],
       [32, 65, 32, 96]])

>>> (values < 25) | (values > 75)
array([[False, False, False, False],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False,  True]])

>>> (values > 25) & (values < 75)
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True, False]])
```

---

**Consejo:** El uso de paréntesis es obligatorio si queremos mantener la precedencia y que funcione correctamente.

---

## Ejercicio

Extraiga todos los números impares de la siguiente matriz:

$$\text{values} = \begin{bmatrix} 10 & 11 & 12 & 13 \\ 14 & 15 & 16 & 17 \\ 18 & 19 & 20 & 21 \end{bmatrix}$$

---

Si lo que nos interesa es **obtener los índices** del array que satisfacen una determinada condición, NumPy nos proporciona el método `where()` cuyo comportamiento se exemplifica a continuación:

```
>>> values
array([[60, 47, 34, 38],
       [43, 63, 37, 68],
       [58, 28, 31, 43],
       [32, 65, 32, 96]])

>>> idx = np.where(values > 50)

>>> idx
(array([0, 1, 1, 2, 3, 3]), array([0, 1, 3, 0, 1, 3]))

>>> values[idx]
array([60, 63, 68, 58, 65, 96])
```

---

### Ejercicio

Partiendo de una matriz de 10 filas y 10 columnas con valores aleatorios enteros en el intervalo  $[0, 100]$ , realice las operaciones necesarias para obtener una matriz de las mismas dimensiones donde:

- Todos los elementos de la diagonal sean 50.
  - Los elementos mayores que 50 tengan valor 100.
  - Los elementos menores que 50 tengan valor 0.
- 

### Comparando arrays

Dados dos arrays podemos compararlos usando el operador `==` del mismo modo que con cualquier otro objeto en Python. La cuestión es que el resultado se evalúa a nivel de elemento:

```
>>> m1 = np.array([[1, 2], [3, 4]])
>>> m3 = np.array([[1, 2], [3, 4]])
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> m1 == m2
array([[ True,  True],
       [ True,  True]])
```

Si queremos comparar arrays en su totalidad, podemos hacer uso de la siguiente función:

```
>>> np.array_equal(m1, m2)
True
```

## Operaciones de conjunto

Al igual que existen *operaciones sobre conjuntos* en Python, también podemos llevarlas a cabo sobre arrays en NumPy.

### Unión de arrays

 $x \cup y$ 

```
>>> x
array([ 9,  4, 11,  3, 14,  5, 13, 12,  7, 14])
>>> y
array([17,  9, 19,  4, 18,  4,  7, 13, 11, 10])

>>> np.union1d(x, y)
array([ 3,  4,  5,  7,  9, 10, 11, 12, 13, 14, 17, 18, 19])
```

### Intersección de arrays

 $x \cap y$ 

```
>>> x
array([ 9,  4, 11,  3, 14,  5, 13, 12,  7, 14])
>>> y
array([17,  9, 19,  4, 18,  4,  7, 13, 11, 10])

>>> np.intersect1d(x, y)
array([ 4,  7,  9, 11, 13])
```

## Diferencia de arrays

$x \setminus y$

```
>>> x
array([ 9,  4, 11,  3, 14,  5, 13, 12,  7, 14])
>>> y
array([17,  9, 19,  4, 18,  4,  7, 13, 11, 10])

>>> np.setdiff1d(x, y)
array([ 3,  5, 12, 14])
```

## Ordenación de arrays

En términos generales, existen dos formas de ordenar cualquier estructura de datos, una que modifica «in-situ» los valores (destructiva) y otra que devuelve «nuevos» valores (no destructiva). En el caso de NumPy también es así.

### Ordenación sobre arrays unidimensionales

```
>>> values
array([23, 24, 92, 88, 75, 68, 12, 91, 94, 24,  9, 21, 42,  3, 66])

>>> np.sort(values) # no destructivo
array([ 3,  9, 12, 21, 23, 24, 24, 42, 66, 68, 75, 88, 91, 92, 94])

>>> values.sort() # destructivo

>>> values
array([ 3,  9, 12, 21, 23, 24, 24, 42, 66, 68, 75, 88, 91, 92, 94])
```

### Ordenación sobre arrays multidimensionales

```
>>> values
array([[52, 23, 90, 46],
       [61, 63, 74, 59],
       [75,  5, 58, 70],
       [21,  7, 80, 52]])

>>> np.sort(values, axis=1) # equivale a np.sort(values)
array([[23, 46, 52, 90],
```

(continué en la próxima página)

(provien de la página anterior)

```
[59, 61, 63, 74],
[ 5, 58, 70, 75],
[ 7, 21, 52, 80]])

>>> np.sort(values, axis=0)
array([[21,  5, 58, 46],
       [52,  7, 74, 52],
       [61, 23, 80, 59],
       [75, 63, 90, 70]])
```

**Nota:** También existe `values.sort(axis=1)` y `values.sort(axis=0)` como métodos «destructivos» de ordenación.

## Contando valores

Otra de las herramientas útiles que proporciona NumPy es la posibilidad de contar el número de valores que existen en un array en base a ciertos criterios.

Para exemplificarlo, partiremos de un array unidimensional con valores de una distribución aleatoria uniforme en el intervalo [1, 10]:

```
>>> randomized = np.random.randint(1, 11, size=1000)

>>> randomized
array([ 7,  9,  7,  8,  3,  7,  6,  4,  3,  9,  3,  1,  6,  7, 10,  4,  8,
       1,  3,  3,  8,  5,  4,  7,  5,  8,  8,  3, 10,  1,  7, 10,  3, 10,
       2,  9,  5,  1,  2,  4,  4, 10,  5, 10,  5,  2,  5,  2, 10,  3,  4,
       ...])
```

Valores únicos:

```
>>> np.unique(randomized)
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Valores únicos (incluyendo frecuencias):

```
>>> np.unique(randomized, return_counts=True)
(array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]),
array([101,  97, 117,  94, 101,  88,  94, 110,  93, 105]))
```

Valores distintos de cero:

```
>>> np.count_nonzero(randomized)
1000
```

Valores distintos de cero (incluyendo condición):

```
>>> np.count_nonzero(randomized > 5)
490
```

## Operaciones aritméticas

Una de las grandes ventajas del uso de arrays numéricos en NumPy es la posibilidad de trabajar con ellos como si fueran objetos «simples» pero sacando partido de la aritmética vectorial. Esto redundá en una mayor eficiencia y rapidez de cómputo.

### Operaciones aritméticas con mismas dimensiones

Cuando operamos entre arrays de las mismas dimensiones, las operaciones aritméticas se realizan elemento a elemento (ocupando misma posición) y el resultado, obviamente, tiene las mismas dimensiones:

```
>>> m1
array([[21, 86, 45],
       [31, 36, 78],
       [31, 64, 70]])

>>> m2
array([[58, 67, 17],
       [99, 53,  9],
       [92, 42, 75]])

>>> m1 + m2
array([[ 79, 153,  62],
       [130,  89,  87],
       [123, 106, 145]])

>>> m1 - m2
array([[-37,  19,  28],
       [-68, -17,  69],
       [-61,  22,  -5]])

>>> m1 * m2
array([[1218, 5762,  765],
       [3069, 1908,  702],
```

(continuó en la próxima página)

(provine de la página anterior)

```
[2852, 2688, 5250]]))

>>> m1 / m2 # división flotante
array([[0.36206897, 1.28358209, 2.64705882],
       [0.31313131, 0.67924528, 8.66666667],
       [0.33695652, 1.52380952, 0.93333333]])

>>> m1 // m2 # división entera
array([[0, 1, 2],
       [0, 0, 8],
       [0, 1, 0]])
```

## Operaciones aritméticas con distintas dimensiones

Cuando operamos entre arrays con dimensiones diferentes, siempre y cuando se cumplan ciertas restricciones en tamaños de filas y/o columnas, lo que se produce es un «broadcasting» (o difusión) de los valores.

### Suma con array «fila»:

```
>>> m
array([[9, 8, 1],
       [7, 6, 7]])

>>> v
array([[2, 3, 6]])

>>> m + v # broadcasting
array([[11, 11, 7],
       [9, 9, 13]])
```

### Suma con array «columna»:

```
>>> m
array([[9, 8, 1],
       [7, 6, 7]])

>>> v
array([[1],
       [6]])

>>> m + v # broadcasting
array([[10, 9, 2],
       [13, 12, 13]])
```

**Advertencia:** En el caso de que no coincidan dimensiones de filas y/o columnas, NumPy no podrá ejecutar la operación y obtendremos un error `ValueError: operands could not be broadcast together with shapes.`

### Operaciones entre arrays y escalares

Al igual que ocurría en los casos anteriores, si operamos con un array y un escalar, éste último será difundido para abarcar el tamaño del array:

```
>>> m
array([[9, 8, 1],
       [7, 6, 7]])

>>> m + 5
array([[14, 13, 6],
       [12, 11, 12]])

>>> m - 5
array([[ 4,  3, -4],
       [ 2,  1,  2]])

>>> m * 5
array([[45, 40, 5],
       [35, 30, 35]])

>>> m / 5
array([[1.8, 1.6, 0.2],
       [1.4, 1.2, 1.4]])

>>> m // 5
array([[1, 1, 0],
       [1, 1, 1]])

>>> m ** 5
array([[59049, 32768, 1],
       [16807, 7776, 16807]])
```

## Operaciones unarias

Existen multitud de operaciones sobre un único array. A continuación veremos algunas de las más utilizadas en NumPy.

## Funciones universales

Las funciones universales «ufunc» son funciones que operan sobre arrays **elemento a elemento**. Existen muchas funciones universales definidas en Numpy, parte de ellas operan sobre dos arrays y parte sobre un único array.

Un ejemplo de algunas de estas funciones:

```
>>> values
array([[48.32172375, 24.89651106, 77.49724241],
       [77.81874191, 22.54051494, 65.11282444],
       [ 5.54960482, 59.06720303, 62.52817198]])  
  
>>> np.sqrt(values)
array([[6.95138287, 4.98964037, 8.80325181],
       [8.82149318, 4.74768522, 8.06925179],
       [2.35575992, 7.68551905, 7.9074757]])  
  
>>> np.sin(values)
array([[-0.93125201, -0.23403917,  0.86370435],
       [ 0.66019205, -0.52214693,  0.75824777],
       [-0.66953344,  0.58352079, -0.29903488]])  
  
>>> np.ceil(values)
array([[49., 25., 78.],
       [78., 23., 66.],
       [ 6., 60., 63.]])  
  
>>> np.floor(values)
array([[48., 24., 77.],
       [77., 22., 65.],
       [ 5., 59., 62.]])  
  
>>> np.log(values)
array([[3.87788123, 3.21472768, 4.35024235],
       [4.3543823 , 3.11531435, 4.17612153],
       [1.71372672, 4.07867583, 4.13561721]])
```

## Reduciendo el resultado

NumPy nos permite aplicar cualquier función sobre un array **reduciendo** el resultado por alguno de sus ejes. Esto abre una amplia gama de posibilidades.

A modo de ilustración, veamos un par de ejemplos con la suma y el producto:

```
>>> values
array([[8, 2, 7],
       [2, 0, 6],
       [6, 3, 4]])

>>> np.sum(values, axis=0) # suma por columnas
array([16, 5, 17])

>>> np.sum(values, axis=1) # suma por filas
array([17, 8, 13])

>>> np.prod(values, axis=0) # producto por columnas
array([ 96,  0, 168])

>>> np.prod(values, axis=1) # producto por filas
array([112, 0, 72])
```

---

## Ejercicio

Compruebe que, para  $\theta = 2\pi$  (*radianes*) y  $k = 20$  se cumple la siguiente igualdad del *producto infinito de Euler*:

$$\cos\left(\frac{\theta}{2}\right) \cdot \cos\left(\frac{\theta}{4}\right) \cdot \cos\left(\frac{\theta}{8}\right) \cdots = \prod_{i=1}^k \cos\left(\frac{\theta}{2^i}\right) \approx \frac{\sin(\theta)}{\theta}$$

---

## Funciones estadísticas

NumPy proporciona una gran cantidad de funciones estadísticas que pueden ser aplicadas sobre arrays.

Veamos algunas de ellas:

```
>>> dist
array([-6.79006504, -0.01579498, -0.29182173,  0.3298951 , -5.30598975],
      [ 3.10720923, -4.09625791, -7.60624152,  2.3454259 ,  9.23399023],
      [-7.4394269 , -9.68427195,  3.04248586, -5.9843767 ,  1.536578 ],
```

(continué en la próxima página)

(proviene de la página anterior)

```
[ 3.33953286, -8.41584411, -9.530274 , -2.42827813, -7.34843663],
 [ 7.1508544 ,  5.51727548, -3.20216834, -5.00154367, -7.15715252]])
```

```
>>> np.mean(dist)
-2.1877878715377777
```

```
>>> np.std(dist)
5.393254994089515
```

```
>>> np.median(dist)
-3.2021683412383295
```

## Máximos y mínimos

Una de las operaciones más comunes en el manejo de datos es encontrar máximos o mínimos. Para ello, disponemos de las típicas funciones con las ventajas del uso de arrays multidimensionales:

```
>>> values
array([[66, 54, 33, 15, 58],
       [55, 46, 39, 16, 38],
       [73, 75, 79, 25, 83],
       [81, 30, 22, 32, 8],
       [92, 25, 82, 10, 90]])
```

```
>>> np.min(values)
8
```

```
>>> np.min(values, axis=0)
array([55, 25, 22, 10, 8])
>>> np.min(values, axis=1)
array([15, 16, 25, 8, 10])
```

```
>>> np.max(values)
92
```

```
>>> np.max(values, axis=0)
array([92, 75, 82, 32, 90])
>>> np.max(values, axis=1)
array([66, 55, 83, 81, 92])
```

Si lo que interesa es obtener los **índices** de aquellos elementos con valores máximos o mínimos, podemos hacer uso de las funciones `argmax()` y `argmin()` respectivamente.

Veamos un ejemplo donde obtenemos los valores máximos por columnas (mediante sus

índices):

```
>>> values
array([[66, 54, 33, 15, 58],
       [55, 46, 39, 16, 38],
       [73, 75, 79, 25, 83],
       [81, 30, 22, 32, 8],
       [92, 25, 82, 10, 90]])

>>> idx = np.argmax(values, axis=0)

>>> idx
array([4, 2, 4, 3, 4])

>>> values[idx, range(values.shape[1])]
array([92, 75, 82, 32, 90])
```

## Vectorizando funciones

Una de las ventajas de trabajar con arrays numéricos en NumPy es sacar provecho de la optimización que se produce a nivel de la propia estructura de datos. En el caso de que queramos implementar una función propia para realizar una determinada acción, sería deseable seguir aprovechando esa característica.

Veamos un ejemplo en el que queremos realizar el siguiente cálculo entre dos matrices  $A$  y  $B$ :

$$A_{ij} * B_{ij} = \begin{cases} A_{ij} + B_{ij} & , \text{si } A_{ij} > B_{ij} \\ A_{ij} - B_{ij} & , \text{si } A_{ij} < B_{ij} \\ 0 & , \text{e.o.c.} \end{cases}$$

Esta función, definida en Python, quedaría tal que así:

```
>>> def customf(a, b):
...     if a > b:
...         return a + b
...     elif a < b:
...         return a - b
...     else:
...         return 0
... 
```

Las dos matrices de partida tienen 9M de valores aleatorios entre -100 y 100:

```
>>> A = np.random.randint(-100, 100, size=(3000, 3000))
>>> B = np.random.randint(-100, 100, size=(3000, 3000))
```

Una primera aproximación para aplicar esta función a cada elemento de las matrices de entrada sería la siguiente:

```
>>> result = np.zeros_like(A)

>>> %%timeit
... for i in range(A.shape[0]):
...     for j in range(A.shape[1]):
...         result[i, j] = customf(A[i, j], B[i, j])
...
3 s ± 23.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

## Mejorando rendimiento con funciones vectorizadas

Con un pequeño detalle podemos mejorar el rendimiento de la función que hemos diseñado anteriormente. Se trata de decorarla con `np.vectorize` con lo que estamos otorgándole un comportamiento distinto y enfocado al procesamiento de arrays numéricos:

```
>>> @np.vectorize
... def customf(a, b):
...     if a > b:
...         return a + b
...     elif a < b:
...         return a - b
...     else:
...         return 0
...
```

Dado que ahora ya se trata de una **función vectorizada** podemos aplicarla directamente a las matrices de entrada (aprovechamos para medir su tiempo de ejecución):

```
>>> %%timeit customf(A, B)
1.29 s ± 7.33 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Hemos obtenido una mejora de 2.32x con respecto al uso de funciones simples.

---

**Truco:** La mejora de rendimiento se aprecia más claramente a medida que los tamaños de las matrices (arrays) de entrada son mayores.

---

**Consejo:** El uso de *funciones lambda* puede ser muy útil en vectorización: `np.vectorize(lambda a, b: return a + b)`.

---

---

### Ejercicio

1. Cree dos matrices cuadradas de 20x20 con valores aleatorios flotantes uniformes en el intervalo [0, 1000)
  2. Vectorice una función que devuelva la media (elemento a elemento) entre las dos matrices.
  3. Realice la misma operación que en 2) pero usando suma de matrices y división por escalar.
  4. Compute los tiempos de ejecución de 2) y 3)
- 

## 8.2.5 Álgebra lineal

NumPy tiene una sección dedicada al [álgebra lineal](#) cuyas funciones pueden resultar muy interesantes según el contexto en el que estemos trabajando.

### Producto de matrices

Si bien hemos hablado del producto de arrays elemento a elemento, NumPy nos permite hacer la [multiplicación clásica de matrices](#):

```
>>> m1
array([[1, 8, 4],
       [8, 7, 1],
       [1, 3, 8]])

>>> m2
array([[1, 5, 7],
       [9, 4, 2],
       [1, 4, 2]])

>>> np.dot(m1, m2)
array([[77, 53, 31],
       [72, 72, 72],
       [36, 49, 29]])
```

---

### Ejercicio

Compruebe que la matriz  $\begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix}$  satisface la ecuación matricial:  $X^2 - 6X - I = 0$  donde  $I$  es la matriz identidad de orden 2.

---

## Determinante de una matriz

El cálculo del determinante es una operación muy utilizada en álgebra lineal. Lo podemos realizar en NumPy de la siguiente manera:

```
>>> m
array([[4, 1, 6],
       [4, 8, 8],
       [2, 1, 7]])

>>> np.linalg.det(m)
108.0000000000003
```

## Inversa de una matriz

La inversa de una matriz se calcula de la siguiente manera:

```
>>> m
array([[4, 1, 6],
       [4, 8, 8],
       [2, 1, 7]])

>>> m_inv = np.linalg.inv(m)

>>> m_inv
array([[ 0.44444444, -0.00925926, -0.37037037],
       [-0.11111111,  0.14814815, -0.07407407],
       [-0.11111111, -0.01851852,  0.25925926]])
```

Una propiedad de la matriz inversa es que si la multiplicamos por la matriz de partida obtenemos la matriz identidad. Vemos que se cumple  $\mathcal{A} \cdot \mathcal{A}^{-1} = \mathcal{I}$ :

```
>>> np.dot(m, m_inv)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

## Traspuesta de una matriz

La traspuesta de una matriz  $\mathcal{A}$  se denota por:  $(\mathcal{A}^t)_{ij} = \mathcal{A}_{ji}$ ,  $1 \leq i \leq n$ ,  $1 \leq j \leq m$ , pero básicamente consiste en intercambiar filas por columnas.

Aún más fácil es computar la traspuesta de una matriz con NumPy:

```
>>> m
array([[1, 2, 3],
       [4, 5, 6]])

>>> m.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```

---

## Ejercicio

Dadas las matrices:

$$A = \begin{bmatrix} 1 & -2 & 1 \\ 3 & 0 & 1 \end{bmatrix}; B = \begin{bmatrix} 4 & 0 & -1 \\ -2 & 1 & 0 \end{bmatrix}$$

, compruebe que se cumplen las siguientes igualdades:

- $(A + B)^t = A^t + B^t$
  - $(3A)^t = 3A^t$
- 

## Elevar matriz a potencia

En el mundo del álgebra lineal es muy frecuente recurrir a la exponenciación de matrices a través de su producto clásico. En este sentido, NumPy nos proporciona una función para computarlo:

```
>>> m
array([[4, 1, 6],
       [4, 8, 8],
       [2, 1, 7]])

>>> np.linalg.matrix_power(m, 3) # más eficiente que np.dot(m, np.dot(m, m))
array([[ 348,  250,  854],
       [ 848,  816, 2000],
       [ 310,  231,  775]])
```

---

## Ejercicio

Dada la matriz  $A = \begin{bmatrix} 4 & 5 & -1 \\ -3 & -4 & 1 \\ -3 & -4 & 0 \end{bmatrix}$  calcule:  $A^2, A^3, \dots, A^{128}$

¿Nota algo especial en los resultados?

## Sistemas de ecuaciones lineales

NumPy también nos permite resolver sistemas de ecuaciones lineales. Para ello debemos modelar nuestro sistema a través de arrays.

Veamos un ejemplo en el que queremos resolver el siguiente sistema de ecuaciones lineales:

$$\begin{cases} -x_1 + 2x_3 = 1 \\ x_1 - x_2 = -2 \\ x_2 + x_3 = -1 \end{cases} \Rightarrow \begin{pmatrix} 1 & 0 & 2 \\ 1 & -1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ -1 \end{pmatrix} \Rightarrow \mathcal{A}\mathcal{X} = \mathcal{B}$$

Podemos almacenar las matrices de coeficientes  $\mathcal{A}$  y  $\mathcal{B}$  de la siguiente manera:

```
>>> A = np.array([[1, 0, 2], [1, -1, 0], [0, 1, 1]])
>>> B = np.array([1, -2, -1]).reshape(-1, 1)

>>> A
array([[ 1,  0,  2],
       [ 1, -1,  0],
       [ 0,  1,  1]])

>>> B
array([[ 1],
       [-2],
       [-1]])
```

La solución al sistema viene dada por la siguiente función:

```
>>> np.linalg.solve(A, B)
array([-7.,
       -5.,
        4.])
```

La solución del sistema debe ser la misma que si obtenemos  $\mathcal{X} = \mathcal{A}^{-1} \cdot \mathcal{B}$ :

```
>>> np.dot(np.linalg.inv(A), B)
array([-7.,
       -5.,
        4.])
```

## Ejercicio

Resuelva el siguiente sistema de ecuaciones lineales:

$$\begin{cases} 3x + 4y - z = 8 \\ 5x - 2y + z = 4 \\ 2x - 2y + z = 1 \end{cases}$$

---

### 8.3 pandas



pandas es un paquete open-source que nos proporciona una forma sencilla y potente de trabajar con estructuras de datos a través de múltiples herramientas para su análisis.<sup>1</sup>

```
$ pip install pandas
```

La forma más común de importar esta librería es usar el alias pd:

```
>>> import pandas as pd
```

Si bien en *Numpy* la estructura de datos fundamental es el ndarray, en pandas existen dos estructuras de datos sobre las que giran todas las operaciones:

- Series.
- Dataframes.

---

<sup>1</sup> Foto original de portada por Sid Balachandran en Unsplash.

### 8.3.1 Series

Podríamos pensar en una **serie** como un *ndarray* en el que cada valor tiene asignada una etiqueta (índice) y además admite un título (nombre).

#### Creación de una serie

Veamos varios ejemplos de creación de la serie [1, 2, 3].

**Creación de series usando listas:**

```
>>> pd.Series([1, 2, 3])
0    1
1    2
2    3
dtype: int64
```

---

**Nota:** El índice por defecto se crea con números enteros positivos empezando desde 0.

---

**Especificando un índice personalizado (etiqueta a cada valor):**

```
>>> pd.Series(range(1, 4), index=['a', 'b', 'c'])
a    1
b    2
c    3
dtype: int64
```

**Especificando un diccionario con etiquetas y valores:**

```
>>> items = {'a': 1, 'b': 2, 'c': 3}

>>> pd.Series(items)
a    1
b    2
c    3
dtype: int64
```

Todas las series que hemos visto hasta ahora no tienen asignado ningún nombre. Lo podemos hacer usando el parámetro `name` en la creación de la serie:

```
>>> pd.Series([1, 2, 3], name='integers')
0    1
1    2
```

(continué en la próxima página)

(proviene de la página anterior)

```
2    3  
Name: integers, dtype: int64
```

---

## Ejercicio

Cree una serie de pandas con valores enteros en el intervalo [1, 26] y etiquetas 'ABCDEFGHIJKLMNPQRSTUVWXYZ'. Busque una manera programática (no manual) de hacerlo (recuerde el *módulo string*).

---

## Atributos de una serie

Las series en pandas contienen gran cantidad de atributos. A continuación destacaremos algunos de ellos.

Trabajaremos con datos que contienen el número de empleados/as de diferentes empresas tecnológicas<sup>2</sup>:

```
>>> data  
{'Apple': 147000,  
'Samsung': 267937,  
'Google': 135301,  
'Microsoft': 163000,  
'Huawei': 197000,  
'Dell': 158000,  
'Facebook': 58604,  
'Foxconn': 878429,  
'Sony': 109700}  
  
>>> employees = pd.Series(data, name='Tech Employees')
```

### Índice de la serie:

```
>>> employees.index  
Index(['Apple', 'Samsung', 'Google', 'Microsoft', 'Huawei', 'Dell', 'Facebook',  
       'Foxconn', 'Sony'],  
      dtype='object')
```

### Valores de la serie:

```
>>> employees.values  
array([147000, 267937, 135301, 163000, 197000, 158000, 58604, 878429,  
     109700])
```

---

<sup>2</sup> Fuente: [Wikipedia](#).

**Tipo de la serie:**

```
>>> employees.dtype  
dtype('int64')
```

**Nombre de la serie:**

```
>>> employees.name  
'Tech Employees'
```

**Memoria ocupada por la serie:**

```
>>> employees nbytes  
72
```

**Número de registros de la serie:**

```
>>> employees.size  
9
```

## Selección de registros

La selección de los datos se puede realizar desde múltiples aproximaciones. A continuación veremos las posibilidades que nos ofrece pandas para seleccionar/filtrar los registros de una serie.

```
>>> employees  
Apple      147000  
Samsung   267937  
Google     135301  
Microsoft 163000  
Huawei    197000  
Dell      158000  
Facebook   58604  
Foxconn   878429  
Sony      109700  
Name: Tech Employees, dtype: int64
```

## Selección usando indexado numérico

Para acceder a los registros por su posición (índice numérico) basta usar corchetes como ya se ha visto en cualquier secuencia:

```
>>> employees[0]
147000

>>> employees[-1]
109700

>>> employees[2:5]
Google      135301
Microsoft   163000
Huawei      197000
Name: Tech Employees, dtype: int64

>>> employees[1:6:2]
Samsung    267937
Microsoft  163000
Dell       158000
Name: Tech Employees, dtype: int64
```

El atributo `iloc` es un alias (algo más expresivo) que permite realizar las mismas operaciones de indexado (con corchetes) que hemos visto anteriormente:

```
>>> employees.iloc[1:6:2]
Samsung    267937
Microsoft  163000
Dell       158000
Name: Tech Employees, dtype: int64
```

---

**Truco:** Python, y en este caso pandas, se dicen «0-index» porque sus índices (posiciones) comienzan en cero.

---

## Selección usando etiquetas

En el caso de aquellas series que dispongan de un índice con etiquetas, podemos acceder a sus registros utilizando las mismas:

```
>>> employees['Apple'] # equivalente a employees.Apple
147000
```

(continué en la próxima página)

(provien de la página anterior)

```
>>> employees['Apple':'Huawei']
Apple      147000
Samsung    267937
Google     135301
Microsoft   163000
Huawei     197000
Name: Tech Employees, dtype: int64
```

```
>>> employees['Apple':'Huawei':2]
Apple      147000
Google     135301
Huawei     197000
Name: Tech Employees, dtype: int64
```

El atributo `loc` es un alias (algo más expresivo) que permite realizar las mismas operaciones de indexado (con corchetes) que hemos visto anteriormente:

```
>>> employees.loc['Apple':'Huawei':2]
Apple      147000
Google     135301
Huawei     197000
Name: Tech Employees, dtype: int64
```

## Fragmentos de comienzo y fin

A nivel exploratorio, es bastante cómodo acceder a una porción inicial (o final) de los datos que manejamos. Esto se puede hacer de forma muy sencilla con series:

```
>>> employees.head(3)
Apple      147000
Samsung    267937
Google     135301
Name: Tech Employees, dtype: int64

>>> employees.tail(3)
Facebook    58604
Foxconn     878429
Sony        109700
Name: Tech Employees, dtype: int64
```

## Operaciones con series

Si tenemos en cuenta que una serie contiene valores en formato `ndarray` podemos concluir que las *operaciones sobre arrays* son aplicables al caso de las series. Veamos algunos ejemplos de operaciones que podemos aplicar sobre series.

## Operaciones lógicas

Supongamos que queremos filtrar aquellas empresas que tengan más de 200000 trabajadores/as:

```
>>> employees > 200_000
Apple      False
Samsung    True
Google     False
Microsoft  False
Huawei     False
Dell       False
Facebook   False
Foxconn    True
Sony       False
Name: Tech Employees, dtype: bool
```

Hemos obtenido una serie «booleana». Si queremos aplicar esta «máscara», podemos hacerlo con indexado:

```
>>> employees[employees > 200_000] # empresas con más de 200K trabajadores/as
Samsung    267937
Foxconn   878429
Name: Tech Employees, dtype: int64
```

## Ordenación

Ordenación de una serie por sus valores:

```
>>> employees.sort_values()
Facebook    58604
Sony        109700
Google      135301
Apple       147000
Dell        158000
Microsoft   163000
Huawei     197000
```

(continué en la próxima página)

(proviene de la página anterior)

Samsung	267937
Foxconn	878429
Name: Tech Employees, dtype: int64	

**Ordenación de una serie por su índice:**

>>> employees.sort_index()	
Apple	147000
Dell	158000
Facebook	58604
Foxconn	878429
Google	135301
Huawei	197000
Microsoft	163000
Samsung	267937
Sony	109700
Name: Tech Employees, dtype: int64	

**Truco:** Ambos métodos admiten el parámetro `ascending` para indicar si la ordenación es ascendente (`True`) o descendente (`False`); y también admiten el parámetro `inplace` para indicar si se quiere modificar los valores de la serie (`True`) o devolver una nueva ya ordenada (`False`).

**Contando valores**

Si queremos obtener una «tabla de frecuencias» podemos contar los valores que existen en nuestra serie:

>>> marks = pd.Series([5, 5, 3, 6, 5, 2, 8, 3, 8, 7, 6])	
>>> marks.value_counts()	
5	3
3	2
6	2
8	2
2	1
7	1
dtype: int64	

Vinculado con el caso anterior, podemos obtener el número de valores únicos en la serie:

```
>>> marks.nunique()  
6
```

El método `count()` devuelve el número de valores «no nulos» que contiene la serie:

```
>>> marks.count() # en este caso es equivalente a marks.size  
11
```

## Operaciones aritméticas

### Operaciones entre series y escalares

Podemos operar entre series y escalares sin ningún tipo de problema:

```
>>> employees / 1000  
Apple      147.000  
Samsung   267.937  
Google     135.301  
Microsoft 163.000  
Huawei    197.000  
Dell       158.000  
Facebook   58.604  
Foxconn   878.429  
Sony       109.700  
Name: Tech Employees, dtype: float64
```

## Operaciones entre series

Para el caso de operaciones entre series, vamos a ejemplificarlo con las dos siguientes<sup>3</sup>:

```
>>> employees  
Apple      147000  
Samsung   267937  
Google     135301  
Microsoft 163000  
Huawei    197000  
Dell       158000  
Facebook   58604  
Foxconn   878429  
Sony       109700  
Name: Tech Employees, dtype: int64
```

(continué en la próxima página)

<sup>3</sup> Los datos de ingresos («revenues») están en billones (americanos) de dólares.

(provien de la página anterior)

```
>>> revenues
Apple      274515
Samsung    200734
Google     182527
Microsoft  143015
Huawei     129184
Dell       92224
Facebook   85965
Foxconn    181945
Sony       84893
Name: Tech Revenues, dtype: int64
```

Supongamos que queremos calcular la ratio de ingresos por trabajador/a:

```
>>> revenues / employees
Apple      1.867449
Samsung    0.749184
Google     1.349044
Microsoft  0.877393
Huawei     0.655756
Dell       0.583696
Facebook   1.466879
Foxconn    0.207125
Sony       0.773865
dtype: float64
```

---

**Truco:** Tener en cuenta que las operaciones se realizan entre registros que tienen el mismo índice (etiqueta).

---

## Funciones estadísticas

Existen multitud de funciones estadísticas que podemos aplicar a una serie. Dependiendo del tipo de dato con el que estamos trabajando, serán más útiles unas que otras. Veamos dos funciones a modo de ejemplo:

```
>>> employees.mean()
234996.7777777778

>>> employees.std()
248027.7840619765
```

## Máximos y mínimos

El abanico de posibilidades es muy amplio en cuanto a la búsqueda de valores máximos y mínimos en una serie. Veamos lo que nos ofrece pandas a este respecto.

**Obtener valor mínimo/máximo de una serie:**

```
>>> employees.min()  
58604  
>>> employees.max()  
878429
```

**Posición (índice) del valor mínimo/máximo de una serie:**

```
>>> employees.argmin() # employees[6] = 58604  
6  
>>> employees.argmax() # employees[7] = 878429  
7
```

**Etiqueta (índice) del valor mínimo/máximo de una serie:**

```
>>> employees.idxmin()  
'Facebook'  
>>> employees.idxmax()  
'Foxconn'
```

**Obtener los  $n$  valores menores/mayores de una serie:**

```
>>> employees.nsmallest(3)  
Facebook      58604  
Sony          109700  
Google         135301  
Name: Tech Employees, dtype: int64  
  
>>> employees.nlargest(3)  
Foxconn      878429  
Samsung       267937  
Huawei        197000  
Name: Tech Employees, dtype: int64
```

## Exportación de series

Suele ser bastante habitual intercambiar datos en distintos formatos (y aplicaciones). Para ello, pandas nos permite exportar una serie a multitud de formatos. Veamos algunos de ellos:

### Exportación de serie a lista:

```
>>> employees.to_list()
[147000, 267937, 135301, 163000, 197000, 158000, 58604, 878429, 109700]
```

### Exportación de serie a diccionario:

```
>>> employees.to_dict()
{'Apple': 147000,
 'Samsung': 267937,
 'Google': 135301,
 'Microsoft': 163000,
 'Huawei': 197000,
 'Dell': 158000,
 'Facebook': 58604,
 'Foxconn': 878429,
 'Sony': 109700}
```

### Exportación de serie a csv:

```
>>> employees.to_csv()
'Tech Employees\nApple,147000\nSamsung,267937\nGoogle,135301\nMicrosoft,163000\
˓→nHuawei,197000\nDell,158000\nFacebook,58604\nFoxconn,878429\nSony,109700\n'
```

### Exportación de serie a json:

```
>>> employees.to_json()
'{"Apple":147000,"Samsung":267937,"Google":135301,"Microsoft":163000,"Huawei":197000,"Dell":158000,"Facebook":58604,"Foxconn":878429,"Sony":109700}'
```

### Exportación de serie a pandas.DataFrame:

```
>>> employees.to_frame()
Tech Employees
Apple           147000
Samsung         267937
Google          135301
Microsoft       163000
Huawei          197000
Dell            158000
Facebook        58604
Foxconn         878429
Sony            109700
```

Y muchos otros como: `to_clipboard()`, `to_numpy()`, `to_pickle()`, `to_string()`, `to_xarray()`, `to_excel()`, `to_hdf()`, `to_latex()`, `to_markdown()`, `to_period()`, `to_sql()` o `to_timestamp()`.

### 8.3.2 DataFrames

Un DataFrame es una estructura tabular compuesta por series. Se trata del tipo de datos fundamental en pandas y sobre el que giran la mayoría de operaciones que podemos realizar.

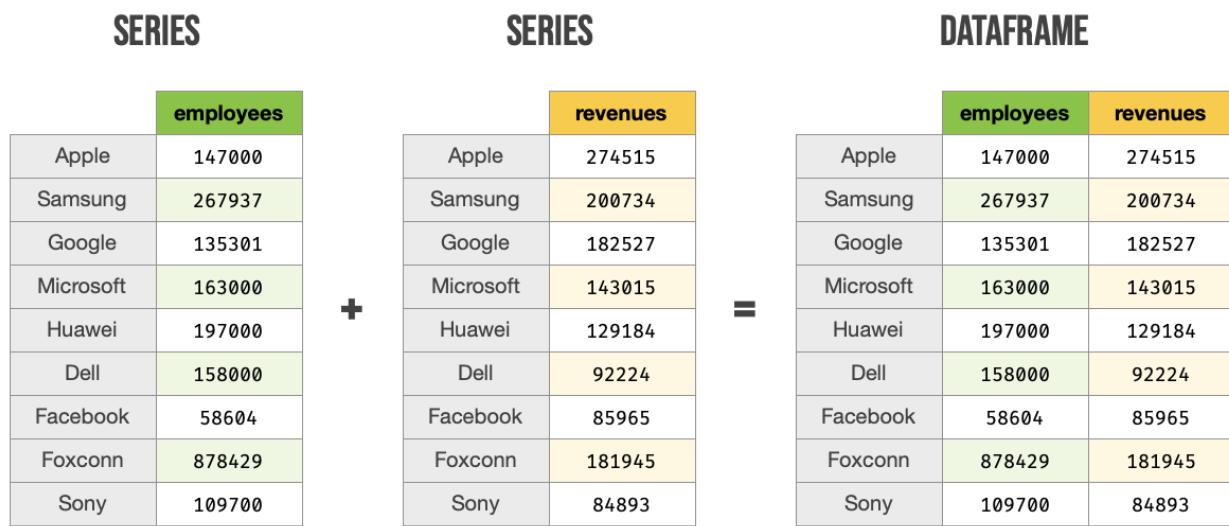


Figura 13: Estructura de un DataFrame a partir de Series

### Creación de un DataFrame

Existen múltiples formas de crear un DataFrame en pandas. Veamos algunas de ellas.

#### DataFrame desde diccionario de listas

Cada elemento del diccionario se convierte en una **columna**, donde su clave es el nombre y sus valores se despliegan en «vertical»:

```
>>> data = {'A': [1, 2, 3], 'B': [4, 5, 6]}

>>> pd.DataFrame(data)
   A   B
0  1   4
1  2   5
2  3   6
```

## DataFrame desde lista de diccionarios

Cada elemento de la lista se convierte en una **fila**. Las claves de cada diccionario serán los nombres de las columnas y sus valores se despliegan en «horizontal»:

```
>>> data = [{‘A’: 1, ‘B’: 2, ‘C’: 3}, {‘A’: 4, ‘B’: 5, ‘C’: 6}]

>>> pd.DataFrame(data)
   A   B   C
0  1   2   3
1  4   5   6
```

## DataFrame desde lista de listas

Cada elemento de la lista se convierte en una **fila** y sus valores se despliegan en «horizontal». Los nombres de las columnas deben pasarse como parámetro opcional:

```
>>> data = [[1, 2], [3, 4], [5, 6]]

>>> pd.DataFrame(data, columns=[‘A’, ‘B’])
   A   B
0  1   2
1  3   4
2  5   6
```

## DataFrame desde series

```
>>> employees
Apple        147000
Samsung      267937
Google        135301
Microsoft     163000
Huawei        197000
Dell          158000
Facebook      58604
Foxconn       878429
Sony          109700
Name: Tech Employees, dtype: int64

>>> revenues
Apple        274515
Samsung      200734
```

(continué en la próxima página)

(provienec de la página anterior)

```
Google      182527
Microsoft   143015
Huawei     129184
Dell        92224
Facebook    85965
Foxconn    181945
Sony       84893
Name: Tech Revenues, dtype: int64
```

```
>>> pd.DataFrame({'employees': employees, 'revenues': revenues})
   employees  revenues
Apple      147000    274515
Samsung    267937    200734
Google     135301    182527
Microsoft   163000    143015
Huawei     197000    129184
Dell       158000    92224
Facebook    58604     85965
Foxconn    878429    181945
Sony       109700    84893
```

---

### Ejercicio

Cree el siguiente DataFrame en Pandas<sup>5</sup>:

---

<sup>5</sup> Datos extraídos de [Wikipedia](#).

Island	Population	Area	Province
Gran Canaria	855521	1560.1	LPGC
Tenerife	928604	2034.38	SCTF
La Palma	83458	708.32	SCTF
Lanzarote	155812	845.94	LPGC
La Gomera	21678	369.76	SCTF
El Hierro	11147	278.71	SCTF
Fuerteventura	119732	1659	LPGC

La superficie (*Area*) está expresada en  $\text{km}^2$  y las provincias corresponden con LPGC: Las Palmas de Gran Canaria y SCTF: Santa Cruz de Tenerife.

---

**Importante:** Nos referiremos a este DataFrame como `democan` de ahora en adelante.

---

## Gestión del índice

Cuando creamos un DataFrame, pandas autocompleta el índice con un valor entero autoincremental comenzando desde cero:

```
>>> pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
```

	A	B
0	1	3
1	2	4

Si queremos convertir alguna columna en el índice de la tabla, podemos hacerlo así:

```
>>> stats = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})

>>> stats.set_index('A') # columna A como índice
      B
```

(continué en la próxima página)

(provine de la página anterior)

A
1 3
2 4

Podemos añadir un parámetro (en la creación) para especificar el índice que queremos incluir:

```
>>> pd.DataFrame({'A': [1, 2], 'B': [3, 4]}, index=['R1', 'R2'])
```

	A	B
R1	1	3
R2	2	4

En aquellos DataFrames que disponen de un índice etiquetado, es posible resetearlo:

```
>>> pd.DataFrame({'A': [1, 2], 'B': [3, 4]}, index=['R1', 'R2']).reset_index()
```

index	A	B	
0	R1	1	3
1	R2	2	4

---

## Ejercicio

Convierta la columna *Island* en el índice de democan. El DataFrame debería de quedar así:

```
>>> df
```

Island	Population	Area	Province
Gran Canaria	855521	1560.10	LPGC
Tenerife	928604	2034.38	SCTF
La Palma	83458	708.32	SCTF
Lanzarote	155812	845.94	LPGC
La Gomera	21678	369.76	SCTF
El Hierro	11147	278.71	SCTF
Fuerteventura	119732	1659.00	LPGC

---

## Lectura de fuentes externas

Lo más habitual cuando se trabaja en ciencia de datos es tener la información en distintas fuentes auxiliares: bases de datos, ficheros, llamadas remotas a APIs, etc. pandas nos ofrece una variedad enorme de funciones para cargar datos desde, prácticamente, cualquier origen.

Tabla 2: Funciones para lectura de datos en pandas

Función	Explicación
read_pickle	Lectura de datos en formato pickle (Python)
read_table	Lectura de ficheros con delimitadores
read_csv	Lectura de ficheros .csv
read_fwf	Lectura de tablas con líneas de ancho fijo
read_clipboard	Lectura de texto del portapapeles
read_excel	Lectura de ficheros excel
read_json	Lectura de ficheros json
read_html	Lectura de tablas HTML
read_html	Lectura de tablas HTML
read_xml	Lectura de documentos XML
read_hdf	Lectura de objetos pandas almacenados en fichero
read_feather	Lectura de objetos en formato «feather»
read_parquet	Lectura de objetos en formato «parquet»
read_orc	Lectura de objetos en formato ORC
read_sas	Lectura de ficheros SAS
read_spss	Lectura de ficheros SPSS
read_sql_table	Lectura de tabla SQL
read_sql_query	Lectura de una consulta SQL
read_sql	Wrapper para <code>read_sql_table</code> y <code>read_sql_query</code>
read_gbq	Lectura de datos desde Google BigQuery
read_stata	Lectura de ficheros Stata

**Nota:** Todas estas funciones tienen su equivalente para escribir datos en los distintos formatos. En vez de `read_` habría que usar el prefijo `to_`. Por ejemplo: `.to_csv()`, `.to_json()` o `.to_sql()`

A modo de ilustración, vamos a leer el contenido del fichero `tech.csv` que contiene la lista de las mayores empresas tecnológicas por ingresos totales<sup>2</sup>.

Este fichero está delimitado por tabuladores, con lo que especificaremos esta circunstancia al usar la función correspondiente. Igualmente, vamos a indicar que se use la primera columna `Company` como índice del DataFrame:

```
>>> df = pd.read_csv('tech.csv', delimiter='\t', index_col='Company')

>>> df
      Revenue Employees          City        Country
Company

```

(continué en la próxima página)

(proviene de la página anterior)

Apple	274515	147000	California	United States
Samsung Electronics	200734	267937	Suwon	South Korea
Alphabet	182527	135301	California	United States
Foxconn	181945	878429	New Taipei City	Taiwan
Microsoft	143015	163000	Washington	United States
Huawei	129184	197000	Shenzhen	China
Dell Technologies	92224	158000	Texas	United States
Facebook	85965	58604	California	United States
Sony	84893	109700	Tokyo	Japan
Hitachi	82345	350864	Tokyo	Japan
Intel	77867	110600	California	United States
IBM	73620	364800	New York	United States
Tencent	69864	85858	Shenzhen	China
Panasonic	63191	243540	Osaka	Japan
Lenovo	60742	71500	Hong Kong	China
HP Inc.	56639	53000	California	United States
LG Electronics	53625	75000	Seoul	South Korea

---

**Truco:** Se suele usar df como nombre para las variables tipo DataFrame.

---

## Ejercicio

Cargue el conjunto de datos democan desde democan.csv en un DataFrame df indicando que la columna *Island* es el índice.

---

## Características de un DataFrame

### Visualización de los datos

Para «echar un vistazo» a los datos, existen dos funciones muy recurridas:

	Revenue	Employees	City	Country
Company				
Apple	274515	147000	California	United States
Samsung Electronics	200734	267937	Suwon	South Korea
Alphabet	182527	135301	California	United States
Foxconn	181945	878429	New Taipei City	Taiwan
Microsoft	143015	163000	Washington	United States

(continué en la próxima página)

(provine de la página anterior)

	Revenue	Employees	City	Country
Company				
Tencent	69864	85858	Shenzhen	China
Panasonic	63191	243540	Osaka	Japan
Lenovo	60742	71500	Hong Kong	China
HP Inc.	56639	53000	California	United States
LG Electronics	53625	75000	Seoul	South Korea

**Truco:** Estas funciones admiten como parámetro el número de registros a visualizar.

## Información sobre los datos

Pandas ofrece algunas funciones que proporcionan un cierto «resumen» de los datos a nivel descriptivo. Veamos algunas de ellas.

Información sobre columnas:

```
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
Index: 17 entries, Apple to LG Electronics
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype  
---  --          --          --      
0   Revenue     17 non-null    int64  
1   Employees   17 non-null    int64  
2   City        17 non-null    object  
3   Country     17 non-null    object  
dtypes: int64(2), object(2)
memory usage: 680.0+ bytes
```

Descripción de las variables numéricas:

```
>>> df.describe()
           Revenue      Employees
count    17.000000    17.000000
mean    112523.235294  204125.470588
std     63236.957691   198345.912495
min     53625.000000   53000.000000
25%    69864.000000   85858.000000
50%    84893.000000   147000.000000
75%    143015.000000  243540.000000
max    274515.000000  878429.000000
```

Uso de memoria:

```
>>> df.memory_usage()  
Index      692  
Revenue    136  
Employees  136  
City       136  
Country    136  
dtype: int64
```

## Atributos de un DataFrame

Tamaños y dimensiones:

```
>>> df.shape # filas por columnas  
(17, 4)  
  
>>> df.size # número total de datos  
68  
  
>>> df.ndim # número de dimensiones  
2
```

Índice, columnas y valores:

```
>>> df.index  
Index(['Apple', 'Samsung Electronics', 'Alphabet', 'Foxconn', 'Microsoft',  
       'Huawei', 'Dell Technologies', 'Facebook', 'Sony', 'Hitachi', 'Intel',  
       'IBM', 'Tencent', 'Panasonic', 'Lenovo', 'HP Inc.', 'LG Electronics'],  
      dtype='object', name='Company')  
  
>>> df.columns  
Index(['Revenue', 'Employees', 'City', 'Country'], dtype='object')  
  
>>> df.values  
array([[274515, 147000, 'California', 'United States'],  
       [200734, 267937, 'Suwon', 'South Korea'],  
       [182527, 135301, 'California', 'United States'],  
       [181945, 878429, 'New Taipei City', 'Taiwan'],  
       [143015, 163000, 'Washington', 'United States'],  
       [129184, 197000, 'Shenzhen', 'China'],  
       [92224, 158000, 'Texas', 'United States'],  
       [85965, 58604, 'California', 'United States'],  
       [84893, 109700, 'Tokyo', 'Japan'],  
       [82345, 350864, 'Tokyo', 'Japan']],
```

(continué en la próxima página)

(provine de la página anterior)

```
[77867, 110600, 'California', 'United States'],
[73620, 364800, 'New York', 'United States'],
[69864, 85858, 'Shenzhen', 'China'],
[63191, 243540, 'Osaka', 'Japan'],
[60742, 71500, 'Hong Kong', 'China'],
[56639, 53000, 'California', 'United States'],
[53625, 75000, 'Seoul', 'South Korea']], dtype=object)
```

## Acceso a un DataFrame

Es fundamental conocer la estructura de un DataFrame para su adecuado manejo:

		Revenue	Employees
0	Apple	274515	147000
1	Samsung	200734	267937
2	Alphabet	182527	135301
3	Microsoft	143015	163000
4	Huawei	129184	197000
5	Dell	92224	158000
6	Facebook	85965	58604
7	Foxconn	181945	878429
8	Sony	84893	109700

Figura 14: Componentes de un DataFrame

Para todos los ejemplos subsiguientes continuamos utilizando el conjunto de datos de empresas tecnológicas cargado previamente:

```
>>> df
```

Company	Revenue	Employees	City	Country
Apple	274515	147000	California	United States
Samsung Electronics	200734	267937	Suwon	South Korea
Alphabet	182527	135301	California	United States
Foxconn	181945	878429	New Taipei City	Taiwan
Microsoft	143015	163000	Washington	United States
Huawei	129184	197000	Shenzhen	China
Dell Technologies	92224	158000	Texas	United States
Facebook	85965	58604	California	United States
Sony	84893	109700	Tokyo	Japan
Hitachi	82345	350864	Tokyo	Japan
Intel	77867	110600	California	United States
IBM	73620	364800	New York	United States
Tencent	69864	85858	Shenzhen	China
Panasonic	63191	243540	Osaka	Japan
Lenovo	60742	71500	Hong Kong	China
HP Inc.	56639	53000	California	United States
LG Electronics	53625	75000	Seoul	South Korea

## Acceso a filas

Si queremos acceder a las filas de un conjunto de datos **mediante la posición (índice numérico)** del registro usamos el atributo `iloc`:

```
>>> df.iloc[0]
```

```
Revenue      274515
Employees    147000
City         California
Country      United States
Name: Apple, dtype: object
```

```
>>> df.iloc[-1]
```

```
Revenue      53625
Employees    75000
City         Seoul
Country      South Korea
Name: LG Electronics, dtype: object
```

```
>>> df.iloc[3:5]
```

```
          Revenue Employees        City      Country
Company
Foxconn   181945     878429  New Taipei City      Taiwan
Microsoft 143015     163000      Washington  United States
```

---

**Nota:** El acceso a un registro individual nos devuelve una serie.

---

Si queremos acceder a las filas de un conjunto de datos **mediante la etiqueta del registro** usamos el atributo `loc`:

```
>>> df.loc['Apple']
Revenue           274515
Employees        147000
City             California
Country          United States
Name: Apple, dtype: object

>>> df.loc['IBM']
Revenue           73620
Employees        364800
City             New York
Country          United States
Name: IBM, dtype: object

>>> df.loc['Sony':'Intel']
      Revenue Employees      City      Country
Company
Sony       84893     109700    Tokyo      Japan
Hitachi    82345     350864    Tokyo      Japan
Intel      77867     110600  California  United States
```

---

**Nota:** El acceso a un registro individual nos devuelve una serie.

---

## Acceso a columnas

El acceso a columnas se realiza directamente utilizando corchetes, como si fuera un diccionario:

```
>>> df['Revenue'] # equivalente a df.Revenue
Company
Apple           274515
Samsung Electronics 200734
Alphabet        182527
Foxconn          181945
Microsoft        143015
Huawei            129184
Dell Technologies 92224
```

(continué en la próxima página)

(proviene de la página anterior)

Facebook	85965
Sony	84893
Hitachi	82345
Intel	77867
IBM	73620
Tencent	69864
Panasonic	63191
Lenovo	60742
HP Inc.	56639
LG Electronics	53625
Name: Revenue, dtype: int64	

---

**Nota:** El acceso a una columna individual nos devuelve una serie.

---

Se pueden seleccionar varias columnas a la vez pasando una lista:

>>> df[['Employees', 'City']].head()		
	Employees	City
Company		
Apple	147000	California
Samsung Electronics	267937	Suwon
Alphabet	135301	California
Foxconn	878429	New Taipei City
Microsoft	163000	Washington

## Acceso a filas y columnas

Si mezclamos los dos accesos anteriores podemos seleccionar datos de forma muy precisa. Como siempre, partimos del «dataset» de empresas tecnológicas:

>>> df.head()				
	Revenue	Employees	City	Country
Company				
Apple	274515	147000	California	United States
Samsung Electronics	200734	267937	Suwon	South Korea
Alphabet	182527	135301	California	United States
Foxconn	181945	878429	New Taipei City	Taiwan
Microsoft	143015	163000	Washington	United States

Acceso al **primer valor del número de empleados/as**. Formas equivalentes de hacerlo:

```
>>> df.iloc[0, 0]
274515

>>> df.loc['Apple', 'Revenue']
274515
```

Acceso a ciudad y país de las empresas Sony, Panasonic y Lenovo:

```
>>> df.loc[['Sony', 'Panasonic', 'Lenovo'], ['City', 'Country']]
          City Country
Company
Sony           Tokyo   Japan
Panasonic      Osaka   Japan
Lenovo         Hong Kong China
```

Acceso a la **última columna** del DataFrame:

```
>>> df.iloc[:, -1]
Company
Apple           United States
Samsung Electronics    South Korea
Alphabet        United States
Foxconn          Taiwan
Microsoft        United States
Huawei            China
Dell Technologies    United States
Facebook          United States
Sony               Japan
Hitachi            Japan
Intel              United States
IBM                United States
Tencent            China
Panasonic          Japan
Lenovo              China
HP Inc.            United States
LG Electronics       South Korea
Name: Country, dtype: object
```

Acceso a las **tres últimas filas (empresas)** y a las **dos primeras columnas**:

```
>>> df.iloc[-3:, :2]
          Revenue Employees
Company
Lenovo      60742     71500
HP Inc.     56639     53000
LG Electronics 53625    75000
```

---

**Truco:** Es posible usar «slicing» (troceado) en el acceso a registros y columnas.

---

## Selección condicional

Es posible aplicar ciertas condiciones en la selección de los datos para obtener el subconjunto que estemos buscando. Veremos distintas aproximaciones a esta técnica.

Supongamos que queremos seleccionar aquellas **empresas con base en Estados Unidos**. Si aplicamos la condición sobre la columna obtendremos una serie de tipo «booleano» en la que se indica para qué registros se cumple la condición (incluyendo el índice):

```
>>> df['Country'] == 'United States'  
Company  
Apple           True  
Samsung Electronics False  
Alphabet        True  
Foxconn          False  
Microsoft        True  
Huawei           False  
Dell Technologies True  
Facebook         True  
Sony              False  
Hitachi           False  
Intel             True  
IBM               True  
Tencent            False  
Panasonic         False  
Lenovo             False  
HP Inc.            True  
LG Electronics     False  
Name: Country, dtype: bool
```

Si aplicamos esta «máscara» al conjunto original de datos, obtendremos las empresas que estamos buscando:

```
>>> df[df['Country'] == 'United States']  
      Revenue Employees      City       Country  
Company  
Apple           274515    147000 California United States  
Alphabet        182527    135301 California United States  
Microsoft       143015    163000 Washington United States  
Dell Technologies 92224    158000 Texas     United States  
Facebook        85965     58604 California United States  
Intel            77867    110600 California United States
```

(continué en la próxima página)

(provine de la página anterior)

IBM	73620	364800	New York	United States
HP Inc.	56639	53000	California	United States

También es posible aplicar condiciones compuestas. Supongamos que necesitamos seleccionar aquellas **empresas con más de 100000 billones de dólares de ingresos y más de 100000 empleados/as**:

>>> revenue_condition = df['Revenue'] > 100_000				
>>> employees_condition = df['Employees'] > 100_000				
>>> df[revenue_condition & employees_condition]				
Revenue Employees City Country				
Company				
Apple	274515	147000	California	United States
Samsung Electronics	200734	267937	Suwon	South Korea
Alphabet	182527	135301	California	United States
Foxconn	181945	878429	New Taipei City	Taiwan
Microsoft	143015	163000	Washington	United States
Huawei	129184	197000	Shenzhen	China

Los operadores lógicos que se pueden utilizar para combinar condiciones de selección son los siguientes:

Operador	Significado
	«or» lógico
&	«and» lógico
~	«not» lógico
^	«xor» lógico

Imaginemos ahora que estamos buscando aquellas **empresas establecidas en California o Tokyo**. Una posible aproximación sería utilizar una condición compuesta, pero existe la función `isin()` que nos permite comprobar si un valor está dentro de una lista de opciones:

>>> mask = df['City'].isin(['California', 'Tokyo'])				
>>> df[mask]				
Revenue Employees City Country				
Company				
Apple	274515	147000	California	United States
Alphabet	182527	135301	California	United States
Facebook	85965	58604	California	United States
Sony	84893	109700	Tokyo	Japan
Hitachi	82345	350864	Tokyo	Japan
Intel	77867	110600	California	United States

(continué en la próxima página)

(provine de la página anterior)

HP Inc.	56639	53000	California	United States
---------	-------	-------	------------	---------------

---

## Ejercicio

Obtenga los siguientes subconjuntos del «dataset» democan:

```
# Use .loc
Population      Area Province
Island
El Hierro      11147  278.71      SCTF
La Gomera       21678  369.76      SCTF
```

```
# Use .loc
Island
Gran Canaria    LPGC
Tenerife        SCTF
La Palma         SCTF
Lanzarote        LPGC
La Gomera        SCTF
El Hierro        SCTF
Fuerteventura   LPGC
Name: Province, dtype: object
```

```
# Use .iloc
Island
Gran Canaria    1560.10
La Palma         708.32
La Gomera        369.76
Fuerteventura   1659.00
Name: Area, dtype: float64
```

```
# Islas con más de 1000 km2 de extensión
Population      Area Province
Island
Gran Canaria    855521  1560.10      LPGC
Tenerife        928604  2034.38      SCTF
Fuerteventura   119732  1659.00      LPGC
```

## Modificación de un DataFrame

### Modificando valores existentes

Partiendo del *acceso a los datos* que ya hemos visto, podemos asignar valores sin mayor dificultad.

Pero antes de modificar el DataFrame original, vamos a hacer una copia del mismo:

```
>>> df_mod = df.copy()

>>> df_mod.equals(df) # comprueba que todos los valores del DataFrame son iguales
True
```

Supongamos que hemos cometido un **error en el número de empleados/as de Apple** y queremos corregirlo:

```
>>> df_mod.head(1)
      Revenue Employees      City      Country
Company
Apple      274515    147000  California  United States

>>> df_mod.loc['Apple', 'Employees'] = 137000

>>> df_mod.head(1)
      Revenue Employees      City      Country
Company
Apple      274616    137000  California  United States
```

Supongamos que no se había contemplado una **subida del 20% en los ingresos** y queremos reflejarla:

```
>>> df_mod['Revenue'] *= 1.20

>>> df_mod['Revenue'].head()
Company
Apple            329418.0
Samsung Electronics 240880.8
Alphabet          219032.4
Foxconn            218334.0
Microsoft          171618.0
Name: Revenue, dtype: float64
```

Supongamos que todas las empresas tecnológicas **mueven su sede a Vigo (España)** y queremos reflejarlo:

```
>>> df_mod['City'] = 'Vigo'  
>>> df_mod['Country'] = 'Spain'  
  
>>> df_mod.head()  
          Revenue Employees City Country  
Company  
Apple           329418.0     137000  Vigo   Spain  
Samsung Electronics 240880.8     267937  Vigo   Spain  
Alphabet        219032.4     135301  Vigo   Spain  
Foxconn          218334.0     878429  Vigo   Spain  
Microsoft         171618.0     163000  Vigo   Spain
```

---

**Nota:** En este último ejemplo se produce un «broadcast» o difusión del valor escalar a todos los registros del «dataset».

---

### Reemplazo de valores

Hay una función muy importante en lo relativo a la modificación de valores. Se trata de `replace()` y admite una amplia variedad de parámetros. Se puede usar tanto para tipos numéricos como textuales.

Uno de los usos más habituales es la recodificación. Supongamos que queremos **recodificar los países en ISO3166 Alpha-3** para el DataFrame de empresas tecnológicas:

```
>>> iso3166 = {  
    'United States': 'USA',  
    'South Korea': 'KOR',  
    'Taiwan': 'TWN',  
    'China': 'CHN',  
    'Japan': 'JPN'  
}  
  
>>> df.replace(iso3166)  
          Revenue Employees          City Country  
Company  
Apple           274515     147000  California   USA  
Samsung Electronics 200734     267937      Suwon   KOR  
Alphabet        182527     135301  California   USA  
Foxconn          181945     878429  New Taipei City   TWN  
Microsoft         143015     163000  Washington   USA  
Huawei            129184     197000    Shenzhen   CHN  
Dell Technologies  92224      158000       Texas   USA  
Facebook          85965      58604  California   USA
```

(continué en la próxima página)

(proviene de la página anterior)

Sony	84893	109700	Tokyo	JPN
Hitachi	82345	350864	Tokyo	JPN
Intel	77867	110600	California	USA
IBM	73620	364800	New York	USA
Tencent	69864	85858	Shenzhen	CHN
Panasonic	63191	243540	Osaka	JPN
Lenovo	60742	71500	Hong Kong	CHN
HP Inc.	56639	53000	California	USA
LG Electronics	53625	75000	Seoul	KOR

## Ejercicio

Recodifique la columna *Province* del «dataset» `democan` de tal manera que aparezcan las provincias con el texto completo: *Santa Cruz de Tenerife* y *Las Palmas de Gran Canaria*.

## Insertando y borrando filas

Podemos insertar datos en un DataFrame como filas o como columnas.

Supongamos que queremos incluir una **nueva empresa Cisco**<sup>4</sup>:

```
>>> cisco = pd.Series(data=[51_904, 75_900, 'California', 'United States'],
...                     index=df_mod.columns, name='Cisco')

>>> cisco
Revenue           51904
Employees         75900
City              California
Country           United States
Name: Cisco, dtype: object

>>> df_mod = df_mod.append(cisco)

>>> df_mod.tail(3)
          Revenue Employees      City        Country
Company
HP Inc.       67966.8     53000      Vigo      Spain
LG Electronics 64350.0     75000      Vigo      Spain
Cisco         51904.0     75900  California  United States
```

**Truco:** El método `append()` devuelve un nuevo DataFrame con los datos añadidos. Es por

<sup>4</sup> Datos del año 2020 según Wikipedia.

eso que si queremos consolidar los cambios, debemos realizar una asignación.

---

Imaginemos ahora que **Facebook**, **Tencent** e **Hitachi** caen en bancarrota y debemos eliminarlas de nuestro conjunto de datos:

```
>>> df_mod = df_mod.drop(labels=['Facebook', 'Tencent', 'Hitachi'])  
  
>>> df_mod.index # ya no aparecen en el índice  
Index(['Apple', 'Samsung Electronics', 'Alphabet', 'Foxconn', 'Microsoft',  
       'Huawei', 'Dell Technologies', 'Sony', 'Intel', 'IBM', 'Panasonic',  
       'Lenovo', 'HP Inc.', 'LG Electronics', 'Cisco'],  
      dtype='object', name='Company')
```

### Insertando y borrando columnas

Insertar una columna en un DataFrame es equivalente a *añadir una clave en un diccionario*.

Supongamos que queremos **añadir una columna «Expenses» (gastos)**. No manejamos esta información, así que, a modo de ejemplo, utilizaremos unos valores aleatorios:

```
>>> expenses = np.random.randint(50_000, 300_000, size=15)  
  
>>> expenses  
array([139655,  97509, 220777, 260609, 121145, 112338,  72815, 159843,  
       205695,  97672,  89614, 260028, 171650, 152049,  57006])  
  
>>> df_mod['Expenses'] = expenses  
  
>>> df_mod.head()  
          Revenue Employees City Country Expenses  
Company  
Apple           329418.0     137000  Vigo  Spain    139655  
Samsung Electronics 240880.8     267937  Vigo  Spain    97509  
Alphabet        219032.4     135301  Vigo  Spain   220777  
Foxconn          218334.0     878429  Vigo  Spain   260609  
Microsoft         171618.0     163000  Vigo  Spain   121145
```

---

**Truco:** También existe la función `insert()` que nos permite insertar una columna en una posición determinada.

---

En el caso de que no nos haga falta una columna podemos borrarla fácilmente. Una opción sería utilizar la función «built-in» `del()`, pero seguiremos con el uso de funciones propias de pandas. Imaginemos que queremos **eliminar la columna Expenses**:

```

>>> df_mod.columns
Index(['Revenue', 'Employees', 'City', 'Country', 'Expenses'], dtype='object')

>>> df_mod = df_mod.drop(labels='Expenses', axis=1)

>>> df_mod.columns
Index(['Revenue', 'Employees', 'City', 'Country'], dtype='object')

```

**Truco:** Recordar que el parámetro `axis` indica en qué «dirección» estamos trabajando. Véase [el acceso a un DataFrame](#).

## El parámetro `inplace`

Muchas de las funciones de pandas se dicen «no destructivas» en el sentido de que no modifican el conjunto de datos original, sino que devuelven uno nuevo con las modificaciones realizadas. Pero este comportamiento se puede modificar utilizando el parámetro `inplace`.

Veamos un ejemplo con el borrado de columnas:

```

>>> df_mod.head()
      Revenue Employees City Country
Company
Apple           329418.0    137000   Vigo   Spain
Samsung Electronics  240880.8    267937   Vigo   Spain
Alphabet         219032.4    135301   Vigo   Spain
Foxconn          218334.0    878429   Vigo   Spain
Microsoft        171618.0    163000   Vigo   Spain

>>> df_mod.drop(labels=['City', 'Country'], axis=1, inplace=True)

>>> df_mod.head()
      Revenue Employees
Company
Apple           329418.0    137000
Samsung Electronics  240880.8    267937
Alphabet         219032.4    135301
Foxconn          218334.0    878429
Microsoft        171618.0    163000

```

---

## Ejercicio

Añada una nueva columna *Density* a `democan` de tal manera que represente la densidad de

población de cada isla del archipiélago canario.

---

### Otras operaciones con un DataFrame

#### Manejando cadenas de texto

A menudo solemos trabajar con datos que incluyen información textual. Pandas también nos ofrece herramientas para cubrir estos casos.

De hecho, simplemente debemos utilizar el manejador `str` y tendremos a disposición la gran mayoría de funciones vistas en la sección de [cadenas de texto](#).

Veamos un primer ejemplo en el que **pasamos a mayúsculas las ciudades en las que se localizan las empresas** tecnológicas:

```
>>> df['City'].str.upper()
Company
Apple           CALIFORNIA
Samsung Electronics   SUWON
Alphabet         CALIFORNIA
Foxconn          NEW TAIPEI CITY
Microsoft        WASHINGTON
Huawei           SHENZHEN
Dell Technologies  TEXAS
Facebook          CALIFORNIA
Sony              TOKYO
Hitachi           TOKYO
Intel             CALIFORNIA
IBM               NEW YORK
Tencent           SHENZHEN
Panasonic         OSAKA
Lenovo            HONG KONG
HP Inc.           CALIFORNIA
LG Electronics    SEOUL
Name: City, dtype: object
```

Otro supuesto sería el de **sustituir espacios por subguiones en los países de las empresas**:

```
>>> df['Country'].str.replace(' ', '_')
Company
Apple           United_States
Samsung Electronics   South_Korea
Alphabet         United_States
Foxconn          Taiwan
```

(continué en la próxima página)

(provine de la página anterior)

Microsoft	United_States
Huawei	China
Dell Technologies	United_States
Facebook	United_States
Sony	Japan
Hitachi	Japan
Intel	United_States
IBM	United_States
Tencent	China
Panasonic	Japan
Lenovo	China
HP Inc.	United_States
LG Electronics	South_Korea
Name: Country, dtype: object	

Incluso podemos recurrir a expresiones regulares. Supongamos que queremos **filtrar las empresas y quedarnos con las que comienzan por vocal**:

```
>>> mask = df.index.str.match(r'^[aeiou]', flags=re.IGNORECASE)

>>> df[mask]
      Revenue Employees        City          Country
Company
Apple       274515     147000  California  United States
Alphabet    182527     135301  California  United States
Intel        77867     110600  California  United States
IBM          73620     364800   New York  United States
```

---

**Nota:** Dado que el nombre de la empresa está actuando como índice del «dataset», hemos aplicado la búsqueda sobre `.index`.

---

## Usando funciones estadísticas

Vamos a aplicar las funciones estadísticas que proporciona pandas sobre la columna **Revenue** de nuestro «dataset», aunque podríamos hacerlo sobre todas aquellas variables numéricas susceptibles:

```
>>> df['Revenue']
Company
Apple           274515
Samsung Electronics  200734
Alphabet        182527
```

(continué en la próxima página)

(proviene de la página anterior)

Foxconn	181945
Microsoft	143015
Huawei	129184
Dell Technologies	92224
Facebook	85965
Sony	84893
Hitachi	82345
Intel	77867
IBM	73620
Tencent	69864
Panasonic	63191
Lenovo	60742
HP Inc.	56639
LG Electronics	53625
Name: Revenue, dtype: int64	

Tabla 3: Funciones estadísticas en pandas

Función	Resultado	Descripción
df['Revenue'].count()	17	Número de observaciones no nulas
df['Revenue'].sum()	1912895	Suma de los valores
df['Revenue'].mean()	112523.23	Media de los valores
df['Revenue'].mad()	51385.95	Desviación absoluta media
df['Revenue'].median()	84893.0	Mediana de los valores
df['Revenue'].min()	53625	Mínimo
df['Revenue'].max()	274515	Máximo
df['Revenue'].mode()	Múltiples	Moda
df['Revenue'].abs()	Múltiples	Valor absoluto
df['Revenue'].valores		
df['Revenue'].prod()	6758064771941226376	Producto de los valores
df['Revenue'].std()	63236.95	Desviación típica
df['Revenue'].var()	39989128	Varianza
df['Revenue'].sem()	15337.21	Error típico de la media
df['Revenue'].skew()	1.33	Asimetría
df['Revenue'].kurt()	1.13	Apuntamiento
df['Revenue'].quantile()	84893.0	Cuantiles (por defecto 50%)
df['Revenue'].cumsum()	Múltiples	Suma acumulativa
valores		
df['Revenue'].cumprod()	Múltiples	Producto acumulativo
valores		

continué en la próxima página

Tabla 3 – proviene de la página anterior

Función	Resultado	Descripción
df['Revenue'].cummax()	Múltiples valores	Máximo acumulativo
df['Revenue'].cummin()	Múltiples valores	Mínimo acumulativo

## Ejercicio

Partiendo del conjunto de datos democan, obtenga aquellas islas cuya población está por encima de la media del archipiélago canario.

Resultado esperado: ['Gran Canaria', 'Tenerife']

## Ordenando valores

Una operación muy típica cuando trabajamos con datos es la de ordenarlos en base a ciertos criterios. Veamos cómo podemos hacerlo utilizando pandas. Volvemos a nuestro «dataset» tecnológico:

>>> df	Revenue	Employees	City	Country
Company				
Apple	274515	147000	California	United States
Samsung Electronics	200734	267937	Suwon	South Korea
Alphabet	182527	135301	California	United States
Foxconn	181945	878429	New Taipei City	Taiwan
Microsoft	143015	163000	Washington	United States
Huawei	129184	197000	Shenzhen	China
Dell Technologies	92224	158000	Texas	United States
Facebook	85965	58604	California	United States
Sony	84893	109700	Tokyo	Japan
Hitachi	82345	350864	Tokyo	Japan
Intel	77867	110600	California	United States
IBM	73620	364800	New York	United States
Tencent	69864	85858	Shenzhen	China
Panasonic	63191	243540	Osaka	Japan
Lenovo	60742	71500	Hong Kong	China
HP Inc.	56639	53000	California	United States
LG Electronics	53625	75000	Seoul	South Korea

Supongamos que queremos tener el conjunto de datos **ordenado por el nombre de empresa**. Como, en este caso, la columna Company constituye el índice, debemos ordenar por el índice:

```
>>> df.sort_index()
```

Company	Revenue	Employees	City	Country
Alphabet	182527	135301	California	United States
Apple	274515	147000	California	United States
Dell Technologies	92224	158000	Texas	United States
Facebook	85965	58604	California	United States
Foxconn	181945	878429	New Taipei City	Taiwan
HP Inc.	56639	53000	California	United States
Hitachi	82345	350864	Tokyo	Japan
Huawei	129184	197000	Shenzhen	China
IBM	73620	364800	New York	United States
Intel	77867	110600	California	United States
LG Electronics	53625	75000	Seoul	South Korea
Lenovo	60742	71500	Hong Kong	China
Microsoft	143015	163000	Washington	United States
Panasonic	63191	243540	Osaka	Japan
Samsung Electronics	200734	267937	Suwon	South Korea
Sony	84893	109700	Tokyo	Japan
Tencent	69864	85858	Shenzhen	China

Ahora imaginemos que necesitamos tener las **empresas ordenadas de mayor a menor número de ingresos**:

```
>>> df.sort_values(by='Revenue', ascending=False)
```

Company	Revenue	Employees	City	Country
Apple	274515	147000	California	United States
Samsung Electronics	200734	267937	Suwon	South Korea
Alphabet	182527	135301	California	United States
Foxconn	181945	878429	New Taipei City	Taiwan
Microsoft	143015	163000	Washington	United States
Huawei	129184	197000	Shenzhen	China
Dell Technologies	92224	158000	Texas	United States
Facebook	85965	58604	California	United States
Sony	84893	109700	Tokyo	Japan
Hitachi	82345	350864	Tokyo	Japan
Intel	77867	110600	California	United States
IBM	73620	364800	New York	United States
Tencent	69864	85858	Shenzhen	China
Panasonic	63191	243540	Osaka	Japan
Lenovo	60742	71500	Hong Kong	China
HP Inc.	56639	53000	California	United States
LG Electronics	53625	75000	Seoul	South Korea

También es posible utilizar varias columnas en la ordenación. Pongamos que deseamos **ordenar los datos por país y por ciudad**. Veamos cómo afrontarlo:

	Revenue	Employees	City	Country
Company				
Lenovo	60742	71500	Hong Kong	China
Huawei	129184	197000	Shenzhen	China
Tencent	69864	85858	Shenzhen	China
Panasonic	63191	243540	Osaka	Japan
Sony	84893	109700	Tokyo	Japan
Hitachi	82345	350864	Tokyo	Japan
LG Electronics	53625	75000	Seoul	South Korea
Samsung Electronics	200734	267937	Suwon	South Korea
Foxconn	181945	878429	New Taipei City	Taiwan
Apple	274515	147000	California	United States
Alphabet	182527	135301	California	United States
Facebook	85965	58604	California	United States
Intel	77867	110600	California	United States
HP Inc.	56639	53000	California	United States
IBM	73620	364800	New York	United States
Dell Technologies	92224	158000	Texas	United States
Microsoft	143015	163000	Washington	United States

## Buscando máximos y mínimos

Al igual que veíamos *en el caso de las series*, podemos aplicar muchas de estas funciones de máximos y mínimos sobre un DataFrame de Pandas.

Podemos obtener los **valores mínimos y máximos de todas las columnas**:

```
>>> df.min()
Revenue      53625
Employees     53000
City          California
Country        China
dtype: object

>>> df.max()
Revenue      274515
Employees     878429
City          Washington
Country       United States
dtype: object
```

También podría ser de utilidad saber **qué empresa tiene el valor mínimo o máximo** para una determinada columna:

```
# LG tiene los menores ingresos
>>> df['Revenue'].idxmin()
'LG Electronics'

# Foxconn tiene el mayor número de empleados/as
>>> df['Employees'].idxmax()
'Foxconn'
```

---

**Nota:** En este caso nos devuelve una cadena de texto con el nombre de la empresa ya que tenemos definido así nuestro índice (etiquetas). En otro caso devolvería la posición (numérica) con un índice por defecto.

---

Si queremos acceder al registro completo, basta con acceder a través de la etiqueta devuelta:

```
>>> company = df['Revenue'].idxmin()

>>> df.loc[company]
Revenue      53625
Employees    75000
City          Seoul
Country       South Korea
Name: LG Electronics, dtype: object
```

Otra de las operaciones muy usuales es encontrar los  $n$  registros con mayores/menores valores. Supongamos que nos interesa conocer las **3 empresas con mayores ingresos y las 3 empresas con menor número de empleados/as**:

```
>>> df['Revenue'].nlargest(3)
Company
Apple           274515
Samsung Electronics  200734
Alphabet        182527
Name: Revenue, dtype: int64

>>> df['Employees'].nsmallest(3)
Company
HP Inc.         53000
Facebook        58604
Lenovo          71500
Name: Employees, dtype: int64
```

---

**Nota:** Si no especificamos un número de registros, estas funciones lo tienen definido por defecto a 5.

---

Si queremos acceder al registro completo, podemos aplicar estas funciones de otro modo:

```
>>> df.nlargest(3, 'Revenue')
      Revenue Employees      City      Country
Company
Apple           274515    147000 California United States
Samsung Electronics 200734    267937 Suwon South Korea
Alphabet        182527    135301 California United States

>>> df.nsmallest(3, 'Employees')
      Revenue Employees      City      Country
Company
HP Inc.         56639     53000 California United States
Facebook        85965     58604 California United States
Lenovo          60742     71500 Hong Kong China
```

## Ejercicio

Partiendo del conjunto de datos `democan` obtenga las 3 islas con menor densidad de población.

El resultado debería ser el siguiente:

	Population	Area	Province	Density
Island				
El Hierro	11147	278.71	SCTF	39.994977
La Gomera	21678	369.76	SCTF	58.627218
Fuerteventura	119732	1659.00	LPGC	72.171187

## Gestionando valores nulos

La limpieza de un «dataset» suele estar vinculado, en muchas ocasiones, a la gestión de los valores nulos. En este sentido, pandas ofrece varias funciones.

Para ejemplificar este apartado, vamos a hacer uso del siguiente DataFrame:

```
>>> df
   A   B   C
0  1  4.0  7.0
1  2  NaN  8.0
2  3  6.0  NaN
```

Si queremos **detectar aquellos valores nulos**, haremos lo siguiente:

```
>>> df.isna()
      A      B      C
0  False  False  False
1  False  True   False
2  False  False  True
```

---

**Nota:** También existe la función `isnull()` que funciona de manera análoga a `isna()`. En StackExchange puedes ver [una explicación de estas funciones](#).

---

En caso de que nos interese **descartar los registros con valores nulos**, procedemos así:

```
>>> df.dropna()
      A      B      C
0  1    4.0   7.0
```

Sin embargo, también existe la posibilidad de **rellenar los valores nulos** con algún sustituto. En este caso podemos ejecutar lo siguiente:

```
>>> df.fillna(0)
      A      B      C
0  1    4.0   0.0
1  2    0.0   0.0
2  3    6.0   9.0
```

Incluso podemos **aplicar interpolación para completar valores nulos**:

```
>>> df.interpolate()
      A      B      C
0  1    4.0   7.0
1  2    5.0   8.0
2  3    6.0   8.0
```

## Agrupando datos

Las operaciones de agregado son muy recurridas y nos permiten extraer información relevante, que, a simple vista, quizás no sea tan evidente.

Veamos un ejemplo en el que calculamos la **suma de los ingresos de las empresas, agrupados por país**:

```
>>> df.groupby('Country')['Revenue'].sum()
Country
China           259790
```

(continué en la próxima página)

(proviene de la página anterior)

Japan	230429
South Korea	254359
Taiwan	181945
United States	986372
Name: Revenue, dtype: int64	

También es posible realizar la agrupación en varios niveles. En el siguiente ejemplo tendremos los datos **agrupados por país y ciudad**:

>>> df.groupby(['Country', 'City'])['Revenue'].sum()	
Country City	
China Hong Kong 60742	
	Shenzhen 199048
Japan Osaka 63191	
	Tokyo 167238
South Korea Seoul 53625	
	Suwon 200734
Taiwan New Taipei City 181945	
United States California 677513	
	New York 73620
	Texas 92224
	Washington 143015
Name: Revenue, dtype: int64	

Incluso podemos aplicar distintas funciones de agregación a cada columna. Supongamos que necesitamos calcular **la media de los ingresos y la mediana del número de empleados/as, con las empresas agrupadas por país**:

>>> df.groupby('Country').agg({'Revenue': 'mean', 'Employees': 'median'})
Revenue Employees
Country
China 86596.666667 85858.0
Japan 76809.666667 243540.0
South Korea 127179.500000 171468.5
Taiwan 181945.000000 878429.0
United States 123296.500000 141150.5

---

**Nota:** Utilizamos la función `agg()` pasando un diccionario cuyas claves son nombres de columnas y cuyos valores son funciones a aplicar.

---

## Ejercicio

Obtenga el porcentaje de población (en relación con el total) de cada provincia de las Islas Canarias en base al «dataset» `democan`.

El resultado debería ser similar a:

- Las Palmas de Gran Canaria: 52%
  - Santa Cruz de Tenerife: 48%
- 

## Aplicando funciones

Pandas permite la aplicación de funciones (tanto propias como «built-in») a filas y/o columnas de un DataFrame.

Numpy nos ofrece una amplia gama de funciones matemáticas. Podemos hacer uso de cualquier de ellas aplicándola directamente a nuestro conjunto de datos. Veamos un ejemplo en el que obtenemos **el máximo de cada columna**:

```
>>> df.apply(np.max)
Revenue           274515
Employees        878429
City             Washington
Country          United States
dtype: object
```

---

**Truco:** En este caso equivalente a `df.max()`.

---

Podemos aplicar funciones sobre determinadas columnas. Supongamos que queremos obtener **el logaritmo de la serie de ingresos**:

```
>>> df['Revenue'].apply(np.log)
Company
Apple            12.522761
Samsung Electronics 12.209736
Alphabet         12.114653
Foxconn           12.111460
Microsoft         11.870705
Huawei             11.768993
Dell Technologies 11.431976
Facebook           11.361696
Sony                11.349147
Hitachi              11.318673
Intel                11.262758
IBM                  11.206672
Tencent              11.154306
Panasonic            11.053917
Lenovo                11.014391
```

(continué en la próxima página)

(proviene de la página anterior)

HP Inc.	10.944453
LG Electronics	10.889771
Name: Revenue, dtype:	float64

Ahora imaginemos un escenario en el que **la normativa de Estados Unidos ha cambiado y se obliga a sus empresas tecnológicas a aumentar un 5% el número de empleados/as** que tienen. Esto lo podríamos abordar escribiendo una función propia que gestione cada fila del «dataset» y devuelva el valor adecuado de empleados/as según las características de cada empresa:

```
>>> def raise_employment(row):
...     num_employees = row['Employees']
...     if row['Country'] == 'United States':
...         return num_employees * 1.05
...     return num_employees
```

Ahora ya podemos aplicar esta función a nuestro DataFrame, teniendo en cuenta que debemos actuar sobre el **eje de filas (axis=1)**:

```
>>> df.apply(raise_employment, axis=1)
Company
Apple           154350.00
Samsung Electronics  267937.00
Alphabet        142066.05
Foxconn          878429.00
Microsoft        171150.00
Huawei           197000.00
Dell Technologies 165900.00
Facebook          61534.20
Sony              109700.00
Hitachi           350864.00
Intel              116130.00
IBM                383040.00
Tencent            85858.00
Panasonic         243540.00
Lenovo             71500.00
HP Inc.            55650.00
LG Electronics      75000.00
dtype: float64
```

El resultado es una serie que se podría incorporar al conjunto de datos, o bien, reemplazar la columna *Employees* con estos valores.

---

## Ejercicio

Supongamos que el Gobierno de Canarias va a dar unas ayudas a cada isla en función de su

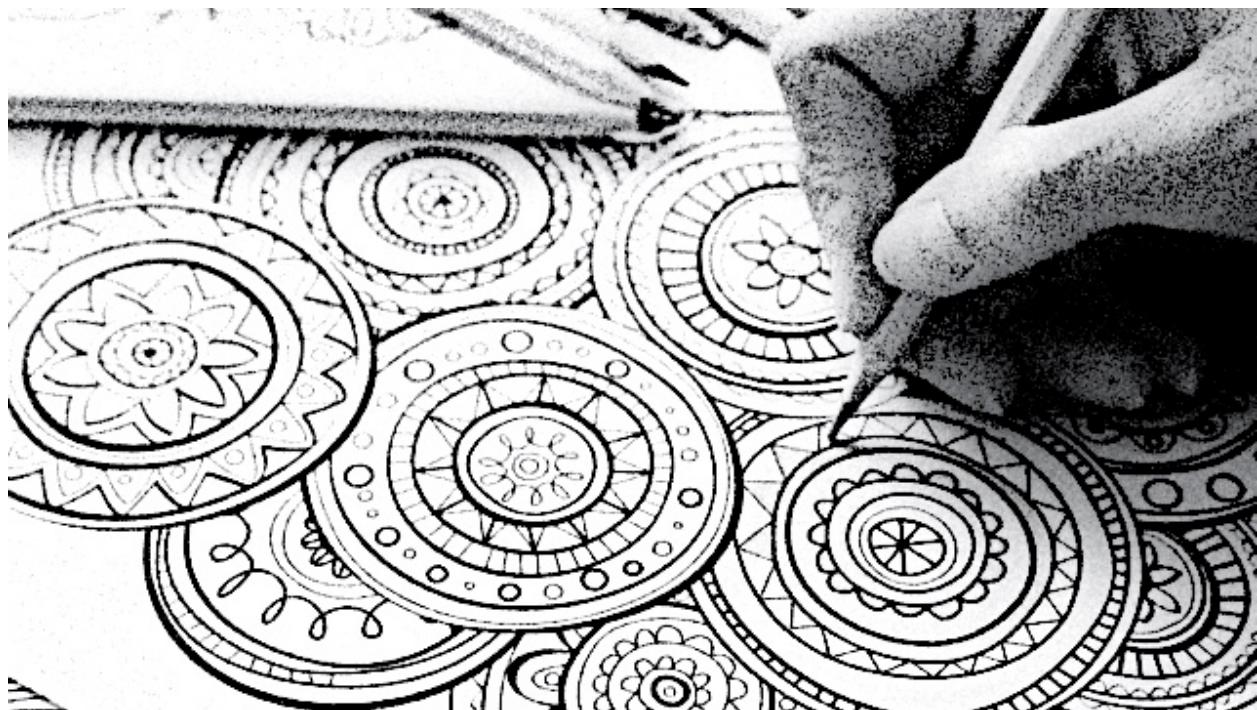
superficie y su población, con las siguientes reglas:

- Islas con menos de 1000 km<sup>2</sup>: ayuda del 30% de su población.
- Islas con más de 1000 km<sup>2</sup>: ayuda del 20% de su población.

Añada una nueva columna *Grant* al «dataset» `democan` donde se contemplen estas ayudas. El DataFrame debería quedar así:

Island	Population	Area	Province	Grant
Gran Canaria	855521	1560.10	LPGC	171104.2
Tenerife	928604	2034.38	SCTF	185720.8
La Palma	83458	708.32	SCTF	25037.4
Lanzarote	155812	845.94	LPGC	46743.6
La Gomera	21678	369.76	SCTF	6503.4
El Hierro	11147	278.71	SCTF	3344.1
Fuerteventura	119732	1659.00	LPGC	23946.4

## 8.4 matplotlib



matplotlib es el paquete Python más utilizado en el ámbito de la ciencia de datos para representaciones gráficas.<sup>1</sup>

<sup>1</sup> Foto original de portada por [Customerbox](#) en Unsplash.

```
$ pip install matplotlib
```

La forma más común de importar esta librería es usar el alias `plt` de la siguiente manera:

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

---

**Importante:** Si bien podemos utilizar `matplotlib` en el intérprete habitual de Python, suele ser muy frecuente trabajar con esta librería mediante entornos *Jupyter*, ya que facilitan la visualización de los gráficos en su interfaz de usuario.

---

### 8.4.1 Figura

La `figura` es el elemento base sobre el que se construyen todos los gráficos en `matplotlib`. Veamos cómo crearla:

```
>>> fig = plt.figure()

>>> type(fig)
matplotlib.figure.Figure

>>> fig
<Figure size 640x480 with 0 Axes>
```

Podemos observar que la resolución (por defecto) de la figura es de 640x480 píxeles y que no dispone de ningún eje («0 Axes»).

---

**Importante:** El término «axes» hace referencia a un conjunto de ejes. Puede resultar confuso en español y he decidido asignar el nombre **marco** cuando haga referencia a «axes».

---

La resolución final de una figura viene determinada por su altura (`height`) y anchura (`width`) especificadas en pulgadas<sup>2</sup> que, a su vez, se multiplican por los puntos por pulgada o `dpi`. Veamos el funcionamiento:

```
>>> fig
<Figure size 640x480 with 0 Axes>

>>> fig.get_figwidth()    # pulgadas
6.4
>>> fig.get_figheight()  # pulgadas
```

(continué en la próxima página)

<sup>2</sup> Se suele usar el término inglés «inches».

(provine de la página anterior)

```
4.8
>>> fig.get_dpi()      # dots per inch
100.0

>>> fig.get_figwidth() * fig.dpi, fig.get_figheight() * fig.dpi
(640.0, 480.0)
```

---

**Importante:** Si utilizamos entornos de desarrollo basados en Jupyter, los valores por defecto son distintos:

- Ancho de figura: 6 in
  - Alto de figura: 4 in
  - DPI: 75
  - Resolución: 450x300 px
- 

Por tanto, cuando creamos una figura podemos modificar los parámetros por defecto para obtener la resolución deseada:

```
>>> fig = plt.figure(figsize=(19.2, 10.8)) # 100 dpi
>>> fig
<Figure size 1920x1080 with 0 Axes>

>>> fig = plt.figure(figsize=(19.2, 10.8), dpi=300)
>>> fig
<Figure size 5760x3240 with 0 Axes>
```

Si nos interesa que cualquier figura tome unos valores concretos de resolución, podemos modificar los **valores por defecto del entorno**. Para ello, matplotlib hace uso de un diccionario `plt.rcParams` que contiene los parámetros globales de configuración. Veamos cómo modificarlo:

```
>>> plt.rcParams['figure.figsize']
[6.4, 4.8]
>>> plt.rcParams['figure.dpi']
100.0

>>> plt.rcParams['figure.figsize'] = (10, 5)
>>> plt.rcParams['figure.dpi'] = 300          # res. final: 3000x1500 px

>>> fig.get_figwidth()
10.0
>>> fig.get_figheight()
```

(continué en la próxima página)

(provien de la página anterior)

```
5.0
>>> fig.dpi
300.0
```

## 8.4.2 Marcos

Para poder empezar a graficar necesitamos tener, al menos, un marco. Utilizaremos la función `add_subplot()` que requiere pasar como parámetros el número de filas, el número de columnas y el marco activo:

```
fig.add_subplot(2, 3, 5)
```

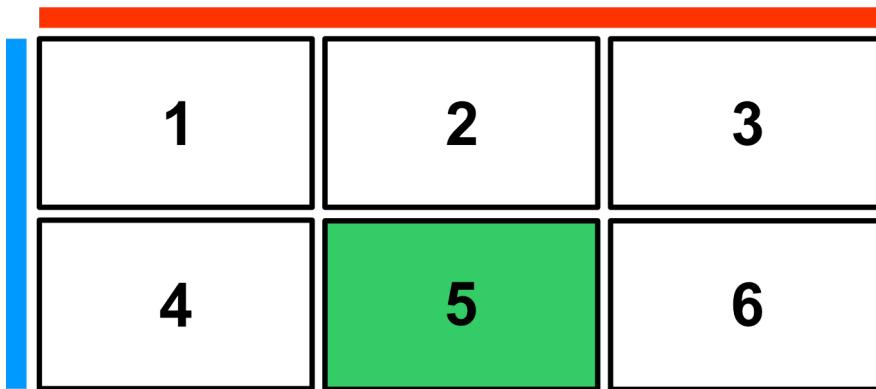


Figura 15: Creación de marcos dentro de una figura

Para comenzar vamos a trabajar únicamente con un marco:

```
>>> fig = plt.figure()

>>> ax = fig.add_subplot(1, 1, 1) # equivalente a fig.add_subplot(111)

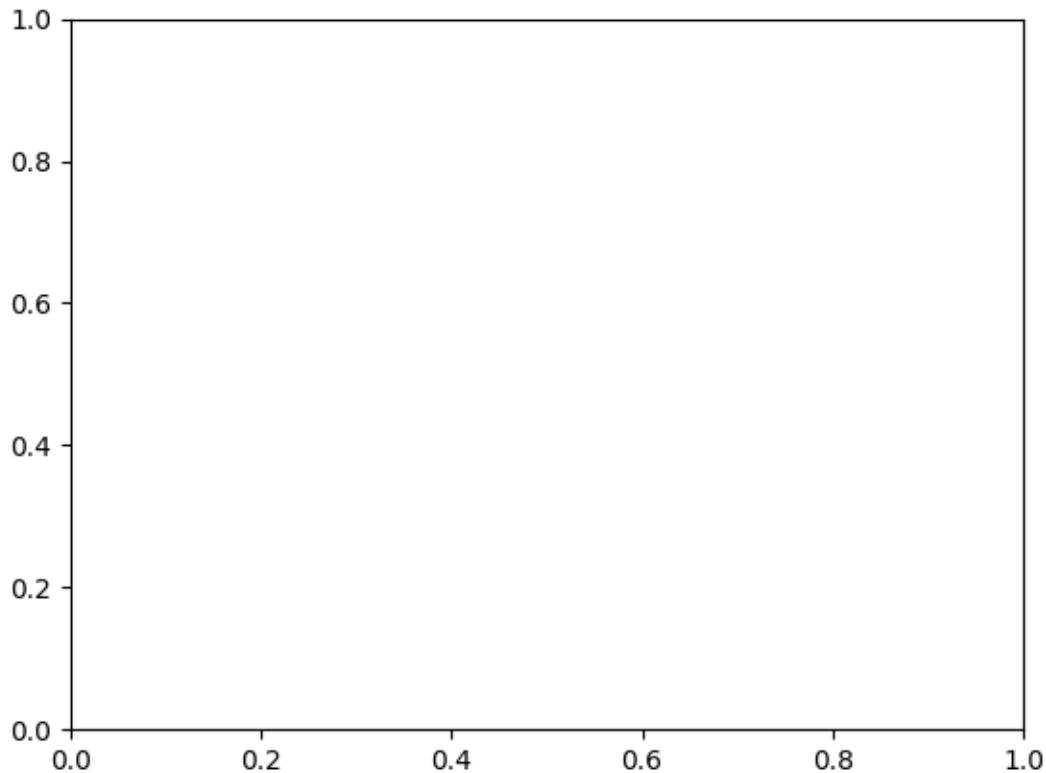
>>> ax
<AxesSubplot:>

>>> fig
<Figure size 640x480 with 1 Axes>
```

---

**Truco:** Suele ser habitual encontrar `ax` como nombre de variable del «axes» devuelto por la función `add_subplot()`.

---



---

**Nota:** La escala por defecto de cada eje va de 0 a 1 con marcas cada 0.2

---

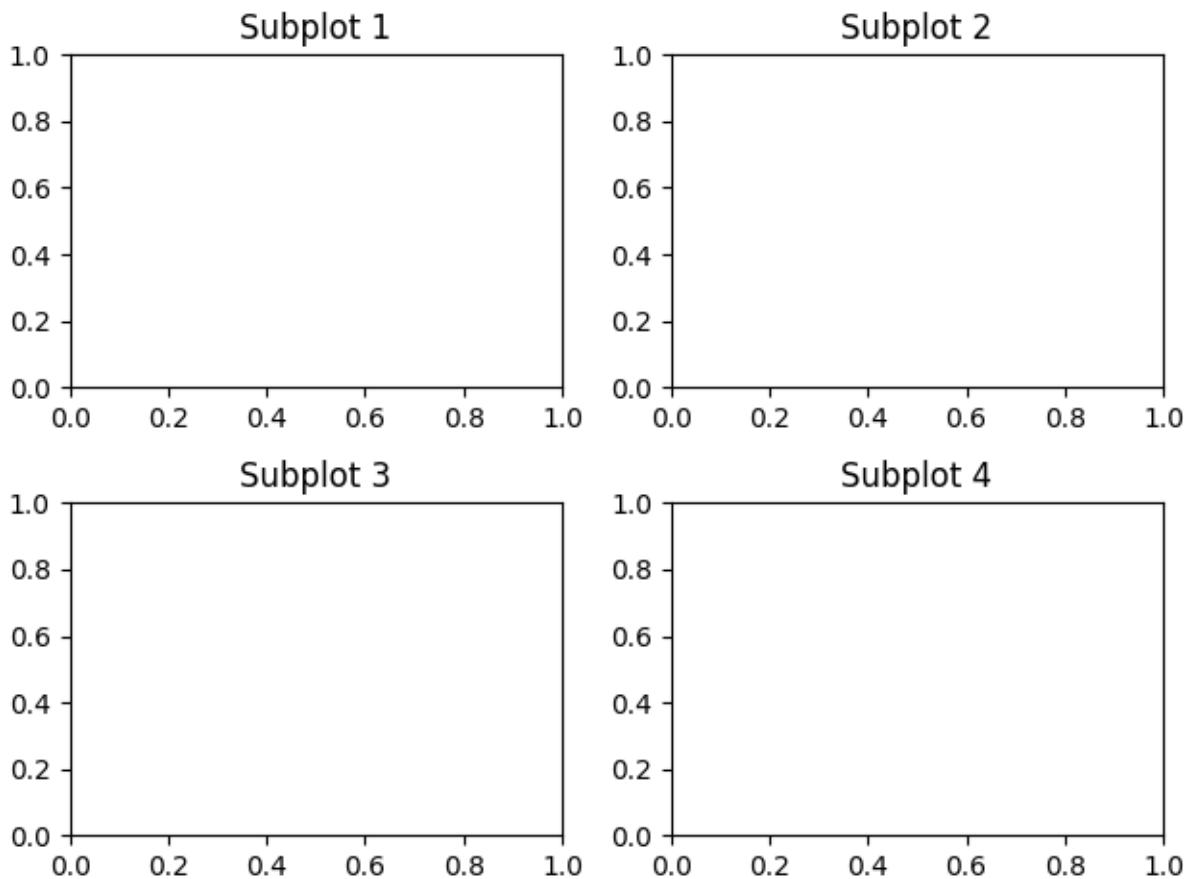
Ahora vamos a generar 4 marcos sobre los que fijaremos un título identificativo:

```
>>> fig = plt.figure()

>>> for i in range(1, 5):
...     ax = fig.add_subplot(2, 2, i)
...     ax.set_title(f'Subplot {i}')

>>> fig.tight_layout(pad=1) # sólo para que no se solapen los títulos

>>> fig
<Figure size 640x480 with 4 Axes>
```



## Atajo para subgráficos

Matplotlib nos ofrece una forma compacta de crear a la vez tanto la **figura** como los **marcos** que necesitemos.

Para ello utilizaremos la función `plt.subplots()` que recibe como parámetros el *número de filas* y el *número de columnas* para la disposición de los marcos, y devuelve una tupla con la figura y los marcos.

En el siguiente ejemplo creamos **una figura con un único marco**:

```
>>> fig, ax = plt.subplots(1, 1)

>>> fig
<Figure size 640x480 with 1 Axes>

>>> ax
<AxesSubplot:>
```

---

**Truco:** Si invocamos la función `plt.subplots()` sin parámetros, creará (por defecto) un

único marco.

---

En el siguiente ejemplo creamos **una figura con 6 marcos** en disposición de 2 filas por 3 columnas:

```
>>> fig, ax = plt.subplots(2, 3)

>>> fig
<Figure size 640x480 with 6 Axes>

>>> ax
array([[<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>],
       [<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>]], dtype=object)

>>> ax.shape
(2, 3)
```

---

**Nota:** Se podría ver la función `subplots()` como una combinación de `figure()` + `add_subplot()`.

---

## Etiquetas

Dentro de un marco también es posible fijar las etiquetas de los ejes (X e Y). Veamos cómo hacerlo:

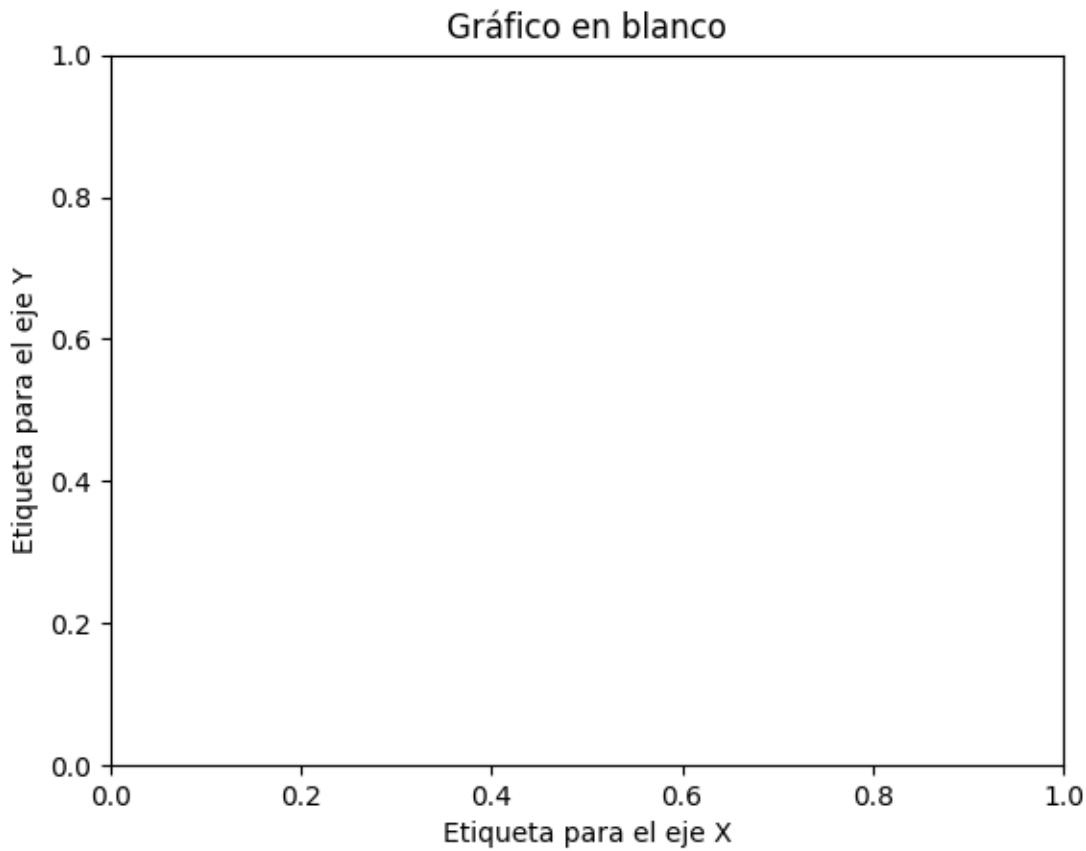
```
>>> fig, ax = plt.subplots()

>>> ax.set_title('Gráfico en blanco')
Text(0.5, 1.0, 'Gráfico en blanco')

>>> ax.set_xlabel('Etiqueta para el eje X')
Text(0.5, 0, 'Etiqueta para el eje X')

>>> ax.set_ylabel('Etiqueta para el eje Y')
Text(0, 0.5, 'Etiqueta para el eje Y')

>>> fig
<Figure size 640x480 with 1 Axes>
```



## Ejes

Un marco (2D) está compuesto por dos ejes: eje X e eje Y. Podemos acceder a cada eje mediante sendos atributos:

```
>>> ax.xaxis  
<matplotlib.axis.XAxis at 0x112b34100>  
  
>>> ax.yaxis  
<matplotlib.axis.YAxis at 0x112b34850>
```

## Rejilla

En cada eje podemos activar o desactivar la rejilla, así como indicar su estilo.

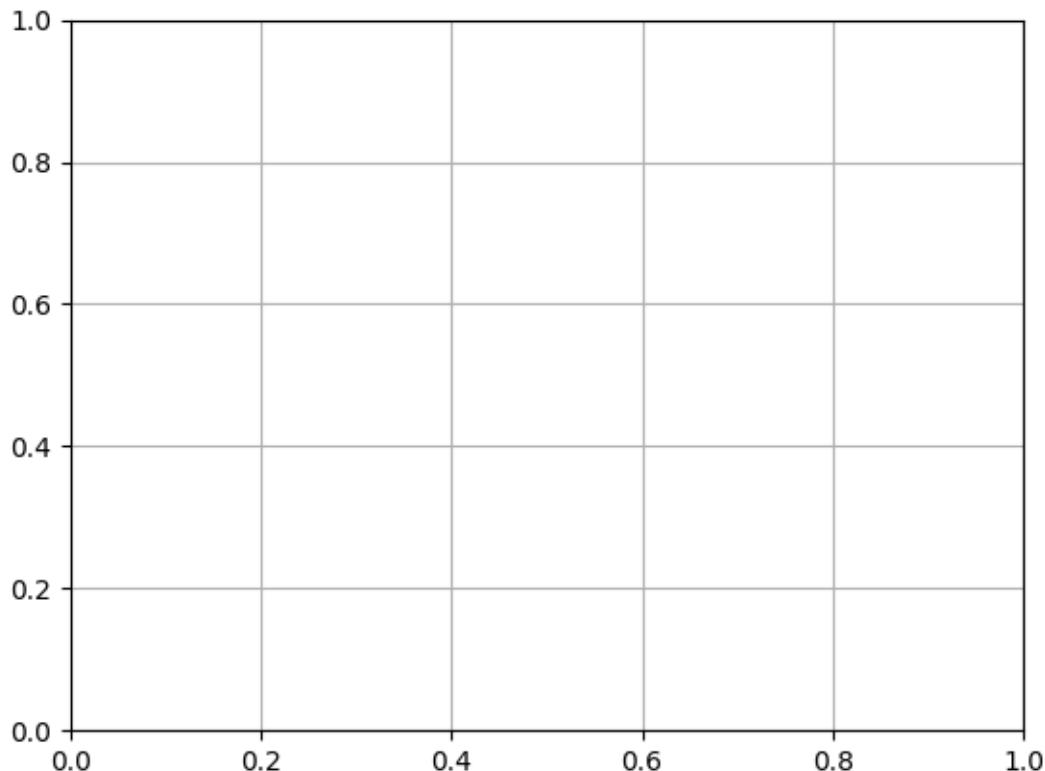
En primer lugar vamos a activar la rejilla en ambos ejes:

```
>>> ax.xaxis.grid(True)  
>>> ax.yaxis.grid(True)
```

Esto sería equivalente a:

```
>>> ax.grid(True)
```

Y obtendríamos una figura con la rejilla (por defecto):



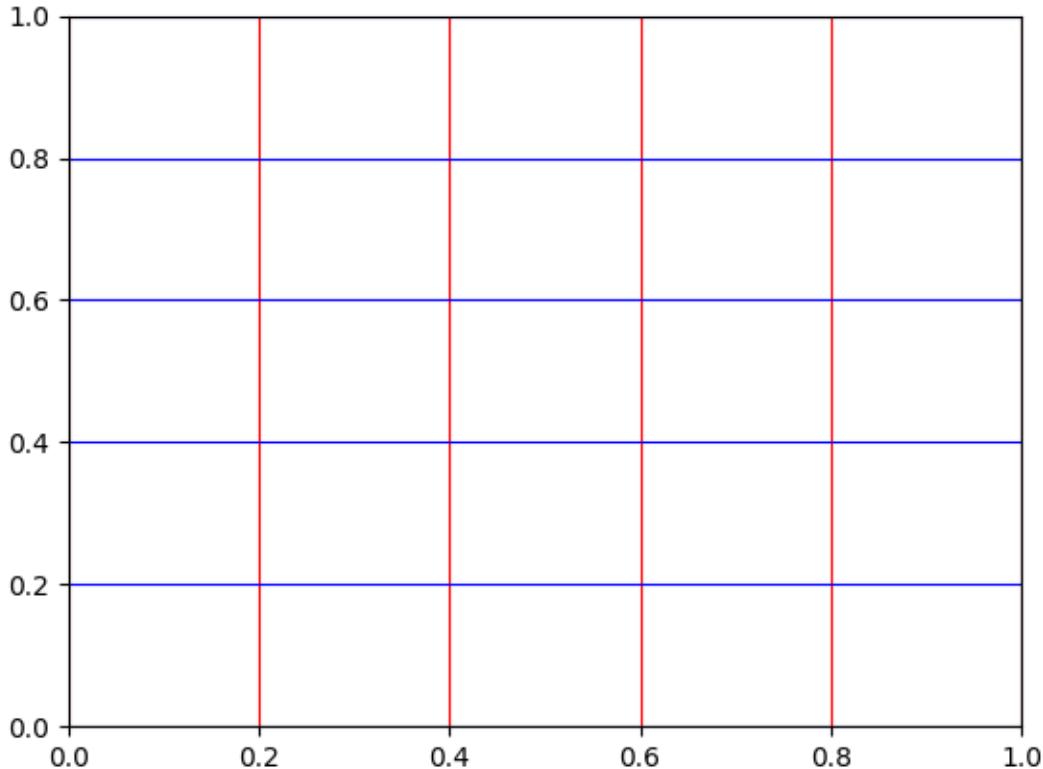
---

**Truco:** Las funciones de matplotlib que actúan como «interruptores» tienen por defecto el valor verdadero. En este sentido `ax.grid()` invocada sin parámetros hace que se muestre la rejilla. Esto se puede aplicar a muchas otras funciones.

---

Supongamos ahora que queremos personalizar la rejilla con **estilos diferentes en cada eje**:

```
>>> ax.xaxis.grid(color='r', linestyle='-' ) # equivale a color='red', linestyle=
   ↵'solid'
>>> ax.yaxis.grid(color='b', linestyle='-' ) # equivale a color='blue', linestyle=
   ↵'solid'
```



- Parámetros disponibles para creación del grid.
- Listado de nombres de colores en matplotlib.
- Estilos de línea en matplotlib.

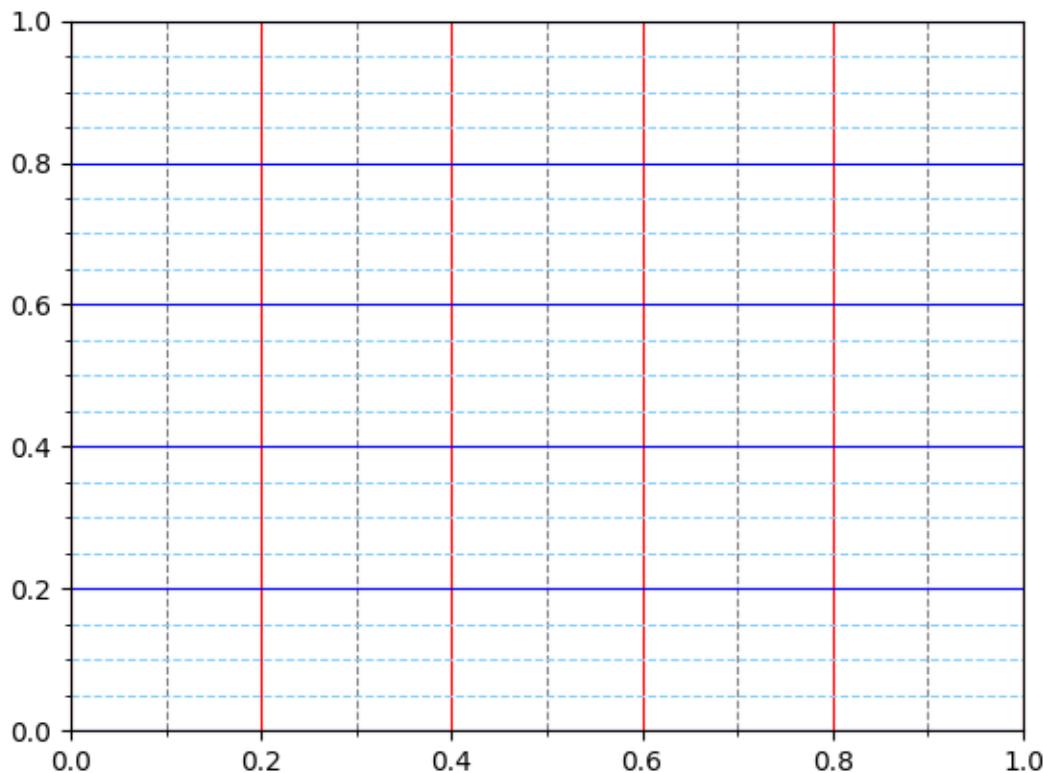
## Marcas

Por defecto, los ejes del marco tienen unas marcas<sup>3</sup> equiespaciadas que constituyen las *marcas mayores*. Igualmente existen unas *marcas menores* que, a priori, no están activadas.

Ambos elementos son susceptibles de modificarse. Veamos un ejemplo en el que establecemos las **marcas menores con distinto espaciado en cada eje** y además le damos un estilo diferente a cada rejilla:

<sup>3</sup> Se suele usar el término inglés «ticks».

```
>>> from matplotlib.ticker import MultipleLocator  
  
>>> ax.xaxis.set_minor_locator(MultipleLocator(0.1)) # X: separación cada 0.1  
    ↵unidades  
>>> ax.yaxis.set_minor_locator(MultipleLocator(0.05)) # Y: separación cada 0.05  
    ↵unidades  
  
>>> ax.xaxis.grid(which='minor', linestyle='dashed', color='gray')  
>>> ax.yaxis.grid(which='minor', linestyle='dashed', color='lightskyblue')
```



También es posible asignar etiquetas a las marcas menores. En ese sentido, veremos un ejemplo en el que incorporamos los **valores a los ejes con estilos propios**:

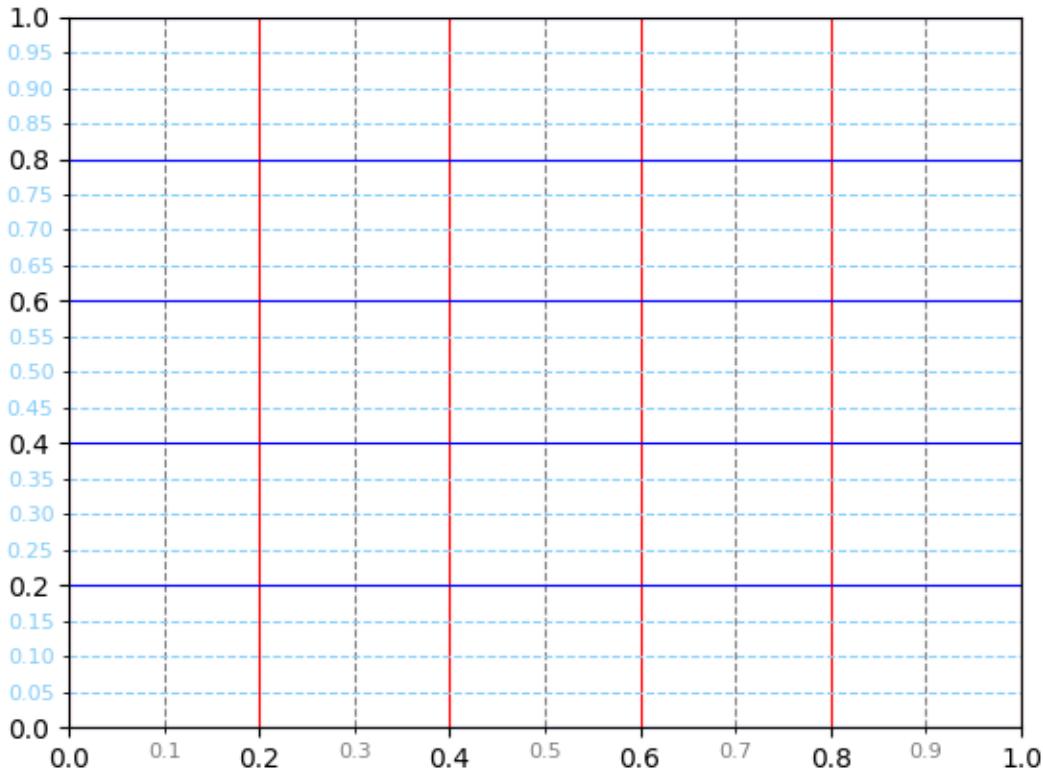
- Marcas menores en el eje X: precisión de 1 decimal, tamaño de letra 8 y color gris.
- Marcas menores en el eje Y: precisión de 2 decimales, tamaño de letra 8 y color azul.

```
>>> # Eje X  
>>> ax.xaxis.set_minor_formatter('{x:.1f}')  
>>> ax.tick_params(axis='x', which='minor', labelsize=8, labelcolor='gray')
```

(continué en la próxima página)

(provine de la página anterior)

```
>>> # Eje Y
>>> ax.yaxis.set_minor_formatter('{x:.2f}')
>>> ax.tick_params(axis='y', which='minor', labelsize=8, labelcolor='lightskyblue')
```

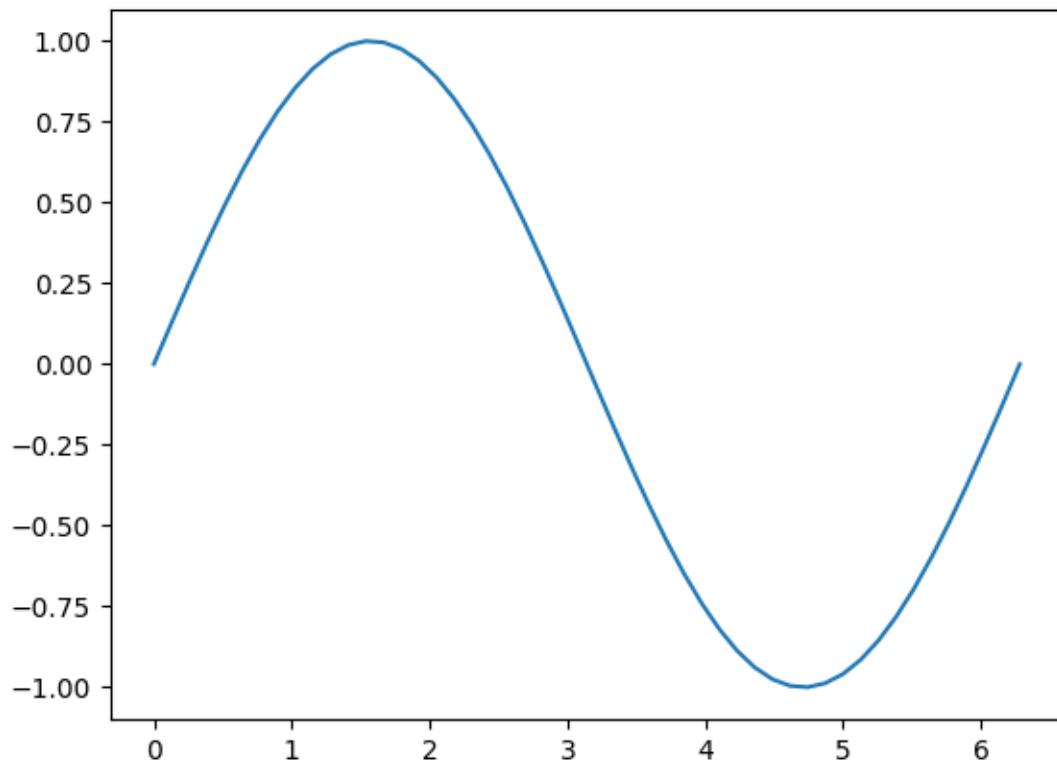


### 8.4.3 Primeros pasos

Vamos a empezar por representar la función  $f(x) = \sin(x)$ . Para ello crearemos una variable  $x$  con valores flotantes equidistantes y una variable  $y$  aplicando la función senoidal. Nos apoyamos en `numpy` para ello. A continuación usaremos la función `plot()` del marco para representar la función creada:

```
>>> x = np.linspace(0, 2 * np.pi)
>>> y = np.sin(x)

>>> fig, ax = plt.subplots()
>>> ax.plot(x, y)
[<matplotlib.lines.Line2D at 0x120914040>]
```



## Múltiples funciones

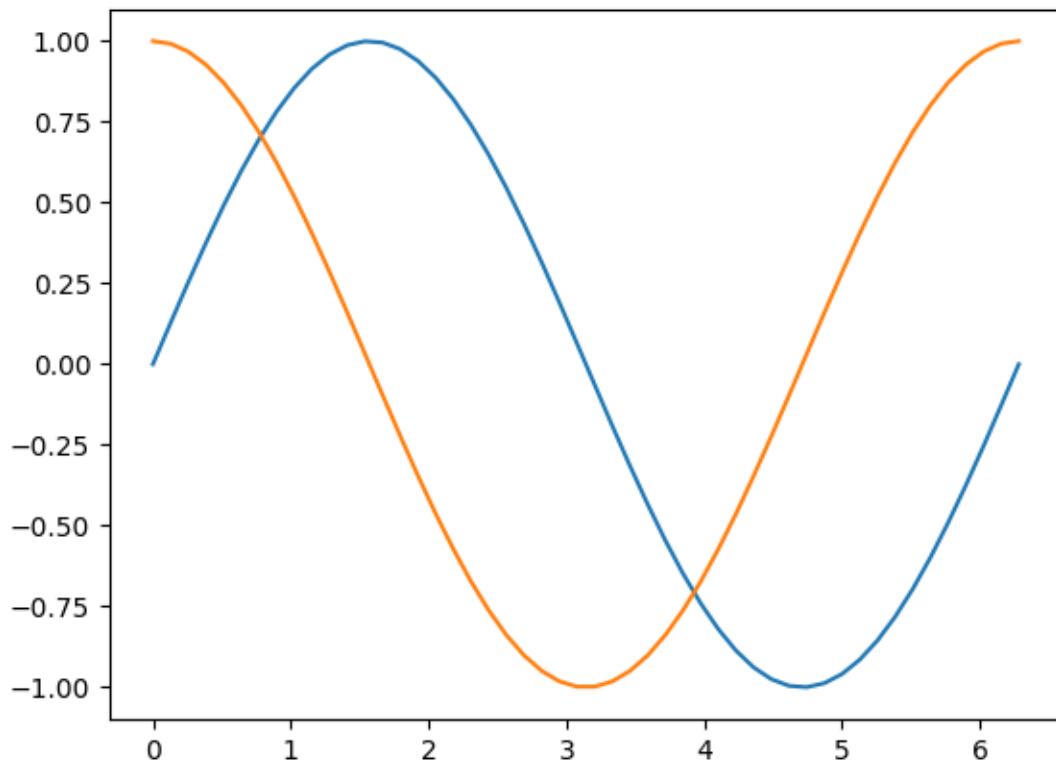
Partiendo de un mismo marco, es posible graficar todas las funciones que necesitemos. A continuación crearemos un marco con las funciones seno y coseno:

```
>>> x = np.linspace(0, 2 * np.pi)
>>> sin = np.sin(x)
>>> cos = np.cos(x)

>>> fig, ax = plt.subplots()

>>> ax.plot(x, sin)
[<matplotlib.lines.Line2D at 0x1247b6310>]

>>> ax.plot(x, cos)
[<matplotlib.lines.Line2D at 0x112b0d4c0>]
```



---

**Nota:** Los colores «auto» asignados a las funciones siguen un ciclo establecido por matplotlib que es igualmente personalizable.

---

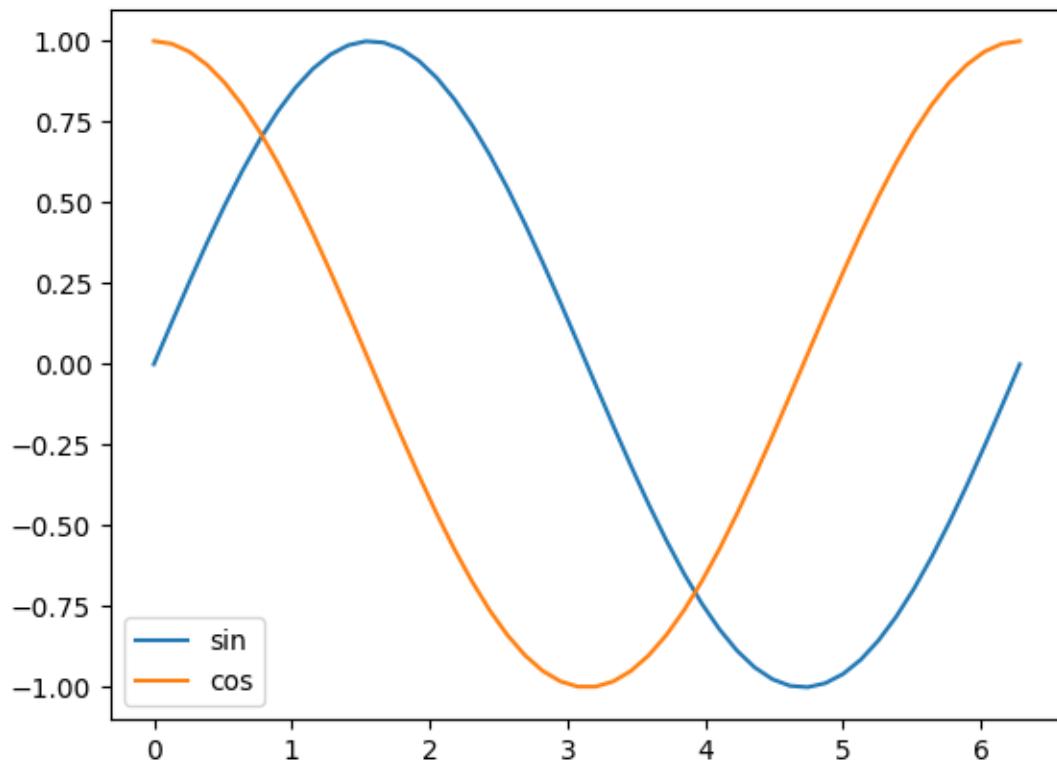
## Leyenda

En el caso de que tengamos múltiples gráficos en el mismo marco puede ser deseable mostrar una leyenda identificativa. Para usarla necesitamos asignar etiquetas a cada función. Veamos a continuación cómo incorporar una leyenda:

```
>>> ax.plot(x, sin, label='sin')
[<matplotlib.lines.Line2D at 0x124e07ac0>]

>>> ax.plot(x, cos, label='cos')
[<matplotlib.lines.Line2D at 0x123c58f10>]

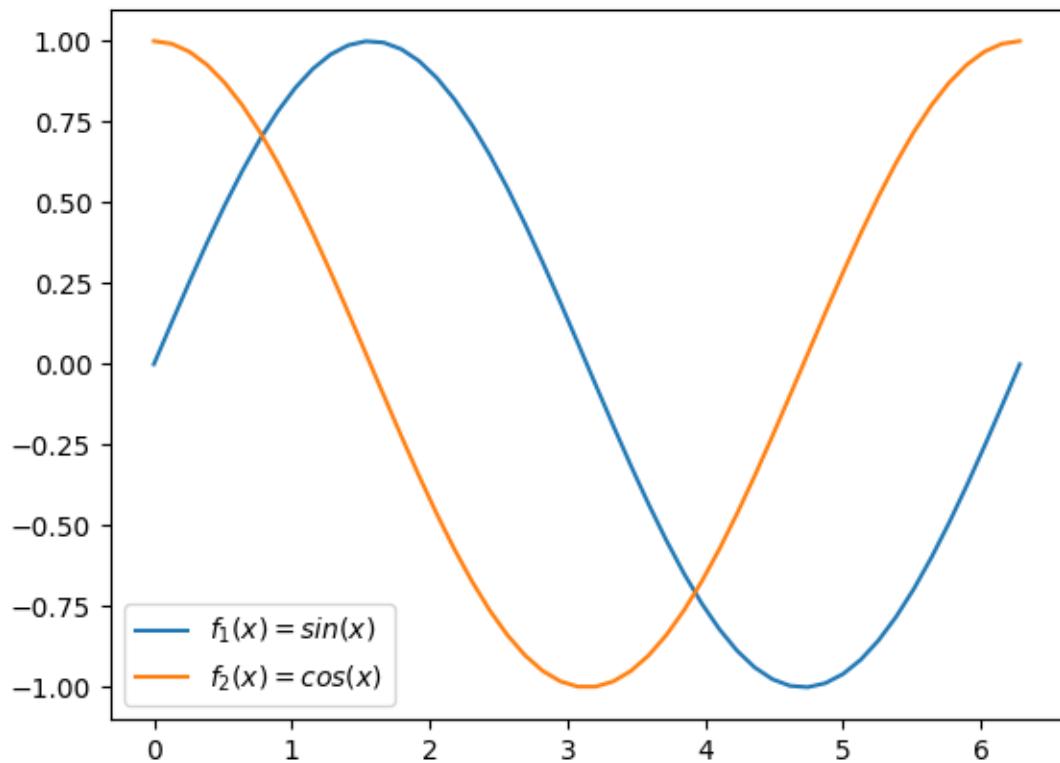
>>> ax.legend()
<matplotlib.legend.Legend at 0x123c8f190>
```



Es posible incorporar sintaxis `Latex` en los distintos elementos textuales de `matplotlib`. En el siguiente ejemplo usaremos esta notación en las etiquetas de las funciones utilizando el símbolo `$ ... $` para ello:

```
>>> ax.plot(x, sin, label='$f_1(x) = \sin(x)$')
[<matplotlib.lines.Line2D at 0x11682f3a0>]

>>> ax.plot(x, cos, label='$f_2(x) = \cos(x)$')
[<matplotlib.lines.Line2D at 0x11682b3a0>]
```

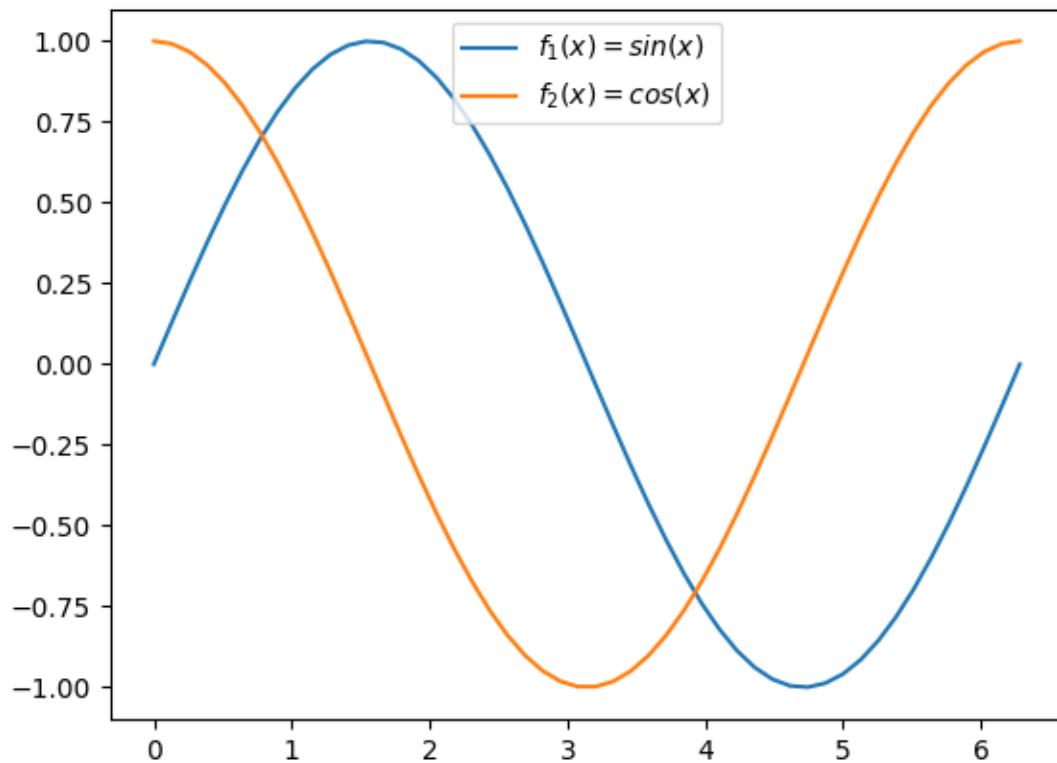


## Ubicación de la leyenda

Matplotlib intenta encontrar la **mejor ubicación** para la leyenda en el marco. Sin embargo, también es posible personalizar el lugar en el que queremos colocarla.

Si nos interesa situar la leyenda en la **parte superior central** del marco haríamos lo siguiente:

```
>>> ax.legend(loc='upper center')
<matplotlib.legend.Legend at 0x1167d43a0>
```

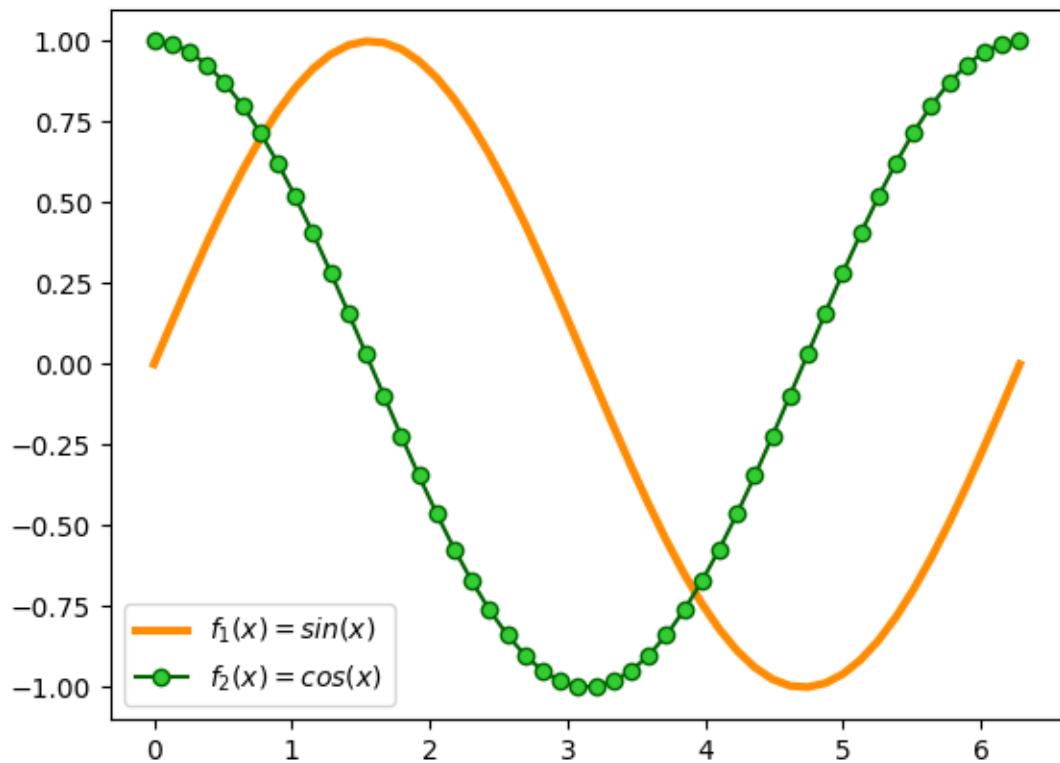


## Aplicando estilos

Para cada función que incluimos en el marco es posible establecer un estilo personalizado con [multitud de parámetros](#). Veamos la aplicación de algunos de estos parámetros a las funciones seno y coseno con las que hemos estado trabajando:

```
>>> sin_style = dict(linewidth=3, color='darkorange')
>>> cos_style = dict(marker='o', markerfacecolor='limegreen', color='darkgreen')

>>> ax.plot(x, sin, label='$f_1(x) = \sin(x)$', **sin_style)
[<matplotlib.lines.Line2D at 0x1131e9fd0>]
>>> ax.plot(x, cos, label='$f_2(x) = \cos(x)$', **cos_style)
[<matplotlib.lines.Line2D at 0x1226d76d0>]
```

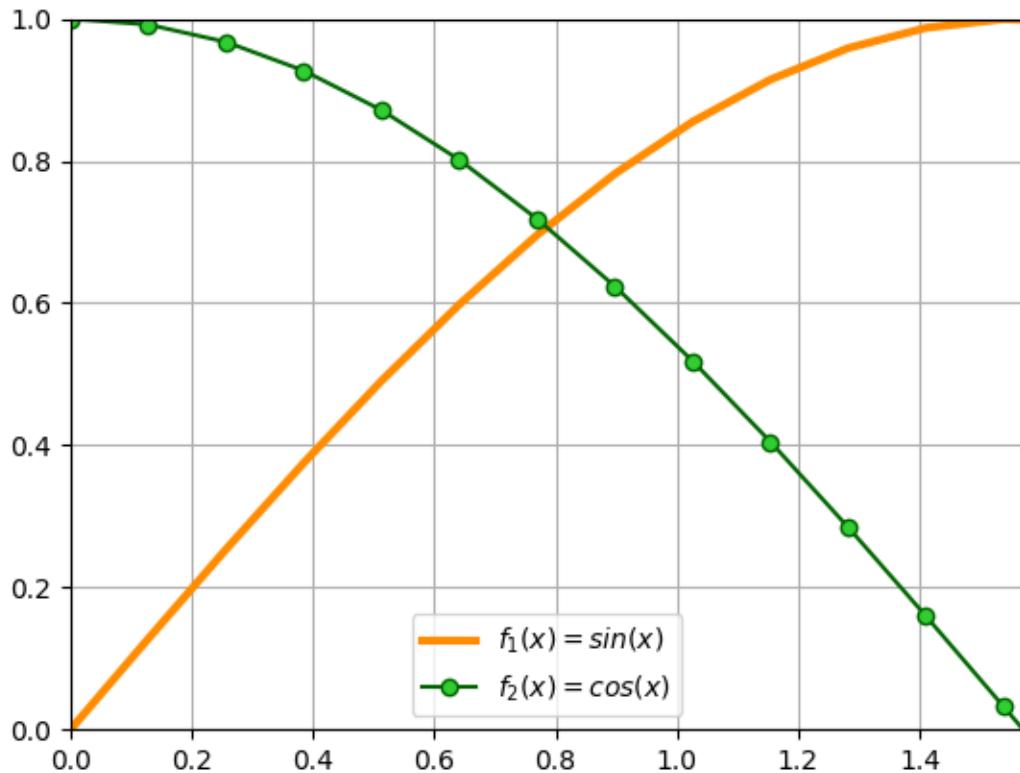


## Acotando ejes

Hay veces que nos interesa definir los límites de los ejes. En ese caso, podemos hacerlo de una manera muy sencilla:

```
>>> ax.set_xlim(0, np.pi / 2)
>>> ax.set_ylim(0, 1)

>>> ax.grid() # sólo a efectos estéticos
```



---

**Truco:** También es posible especificar únicamente límite inferior o superior en ambas funciones `set_xlim()` y `set_ylim()`. En ese caso, el otro valor sería ajustado automáticamente por matplotlib.

---

### Anotaciones

En ocasiones necesitamos añadir ciertas anotaciones al gráfico que estamos diseñando. Esto permite destacar áreas o detalles que pueden ser relevantes.

Partiendo de las funciones seno y coseno con las que hemos estado trabajando, vamos a suponer que queremos obtener sus puntos de corte, es decir, resolver la siguiente ecuación:

$$\begin{aligned} \sin(x) &= \cos(x) \\ \Downarrow \\ x &= \frac{\pi}{4} + \pi n, \quad n \in \mathbb{Z} \end{aligned}$$

Para el caso que nos ocupa haríamos  $n = 0$  con lo que obtendríamos la siguiente solución:

```
>>> xsol = np.pi / 4 + np.pi * 0
>>> ysol = np.sin(xsol)

>>> xsol, ysol
(0.7853981633974483, 0.7071067811865475)
```

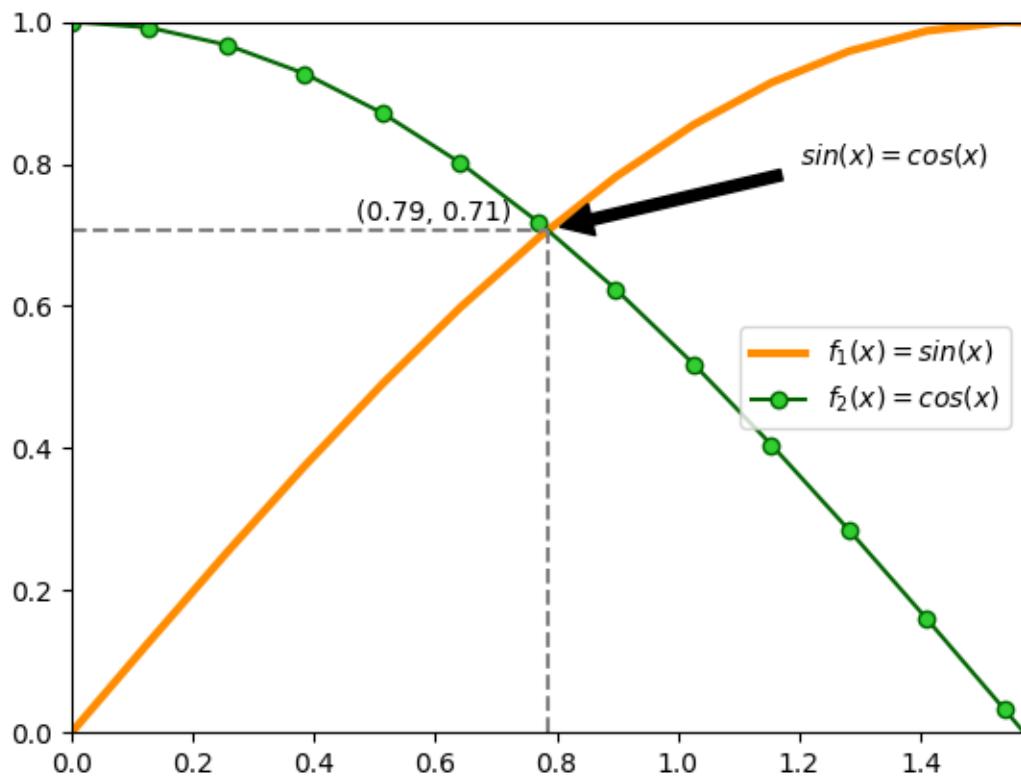
Vamos a insertar una serie de anotaciones en el gráfico:

- Flecha en el punto de corte con etiqueta de ecuación.
- Coordenadas de solución en el punto de corte.
- Proyección del punto de corte hacia ambos ejes.

```
>>> ax.annotate('$\sin(x) = \cos(x)$',
...             xy=(xsol, ysol),
...             xytext=(1.2, 0.8),
...             arrowprops=dict(facecolor='black', shrink=0.05))

>>> ax.text(0.47, 0.72, f'({xsol:.2f}, {ysol:.2f})')

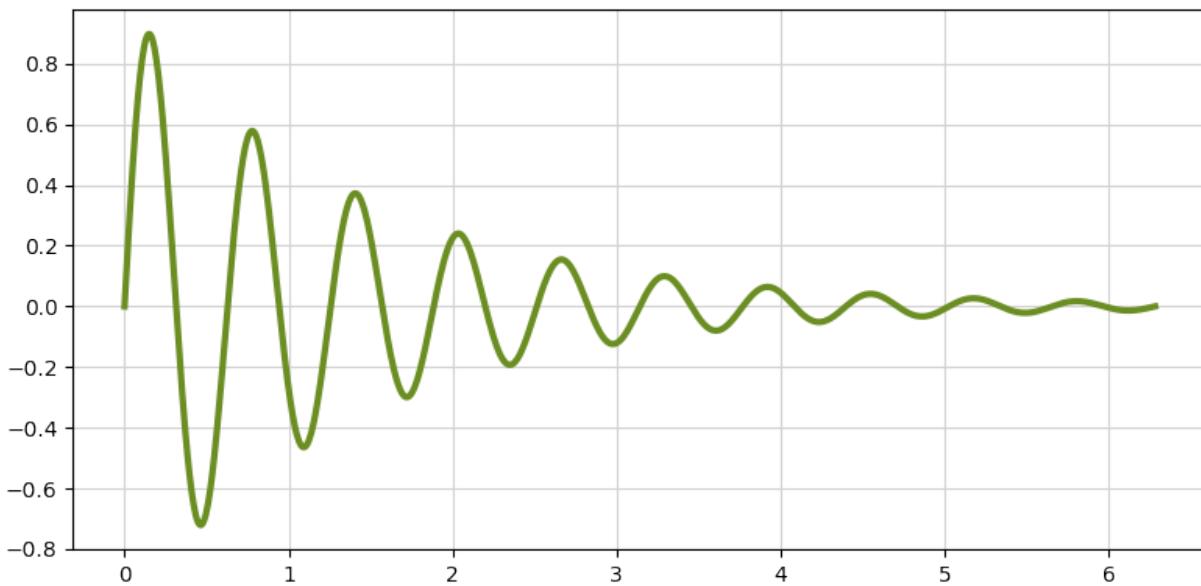
>>> ax.plot([xsol, xsol], [0, ysol], color='gray', linestyle='--')
>>> ax.plot([0, xsol], [ysol, ysol], color='gray', linestyle='--')
```



---

### Ejercicio

Escriba el código Python necesario para obtener el siguiente gráfico:



Datos:

- $x \in [0, 2\pi]$  (1000 puntos)
- $y = e^{-\alpha x} \sin(\beta x)$ , donde  $\alpha = 0.7$  y  $\beta = 10$ .

#### 8.4.4 Tipos de gráficos

Mediante matplotlib podemos hacer prácticamente cualquier tipo de gráfico. En esta sección haremos un repaso por algunos de ellos.

##### Gráficos de barras

Vamos a partir de un «dataset» que contiene los resultados de los Juegos Olímpicos de Tokio 2020. Hemos descargado el fichero `medals.xlsx` desde una página de Kaggle<sup>4</sup>.

En primer lugar cargaremos este fichero en un DataFrame y haremos una pequeña «limpieza»:

```
>>> df = pd.read_excel('pypi/datascience/files/medals.xlsx')
```

```
>>> df.head()
```

		Team/NOC	Gold	Silver	Bronze	Total	Rank by Total
0	1	United States of America	39	41	33	113	1

(continué en la próxima página)

<sup>4</sup> Kaggle es un servicio web que ofrece una gran variedad de «datasets», así como código, cursos y otros recursos en relación con la ciencia de datos.

(proviene de la página anterior)

1	2	People's Republic of China	38	32	18	88	2
2	3	Japan	27	14	17	58	5
3	4	Great Britain	22	21	22	65	4
4	5	ROC	20	28	23	71	3

```
>>> df.rename(columns={'Team/NOC': 'Country'}, inplace=True)
>>> df.set_index('Country', inplace=True)

>>> df.head()
```

Country	Rank	Gold	Silver	Bronze	Total	Rank by Total
United States of America	1	39	41	33	113	1
People's Republic of China	2	38	32	18	88	2
Japan	3	27	14	17	58	5
Great Britain	4	22	21	22	65	4
ROC	5	20	28	23	71	3

---

**Importante:** Para la carga de ficheros Excel, es necesario instalar un paquete adicional denominado `openpyxl`.

---

A continuación crearemos un **gráfico de barras con las medallas de oro, plata y bronce de los 10 primeros países ordenados por su ranking**. Lo primero será crear el subconjunto de datos sobre el que vamos a trabajar. Hay muchas maneras de hacerlo. Una de ellas:

>>> df_best = df.nsmallest(10, 'Rank')						
>>> df_best						
Country	Rank	Gold	Silver	Bronze	Total	Rank by Total
United States of America	1	39	41	33	113	1
People's Republic of China	2	38	32	18	88	2
Japan	3	27	14	17	58	5
Great Britain	4	22	21	22	65	4
ROC	5	20	28	23	71	3
Australia	6	17	7	22	46	6
Netherlands	7	10	12	14	36	9
France	8	10	12	11	33	10
Germany	9	10	11	16	37	8
Italy	10	10	10	20	40	7

Ahora ya podemos centrarnos en el diseño del gráfico de barras:

```
>>> fig, ax = plt.subplots(figsize=(8, 5), dpi=100) # 800x500 px

>>> bar_width = 0.30
>>> x = np.arange(df_best.index.size)

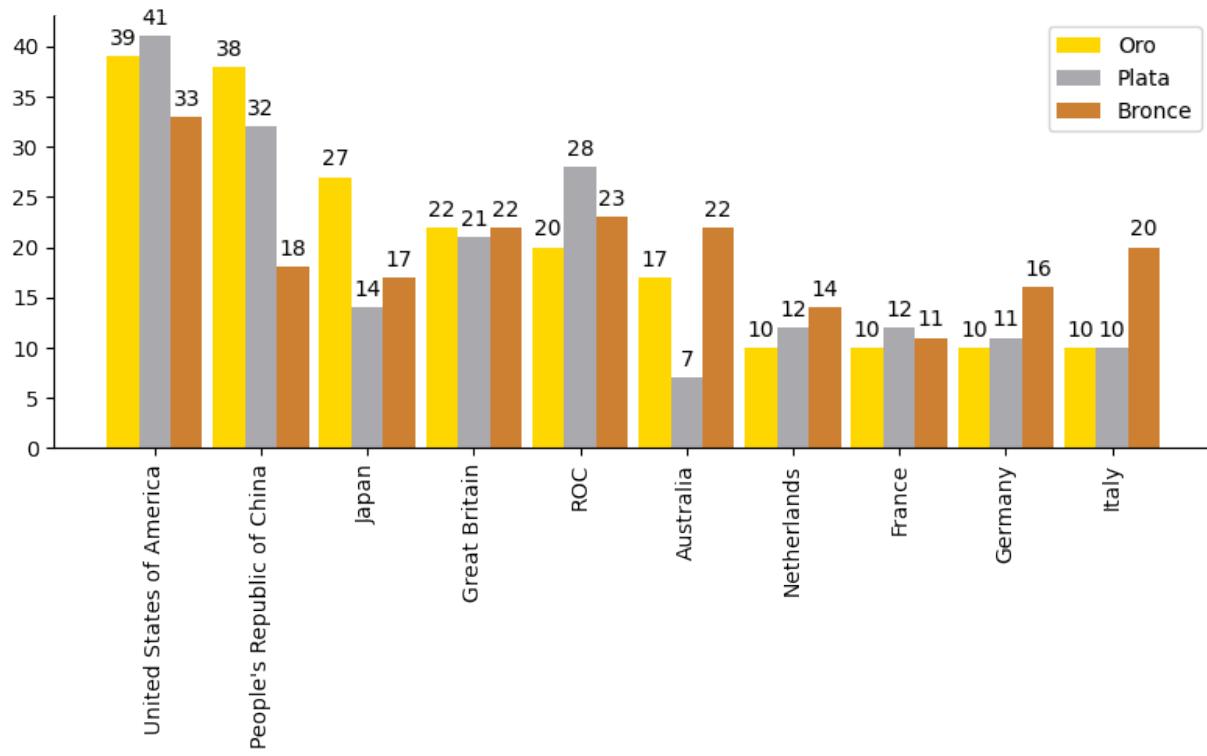
>>> golden_medals = ax.bar(x - bar_width, df_best['Gold'],
...                           bar_width, label='Oro', color='#ffd700')
>>> silver_medals = ax.bar(x, df_best['Silver'],
...                           bar_width, label='Plata', color='#aaa9ad')
>>> bronze_medals = ax.bar(x + bar_width, df_best['Bronze'],
...                           bar_width, label='Bronce', color='#cd7f32')

>>> ax.set_xticks(x)
>>> ax.set_xticklabels(df_best.index, rotation=90)
>>> ax.legend()

>>> # Etiquetas en barras
>>> ax.bar_label(golden_medals, padding=3)
>>> ax.bar_label(silver_medals, padding=3)
>>> ax.bar_label(bronze_medals, padding=3)

>>> ax.spines['right'].set_visible(False) # ocultar borde derecho
>>> ax.spines['top'].set_visible(False) # ocultar borde superior

>>> fig.tight_layout() # ajustar elementos al tamaño de la figura
>>> fig
```



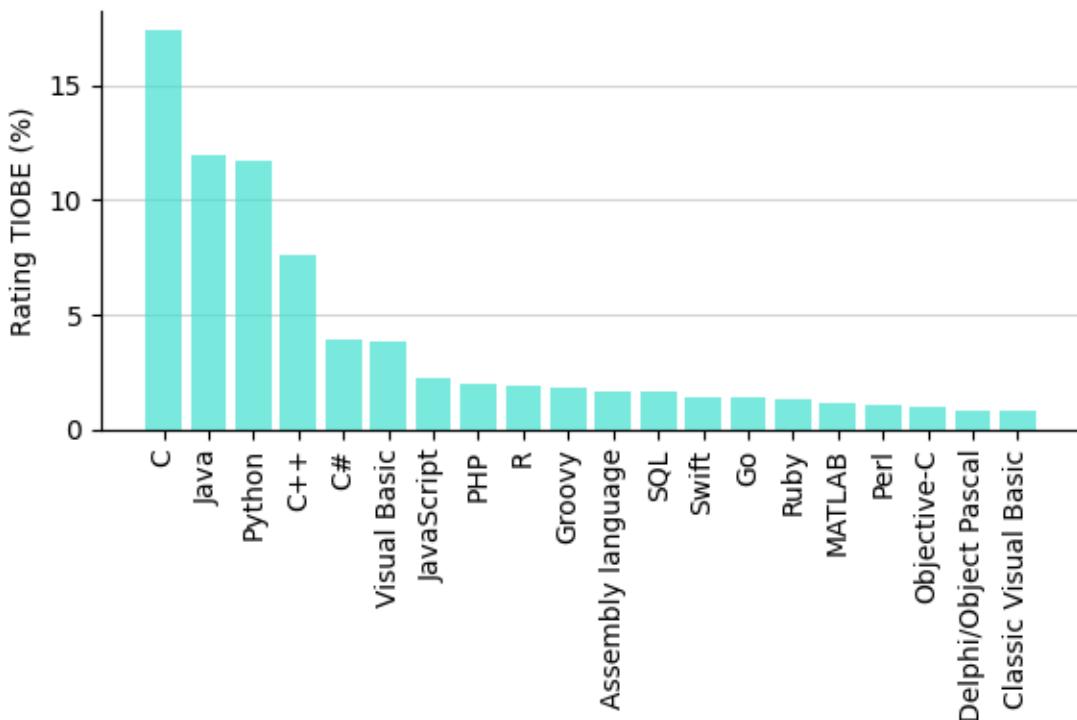
---

## Ejercicio

Partiendo del fichero `tiobe-2020-clean.csv` que contiene las valoraciones de los lenguajes de programación más usados durante el año 2020 (según el índice TIOBE)<sup>8</sup>, cree el siguiente gráfico de barras:

---

<sup>8</sup> Datos extraídos desde [esta](#) página de Kaggle.



## Gráficos de dispersión

Para este gráfico vamos a usar un «dataset» de jugadores de la NBA<sup>5</sup> extraído desde [esta](#) página de Kaggle. El fichero `nba-data.csv` contiene información desde 1996 hasta 2019.

En primer lugar cargamos los datos y nos quedamos con un subconjunto de las columnas:

```
>>> df = pd.read_csv('pypi/datascience/files/nba-data.csv', usecols=['pts', 'reb',
   ↵'ast'])

>>> df.head()
   pts  reb  ast
0  4.8  4.5  0.5
1  0.3  0.8  0.0
2  4.5  1.6  0.9
3  7.8  4.4  1.4
4  3.7  1.6  0.5

>>> df.shape
(11700, 3)
```

<sup>5</sup> National Basketball League (liga estadounidense de baloncesto).

El objetivo es crear un **gráfico de dispersión** en el relacionaremos los puntos anotados con los rebotes capturados, así como las asistencias dadas:

```
>>> fig, ax = plt.subplots(figsize=(8, 6), dpi=100) # 800x600 px

>>> # Crear variables auxiliares
>>> x = df['pts']
>>> y = df['reb']
>>> colors = df['ast']

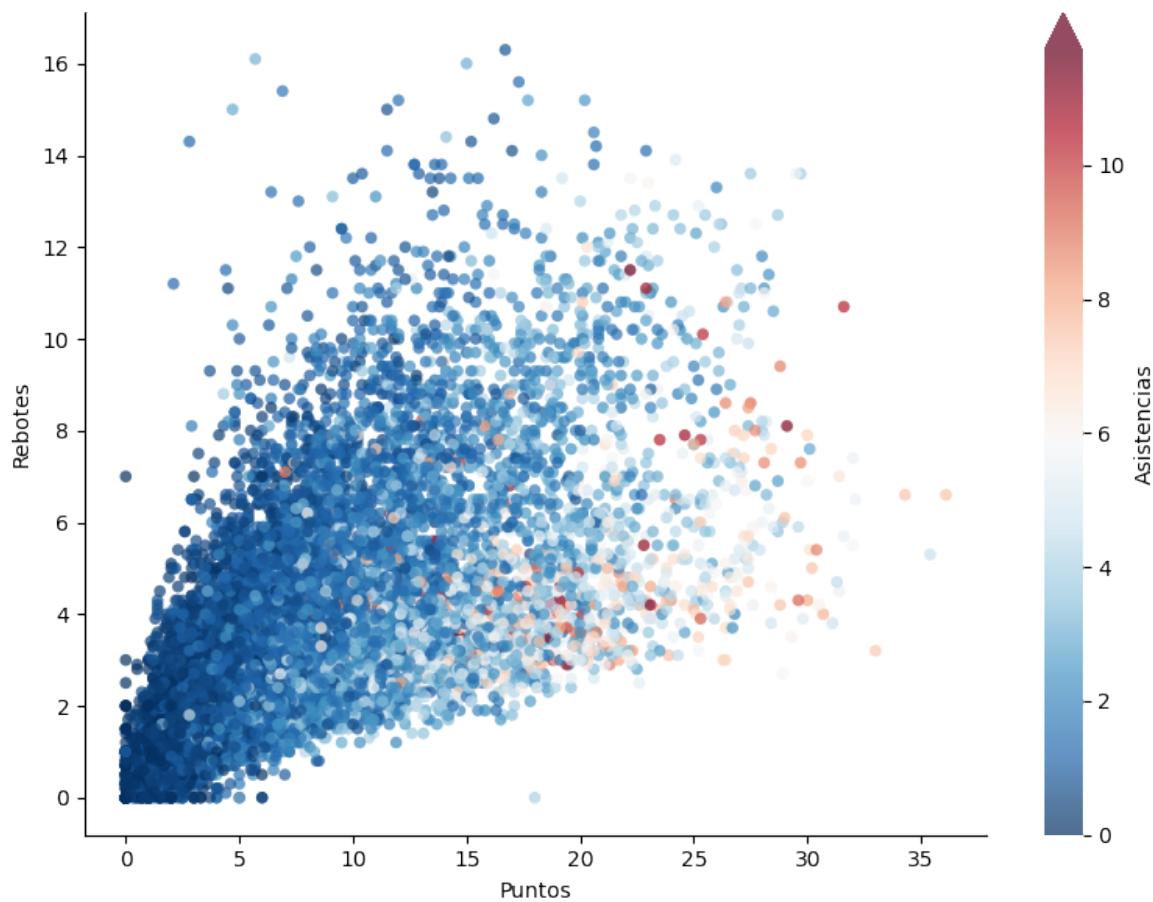
>>> p = ax.scatter(x, y,
...                  s=30, # tamaño de los puntos
...                  c=colors, cmap='RdBu_r', # colores
...                  vmin=colors.min(), vmax=colors.max(), # normalización de colores
...                  alpha=0.7,
...                  edgecolors='none')

>>> # Barra de colores
>>> cb = fig.colorbar(p, ax=ax, label='Asistencias', extend='max')
>>> cb.outline.set_visible(False)

>>> ax.set_xlabel('Puntos')
>>> ax.set_ylabel('Rebotes')

>>> ax.spines['right'].set_visible(False)
>>> ax.spines['top'].set_visible(False)

>>> fig.tight_layout()
```



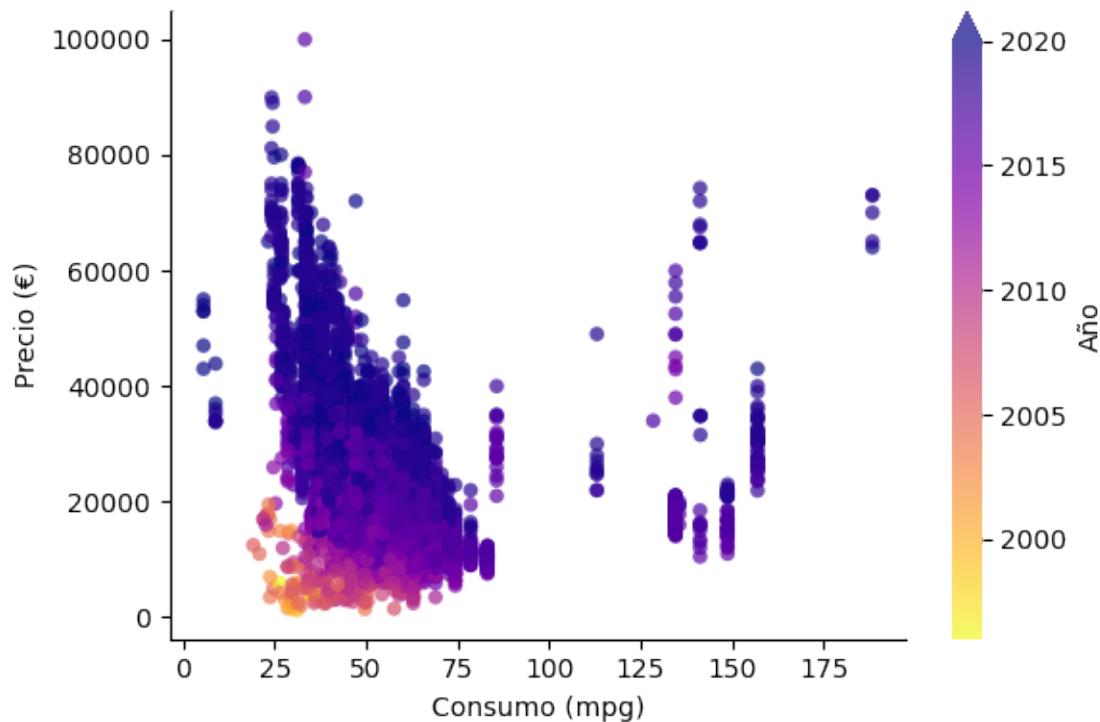
Del gráfico anterior cabe destacar varios aspectos:

- **Normalización:** Cuando aplicamos una estética de color al gráfico basada en los datos de una variable, debemos normalizar dicha variable en el mapa de color («colormap») que elijamos. Para ello, matplotlib nos ofrece la [normalización de mapas de color](#). En el caso concreto de `scatter()` pasariamos esta normalización mediante el parámetro `norm` pero también podemos usar los parámetros `vmin` y `vmax`.
- **Barra de color:** Se trata de una leyenda particular en la que mostramos el gradiente de color vinculado a una determinada estética/variable del gráfico. Matplotlib también nos permite personalizar estas [barras de color](#).

### Ejercicio

Partiendo del fichero `bmw-clean.csv` que contiene información sobre vehículos de la marca BMW<sup>9</sup>, cree el siguiente gráfico de dispersión:

<sup>9</sup> Datos extraídos desde [esta página de Kaggle](#).



El mapa de color que se ha usado es `plasma_r`.

---

## Histogramas

En esta ocasión vamos a trabajar con un «dataset» de «Avengers»<sup>6</sup> extraído desde Kaggle. Hemos descargado el fichero `avengers.csv`.

Como punto de partida vamos a cargar la información y a quedarnos únicamente con la columna que hace referencia al año en el que se crearon los personajes:

```
>>> df = pd.read_csv('pypi/datascience/files/avengers.csv', usecols=['Year'])

>>> df.head()
Year
0 1963
1 1963
2 1963
3 1963
4 1963

>>> df.shape
(173, 1)
```

<sup>6</sup> Los Vengadores son un equipo de superhéroes publicados por Marvel Comics.

Igualmente haremos un pequeño filtrado para manejar sólo registros a partir de 1960:

```
>>> df = df[df['Year'] >= 1960]

>>> df.shape
(159, 1)
```

Ahora ya podemos construir el histograma, que va a representar las **frecuencias absolutas de creación de personajes Marvel según su año de creación**.

Aunque es posible indicar un número determinado de contenedores («bins»), en este caso vamos a especificar directamente los intervalos (cada 5 años):

```
>>> df['Year'].min(), df['Year'].max()
(1963, 2015)

>>> bins = range(1960, 2021, 5)
```

Y a continuación el código necesario para montar el gráfico:

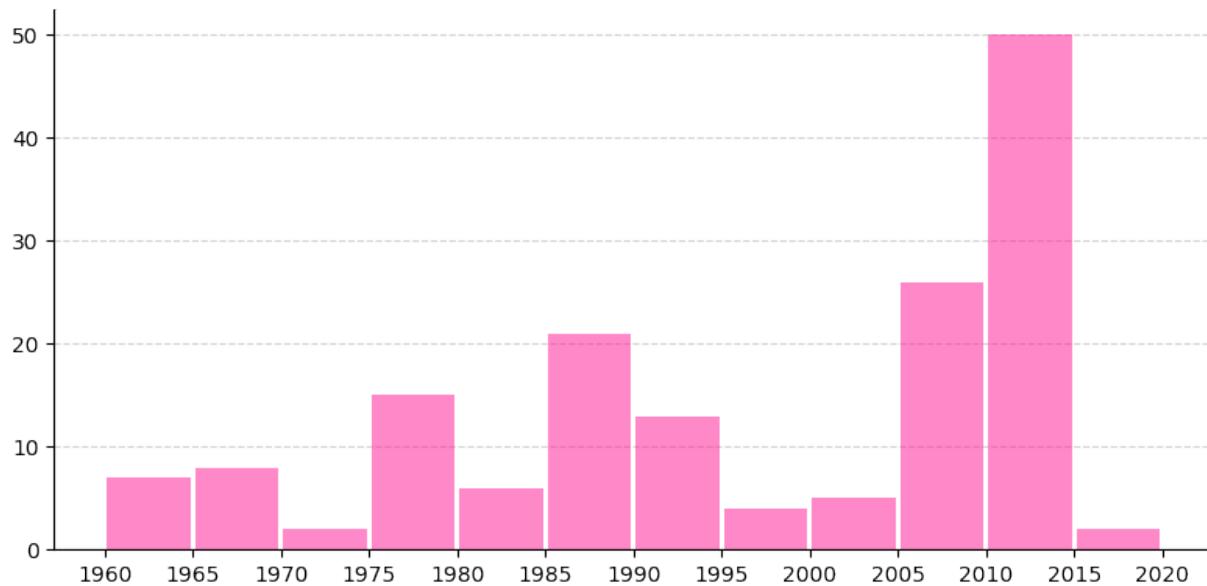
```
>>> fig, ax = plt.subplots(figsize=(8, 4), dpi=100) # 800x400 px

>>> ax.hist(df,
...           bins=bins,      # intervalos de agrupación
...           rwidth=0.95,     # ancho de cada barra
...           zorder=2,        # barras por encima de rejilla
...           color='deeppink',
...           alpha=0.5)

>>> ax.spines['right'].set_visible(False)
>>> ax.spines['top'].set_visible(False)

>>> ax.set_xticks(bins) # etiquetas de intervalos en el eje x
>>> ax.yaxis.grid(color='lightgray', linestyle='--') # rejilla

>>> fig.tight_layout()
```



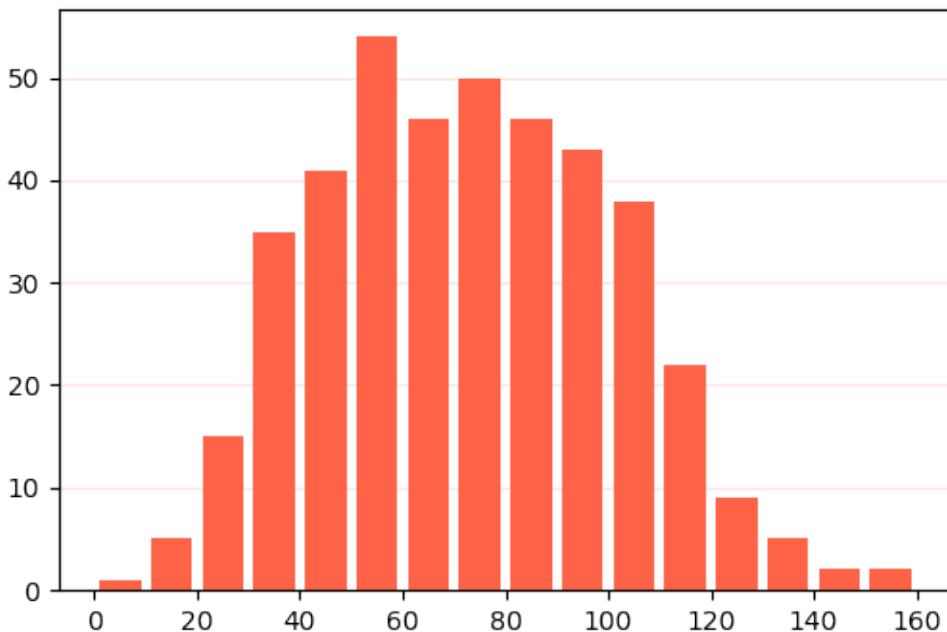
Descargo de responsabilidad: Técnicamente este gráfico no es un histograma ya que los años (fechas en general) no representan categorías válidas, pero sirve a efectos demostrativos de cómo se construyen este tipo de diagramas.

---

### Ejercicio

Partiendo del fichero `pokemon.csv` que contiene información sobre Pokemon<sup>10</sup>, cree el siguiente histograma en el que se analiza el número de personajes «pokemons» en función de su velocidad (columna *Speed*):

<sup>10</sup> Datos extraídos desde [esta página de Kaggle](#).



## Gráficos para series temporales

Vamos a trabajar con un conjunto de datos extraído desde [esta página de Kaggle](#) que contiene información histórica de temperaturas del planeta Tierra. El fichero `global-temperatures.csv` se ha descargado para su tratamiento.

En primer lugar cargamos los datos, renombramos las columnas y eliminamos los valores nulos:

```
>>> df = pd.read_csv('pypi/datascience/files/global-temperatures.csv',
...                     parse_dates=['dt'], # conversión a tipo datetime
...                     usecols=['dt', 'LandAverageTemperature'])

>>> df.rename(columns={'dt': 'when', 'LandAverageTemperature': 'temp'}, inplace=True)
>>> df.dropna(inplace=True)

>>> df.head()
      when      temp
0 1750-01-01    3.034
1 1750-02-01    3.083
2 1750-03-01    5.626
3 1750-04-01    8.490
4 1750-05-01   11.573
```

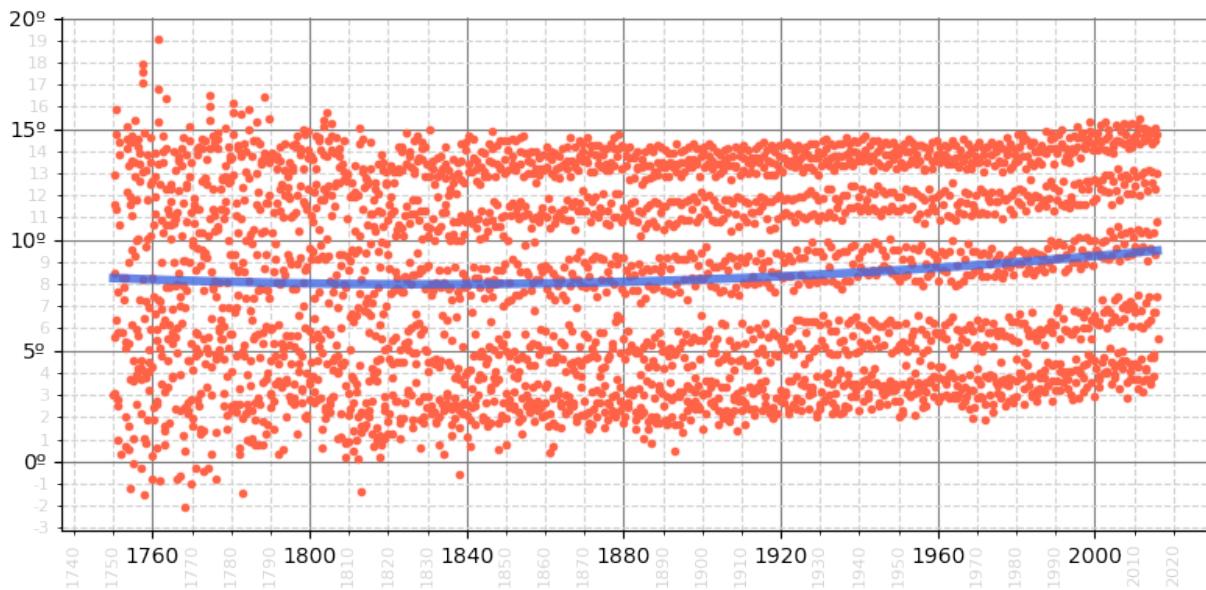
(continué en la próxima página)

(provien de la página anterior)

```
>>> df.shape  
(3180, 2)
```

A continuación montamos un gráfico en el que se representan todas las **mediciones históricas de la temperatura media global del planeta** y añadimos una línea de tendencia:

```
>>> # Necesitamos algunas utilidades de gestión de fechas  
>>> from matplotlib.dates import YearLocator, DateFormatter, date2num  
>>> from matplotlib.ticker import MultipleLocator  
  
>>> fig, ax = plt.subplots(figsize=(8, 4), dpi=100) # 800x400 px  
  
>>> # Alias para simplificar el acceso  
>>> x = df.when  
>>> y = df.temp  
  
>>> ax.plot(df.when, df.temp,  
...           linestyle='None', marker='.', color='tomato', # estilo de línea  
...           zorder=2) # orden para colocar sobre rejilla  
  
>>> # Construcción de la línea de tendencia  
>>> x = date2num(x)  
>>> z = np.polyfit(x, y, 2) # ajuste polinómico de grado 2  
>>> p = np.poly1d(z)  
>>> plt.plot(x, p(x), linewidth=4, alpha=0.8, color='royalblue')  
  
>>> # Formateo de los ejes  
>>> ax.xaxis.set_minor_locator(YearLocator(10))  
>>> ax.xaxis.set_minor_formatter(DateFormatter('%Y'))  
>>> ax.tick_params(axis='x', which='minor',  
...                  labelsize=8, labelcolor='lightgray', rotation=90)  
>>> ax.xaxis.grid(which='minor', color='lightgray', linestyle='dashed')  
>>> ax.yaxis.set_major_formatter('{x:.0f}º')  
>>> ax.yaxis.set_minor_locator(MultipleLocator(1))  
>>> ax.tick_params(axis='y', which='minor',  
...                  labelsize=8, labelcolor='lightgray')  
>>> ax.yaxis.grid(which='minor', linestyle='dashed', color='lightgray')  
>>> ax.yaxis.set_minor_formatter('{x:.0f}º')  
>>> ax.tick_params(axis='y', which='minor', labelsize=8, labelcolor='lightgray')  
  
>>> ax.spines['right'].set_visible(False)  
>>> ax.spines['top'].set_visible(False)  
  
>>> fig.tight_layout()
```



## Mapas de calor

Para este tipo de gráfico vamos a utilizar un «dataset» que recoge las 1000 películas más valoradas en IMDB<sup>7</sup>. Está sacado desde [esta página de Kaggle](#) y se ha descargado el fichero de datos en `imdb-top-1000.csv`.

En primer lugar vamos a cargar los datos quedándonos con las columnas *Certificate* (clasificación de la película según edades), *Genre* (géneros de la película) e *IMDB\_Rating* (valoración de la película en IMDB):

```
>>> df = pd.read_csv('pypi/datascience/files/imdb-top-1000.csv',
...                     usecols=['Certificate', 'Genre', 'IMDB_Rating'])

>>> df.head()
   Certificate          Genre  IMDB_Rating
0           A        Drama      9.3
1           A  Crime, Drama      9.2
2          UA  Action, Crime, Drama      9.0
3           A  Crime, Drama      9.0
4           U  Crime, Drama      9.0
```

Ahora creamos una nueva columna en el DataFrame donde guardaremos únicamente el género principal de cada película:

```
>>> df['Main_Genre'] = df['Genre'].str.split(',', expand=True)[0]
```

(continué en la próxima página)

<sup>7</sup> IMDB es una reconocida página web que contiene valoraciones sobre películas y series.

(provine de la página anterior)

```
>>> df.head()
   Certificate      Genre  IMDB_Rating  Main_Genre
0            A        Drama         9.3     Drama
1            A  Crime, Drama         9.2     Crime
2           UA  Action, Crime, Drama         9.0   Action
3            A  Crime, Drama         9.0     Crime
4            U  Crime, Drama         9.0     Crime
```

A continuación agrupamos y obtenemos los valores medios de las valoraciones:

```
>>> # unstack permite disponer la agrupación en forma tabular (para el heatmap)
>>> ratings = df.groupby(['Certificate', 'Main_Genre'])['IMDB_Rating'].mean().
>>> unstack()

>>> # Nos quedamos con un subconjunto de certificados y géneros
>>> review_certificates = ['U', 'UA', 'PG-13', 'R', 'A']
>>> review_genres = ['Animation', 'Action', 'Adventure', 'Biography',
...                   'Comedy', 'Crime', 'Drama']
>>> ratings = ratings.loc[review_certificates, review_genres]

>>> # Recodificamos los certificados (clasificación) con códigos más entendibles
>>> certs_description = {'U': 'ALL', 'UA': '>12', 'PG-13': '>13', 'R': '>17', 'A': '>
>>> 18'}
>>> ratings.index = ratings.reset_index()['Certificate'].replace(certs_description)

>>> ratings
   Main_Genre  Animation  Action  Adventure  Biography  Comedy  Crime  Drama
   Certificate
   ALL          7.947368  8.165000  7.953571  7.862500  7.940541  8.200000  7.976364
   >12          7.883333  7.992424  7.958333  7.971429  7.885714  7.900000  7.953659
   >13          7.866667  7.783333  7.600000  7.862500  7.785714  8.000000  7.775000
   >17          7.800000  7.812500  7.900000  7.900000  7.824138  7.814286  7.915094
   >18          7.866667  7.873171  7.912500  8.017647  7.877778  8.130233  8.036364
```

Ahora ya podemos construir el mapa de calor usando el DataFrame `ratings` generado previamente:

```
>>> fig, ax = plt.subplots(figsize=(8, 4), dpi=100)

>>> text_colors = ('black', 'white')
>>> im = ax.imshow(ratings, cmap='Reds') # mapa de calor
>>> cbar = fig.colorbar(im, ax=ax, label='IMDB Rating') # leyenda
>>> cbar.outline.set_visible(False)

>>> x = ratings.columns
```

(continué en la próxima página)

(provien de la página anterior)

```

>>> y = ratings.index

>>> # Mostrar las etiquetas. El color del texto cambia en función de su normalización
>>> for i in range(len(y)):
...     for j in range(len(x)):
...         value = ratings.iloc[i, j]
...         text_color = text_colors[int(im.norm(value) > 0.5)] # color etiqueta
...         ax.text(j, i, f'{value:.2f}', color=text_color, va='center', ha='center')

>>> # Formateo de los ejes
>>> ax.set_xticks(range(len(x)))
>>> ax.set_xticklabels(x, rotation=90)
>>> ax.set_yticks(range(len(y)))
>>> ax.set_yticklabels(y)
>>> ax.invert_yaxis()

>>> ax.spines[:].set_visible(False)

>>> fig.tight_layout()

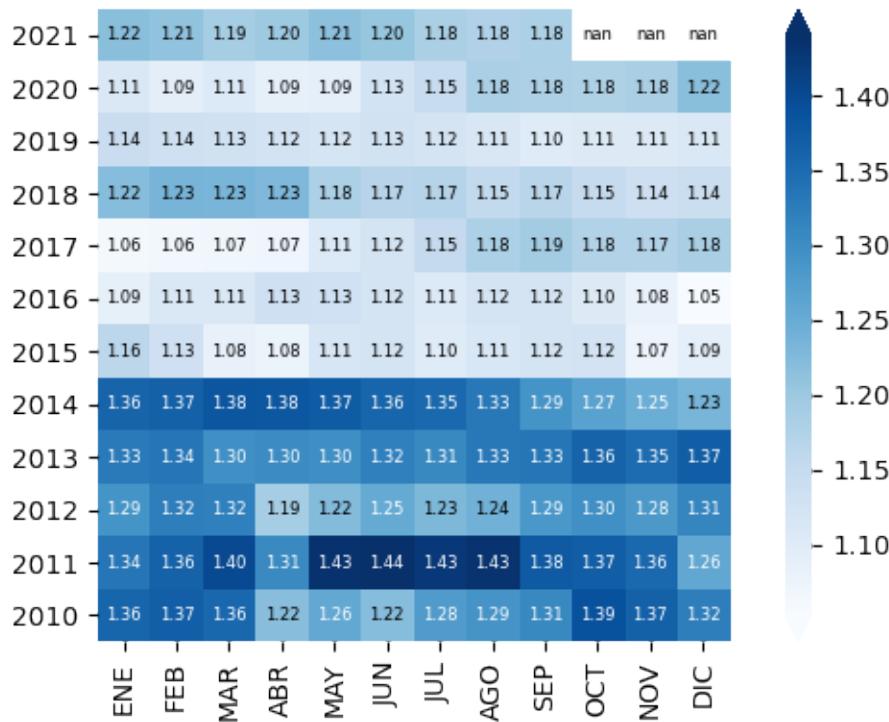
```



## Ejercicio

Partiendo del fichero `euro-dollar-clean.csv` que contiene información sobre el cambio

euro-dollar durante los últimos 12 años<sup>11</sup>, cree el siguiente mapa de calor en el que se analiza la evolución del cambio enfrentando meses y años:



## Diagramas de caja

Para mostrar el funcionamiento de los diagramas de caja en Matplotlib vamos a hacer uso de distintas distribuciones aleatorias que crearemos mediante funciones de Numpy:

```
>>> DIST_SIZE = 100 # tamaño de la muestra

>>> boxplots = []

>>> boxplots.append(dict(
...     dist=np.random.normal(0, 1, size=DIST_SIZE),
...     label='Normal\n$\mu=0, \sigma=1$',
...     fill_color='pink',
...     brush_color='deeppink'))

>>> boxplots.append(dict(
...     dist=np.random.geometric(0.4, size=DIST_SIZE),
...     label='Geometric\n$p=0.4$',
```

(continué en la próxima página)

<sup>11</sup> Datos extraídos desde [esta](#) página de Kaggle.

(proviene de la página anterior)

```

...     fill_color='lightblue',
...     brush_color='navy'))

>>> boxplots.append(dict(
...     dist=np.random.chisquare(2, size=DIST_SIZE),
...     label='Chi-squared\n$df=2$',
...     fill_color='lightgreen',
...     brush_color='darkgreen'))

```

Ahora ya podemos construir el gráfico de cajas que nos permite visualizar la distribución de las muestras:

```

>>> fig, ax = plt.subplots(figsize=(8, 6), dpi=100) # 800x600 px

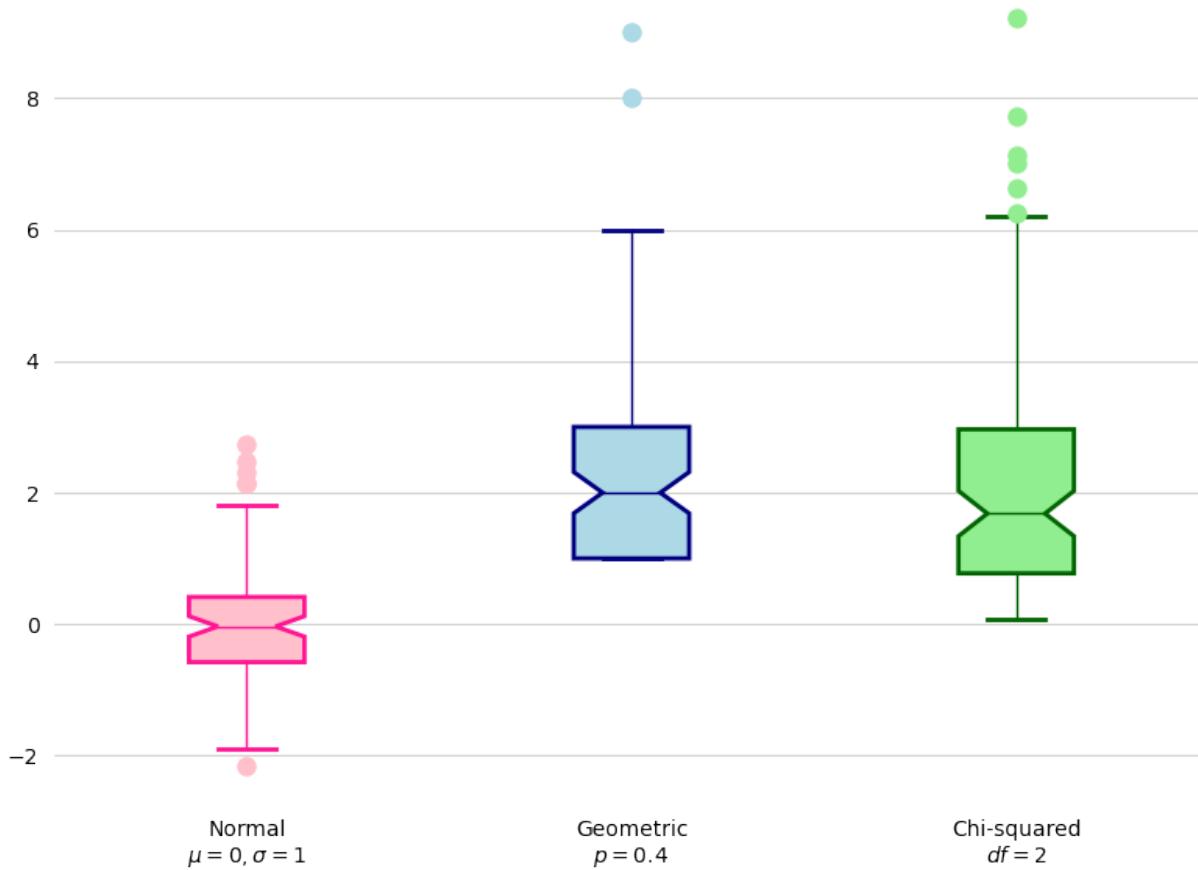
>>> for i, boxplot in enumerate(boxplots):
...     fcolor, bcolor = boxplot['fill_color'], boxplot['brush_color']
...     ax.boxplot(boxplot['dist'],
...                labels=[boxplot['label']],
...                positions=[i],
...                widths=[.3],
...                notch=True,
...                patch_artist=True,
...                boxprops=dict(edgecolor=bcolor,
...                             facecolor=fcolor,
...                             linewidth=2),
...                capprops=dict(color=bcolor, linewidth=2),
...                flierprops=dict(color=bcolor,
...                               markerfacecolor=fcolor,
...                               linestyle='none',
...                               markeredgecolor='none',
...                               markersize=9),
...                medianprops=dict(color=bcolor),
...                whiskerprops=dict(color=bcolor,
...                                  linewidth=1))

>>> ax.yaxis.grid(color='lightgray')
>>> ax.xaxis.set_ticks_position('none')
>>> ax.yaxis.set_ticks_position('none')

>>> ax.spines[:].set_visible(False)

>>> fig.tight_layout()

```



---

**Consejo:** El código para preparar el gráfico se ha complicado porque se ha incidido en mejorar la estética. En cualquier caso, una vez hecho, se puede refactorizar en una función y reutilizarlo para futuros trabajos.

---

### Gráficos de evolución

Partiendo de un conjunto de datos temporales, vamos a aprovechar para elaborar un gráfico de evolución del precio de criptomonedas. En esta ocasión hemos utilizado el «dataset» `eth-usd.csv` descargado desde [esta página de Kaggle](#). Contiene la valoración de la criptomoneda **Ethereum** en función de una marca temporal, así como el volumen de «moneda» existente en cada momento.

El objetivo será crear un **gráfico que represente el valor de la criptomoneda (a lo largo del tiempo) en contraposición al volumen de unidades**.

Lo primero que haremos, además de cargar los datos, será lo siguiente:

- Seleccionar las columnas *Date* (fecha de referencia), *Open* (precio de la moneda a la

apertura) y *Volume* (volumen de moneda).

- Parsear el campo fecha.
- Filtrar sólo aquellos registros a partir del 1 de enero de 2017 (por simplicidad).
- Dividir la columna de volumen por 10M de cara a equiparar cantidades con la valoración (ajuste de gráfico).
- Aplicar una media móvil para suavizar las curvas a representar.

```
>>> import datetime

>>> df = pd.read_csv('pypi/datascience/files/eth-usd.csv',
...                     parse_dates=['Date'],
...                     usecols=['Date', 'Open', 'Volume'],
...                     index_col='Date')

>>> min_date = datetime.datetime(year=2017, month=1, day=1)
>>> df = df.loc[df.index > min_date]

>>> df['Volume'] /= 1e7

>>> df_smooth = df.rolling(20).mean().dropna()

>>> df_smooth.head()
          Open      Volume
Date
2017-01-21    9.968611   2.146882
2017-01-22   10.105573   2.117377
2017-01-23   10.222339   1.985587
2017-01-24   10.273270   1.821968
2017-01-25   10.239854   1.647938
```

Ahora ya podemos montar el gráfico dedicando algo de esfuerzo a la parte estética:

```
>>> fig, ax = plt.subplots(figsize=(8, 4), dpi=100) # 800x400px

>>> # Alias para facilitar el acceso
>>> x = df_smooth.index
>>> y_open = df_smooth['Open']
>>> y_vol = df_smooth['Volume']

>>> # Líneas de evolución
>>> ax.plot(x, y_open, label='Value ($)', color='skyblue', linewidth=1.5)
>>> ax.plot(x, -y_vol, label='Volume (10M ud.)', color='pink', linewidth=1.5)
>>> # Relleno del área
>>> plt.fill_between(x, y_open, alpha=0.5, color='skyblue', zorder=3)
```

(continué en la próxima página)

(provine de la página anterior)

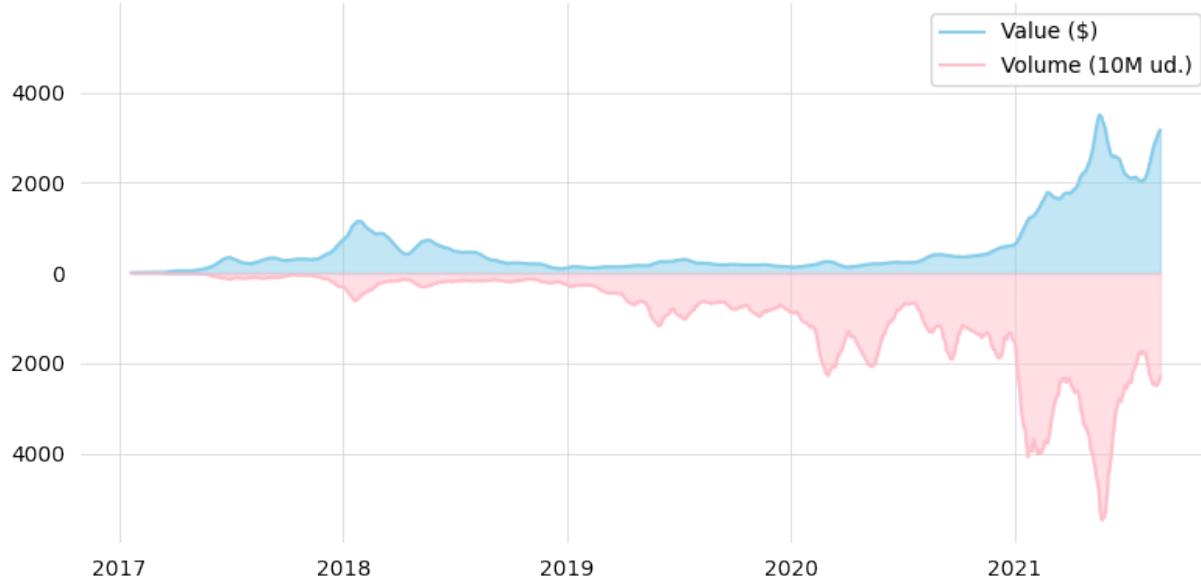
```
>>> plt.fill_between(x, -y_vol, alpha=0.5, color='pink', zorder=3)

>>> # Formateo de los ejes
>>> ax.xaxis.set_ticks_position('none')
>>> ax.yaxis.set_ticks_position('none')
>>> y_ticks = [-4000, -2000, 0, 2000, 4000]
>>> y_tick_labels = ['4000', '2000', '0', '2000', '4000']
>>> ax.set_yticks(y_ticks)
>>> ax.set_yticklabels(y_tick_labels)
>>> ax.set_ylim(-6000, 6000)

>>> # Rejilla
>>> ax.xaxis.grid(color='lightgray', linewidth=.5)
>>> for y_tick in y_ticks:
...     if y_tick != 0:
...         ax.axhline(y_tick, color='lightgray', linewidth=.5)

>>> ax.legend()
>>> ax.spines[:].set_visible(False)

>>> fig.tight_layout()
```



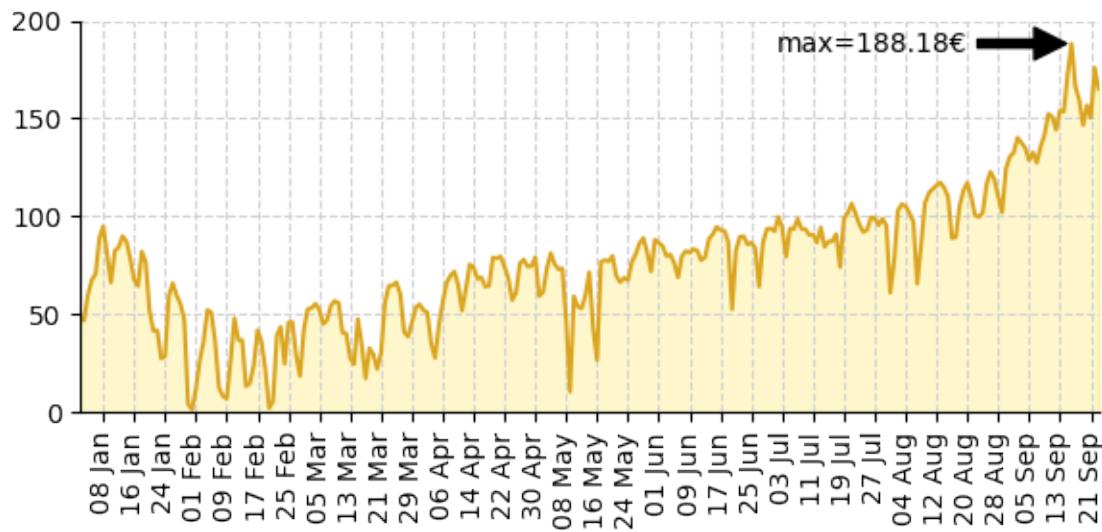
---

## Ejercicio

Partiendo del fichero `mwh-spain-2021-clean.csv` que contiene información sobre el precio de la energía en España durante el año 2021<sup>12</sup>, cree el siguiente diagrama de evolución que

<sup>12</sup> Datos extraídos desde [esta página de El País](#).

representa la variación del precio del MWh<sup>13</sup> en función del tiempo:



Las marcas (en el eje x) tienen una separación de 10 días.

<sup>13</sup> Mega Watio Hora (medida de consumo de energía)