

Introducción al lenguaje de programación con Python



FONDO EUROPEO DE DESARROLLO REGIONAL



INDICE

Capítulo 1: Introducción	2
1. Hablando con la máquina	2
2. Algo de historia	6
3. Python.....	11
Capítulo 2: Entorno de desarrollo	18
1. Thonny.....	18
2. Contexto real.....	24
Capítulo 3: Tipos de datos	30
1. Datos.....	30
2. Números	39
3. Cadenas de texto.....	47
Capítulo 4: Control de flujo	59
1. Condicionales.....	59
2. Bucles	66
Capítulo 5: Estructuras de datos.....	74
1. Listas.....	74
2. Tuplas.....	91
3. Diccionarios	95
4. Conjuntos.....	107
5. Ficheros	112
Capítulo 6: Modularidad.....	118
1. Funciones.....	118
2. Módulos	128

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. Hablando con la máquina

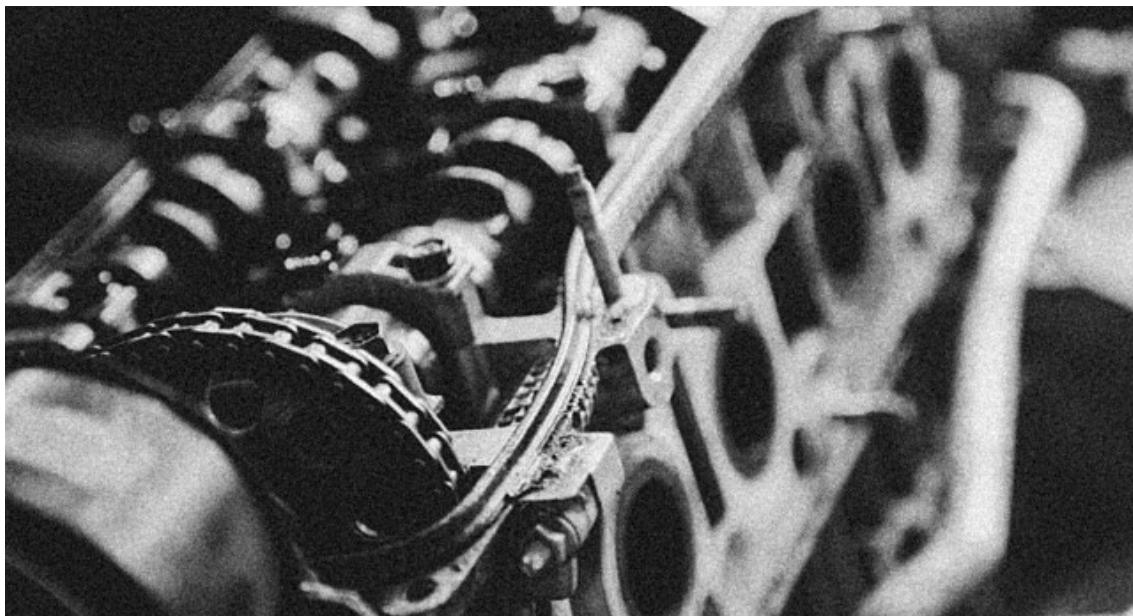


Foto original por [Garett Mizunaka](#) en Unsplash

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**.

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 0111011 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
```

1.2 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»:

```
global _start

section .text
_start: mov    rax, 1           ; system call for write
        mov    rdi, 1           ; file handle 1 is stdout
        mov    rsi, message     ; address of string to output
        mov    rdx, 13          ; number of bytes
        syscall
        mov    rax, 60          ; system call for exit
        xor    rdi, rdi         ; exit code 0
        syscall

section .data
message: db    "Hello, World", 10 ; note the newline at the end
```

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](#).

1.3 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;
}
```

1.4 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](#).

1.5 Compiladores

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

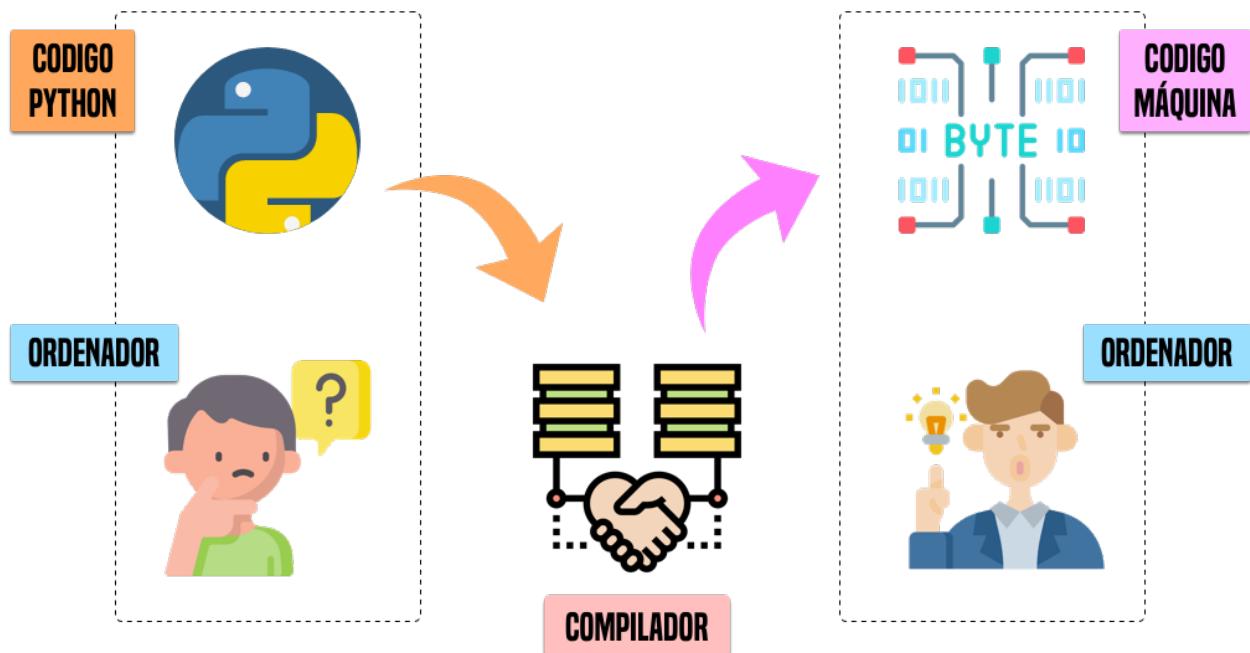


Figura 1: Esquema de funcionamiento de un compilador. Iconos originales por [Flaticon](#)

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.

2. Algo de historia



Foto original por [Dario Veronesi](#) en Unsplash

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 1970.¹

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 operaciones básicas	XV
Charles Babbage	Máquina diferencial para el cálculo de polinomios	XVII
Ada Lovelace	Matemática, informática y escritora británica. Primera programadora de la historia por el desarrollo de algoritmos para la máquina analítica de Babbage	XVII
Herman Hollerit	Creador de un sistema para automatizar la pesada tarea del censo	1890
Alan Turing	Máquina de Turing - una máquina capaz de resolver problemas - Aportes de Lógica Matemática - Computadora con tubos de vacío	1943
George Boole	Contribuyó al algebra binaria y a los sistemas de circuitos de computadora (álgebra booleana)	1854
John Atanasoff	Primera computadora digital electrónica patentada: Atanasoff Berry Computer (ABC)	1942
Howard Aiken	En colaboración con IBM desarrolló el Mark I , 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 programadora que utilizó el Mark I	1945
John W. Mauchly	Junto a John Presper Eckert desarrolló una computadora electrónica completamente operacional a gran escala llamada Electronic Numerical Integrator And Computer (ENIAC)	1946
John Von Neumann	Propuso guardar en memoria no solo la información , sino también los programas , 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.

2.2 De los computadores a la programación

De acuerdo con 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.



Figura 2: Ada Lovelace: primera programadora de la historia. Fuente: [Meatze](#)

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.¹

¹ Fuente: [Universidad Técnica del Norte](#)

2.3 Cronología de lenguajes de programación

Desde la década de los 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:

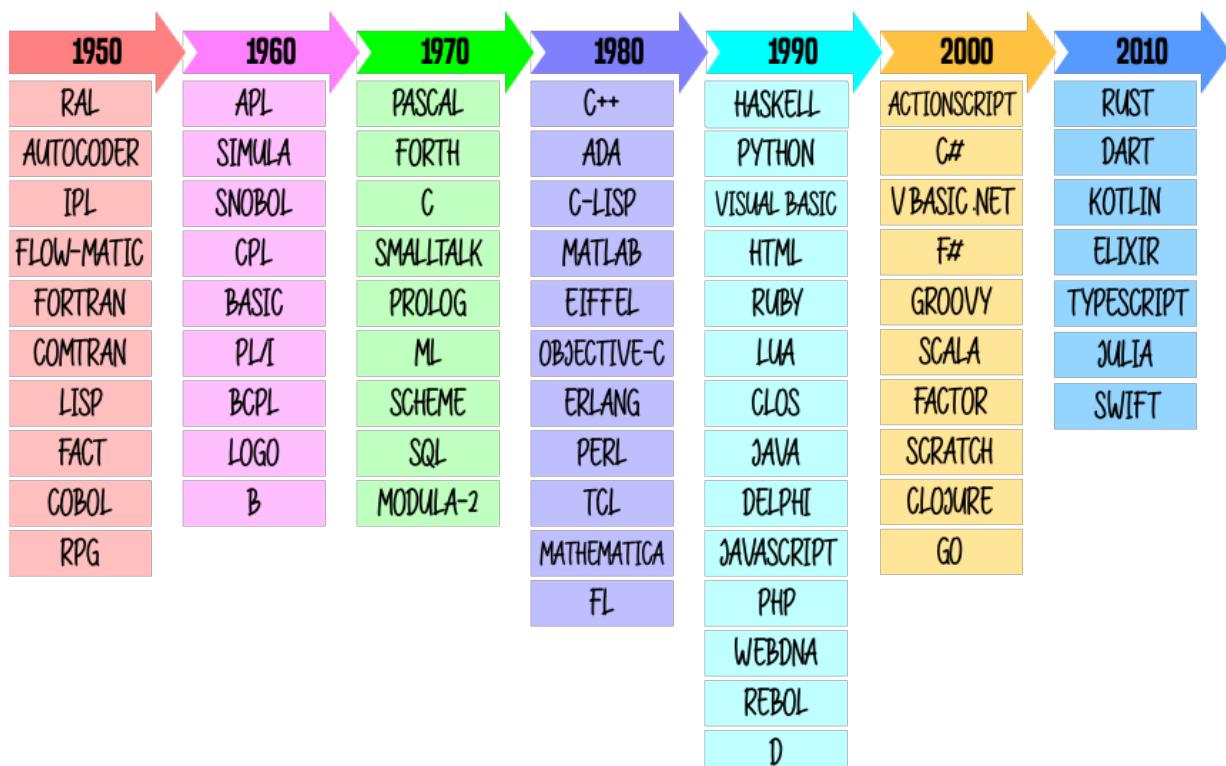


Figura 3: Cronología de los lenguajes de programación más destacados

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](#).

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:²

² Fuente: [Wikipedia](#).

Tabla 2: Creadores y creadoras de lenguajes de programación

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

3. Python



Foto original por [Markéta Marcellová](#) en Unsplash.

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.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**.

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 incluidas»).
- Gran cantidad de librerías de terceros.
- Extensible (C++, C,) y «embebible».
- Gran comunidad, amplio soporte.
- Interpretado.
- Fuertemente tipado, tipado dinámico.
- Hay diferentes implementaciones: CPython, 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.

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.
- Concurrency.
- 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.
- Test 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:



Figura 4: Grandes empresas y organismos que usan Python

Existen ránkings 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 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*.

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:³

³ Fuente: python.org

Tabla 3: Versiones mayores de Python

Versión	Fecha de lanzamiento
1.4	Octubre 1996
1.5	Febrero 1998
1.6	Septiembre 2000
2.0	Octubre 2000
2.1	Abril 2001
2.2	Diciembre 2001
2.3	Julio 2003
2.4	Noviembre 2004
2.5	Septiembre 2006
2.6	Octubre 2008
2.7	Julio 2010
3.0	Diciembre 2008
3.1	Junio 2009
3.2	Febrero 2011
3.3	Septiembre 2012
3.4	Marzo 2014
3.5	Septiembre 2015
3.6	Diciembre 2016
3.7	Junio 2018
3.8	Octubre 2019

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*»⁴, 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 Python3.
- Si vas a desarrollar una aplicación, deberías hacerlo directamente en Python 3.

Importante: Únete a [Python 3](#) y aprovecha todas sus ventajas.

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

3.4 Zen de Python

Existen una serie de *reglas* «filosóficas» que indican una manera de hacer y de pensar dentro del mundo **pitónico**⁵ 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.
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.

⁵ Dícese de algo/alguién que sigue las convenciones de Python

- 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: Entorno 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.

1. Thonny



Foto original de portada por [freddie marriage](#) en Unsplash

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.

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 parecido a la siguiente imagen:

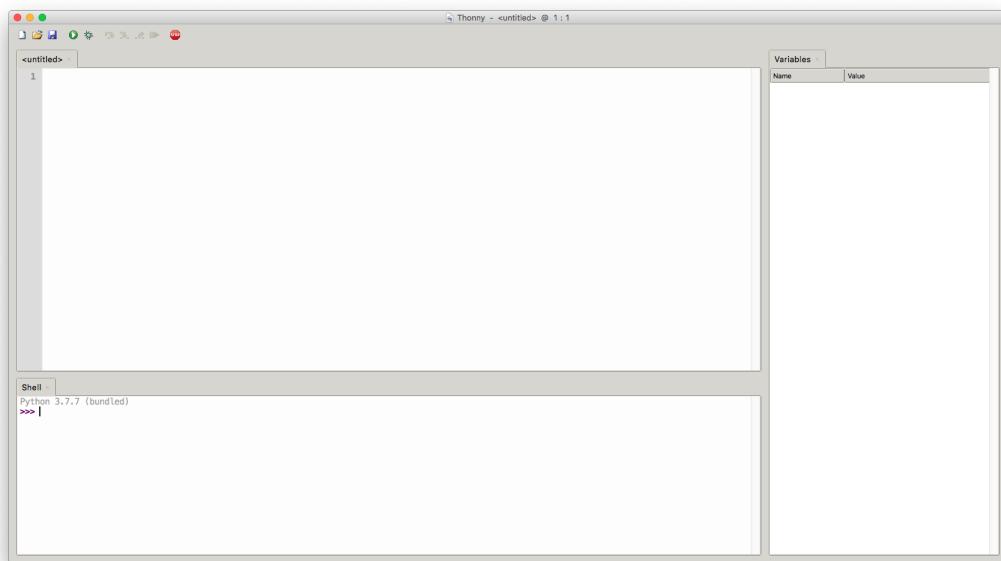


Figura 5: Aspecto de Thonny al arrancarlo

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:

Tabla 4: Versiones de Python

Python 1.0	Jan 1994
Python 1.5	Dec 1997
Python 1.6	Sep 2000
Python 2.0	Oct 2000
Python 2.1	Apr 2001
Python 2.2	Dec 2001
Python 2.3	Jul 2003
Python 2.4	Nov 2004
Python 2.5	Sep 2006
Python 2.6	Oct 2008
Python 2.7	Jul 2010
Python 3.0	Dec 2008
Python 3.1	Jun 2009
Python 3.2	Feb 2011
Python 3.3	Sep 2012
Python 3.4	Mar 2014
Python 3.5	Sep 2015
Python 3.6	Dec 2016
Python 3.7	Jun 2018
Python 3.8	Oct 2019

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

1.2 Probando el intérprete

El intérprete de Python (por lo general) se identifica claramente porque posee un **prompt**⁶ con tres ángulos 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:

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 primera prueba 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 >>>*).

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 guardarla 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`⁷:

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

⁶ Término inglés que se refiere al símbolo que precede la línea de comandos

⁷ La carpeta donde se guarden los archivos de código no es crítica para su ejecución, pero sí es importante mantener un orden y una organización para tener localizados nuestros ficheros y proyectos

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.

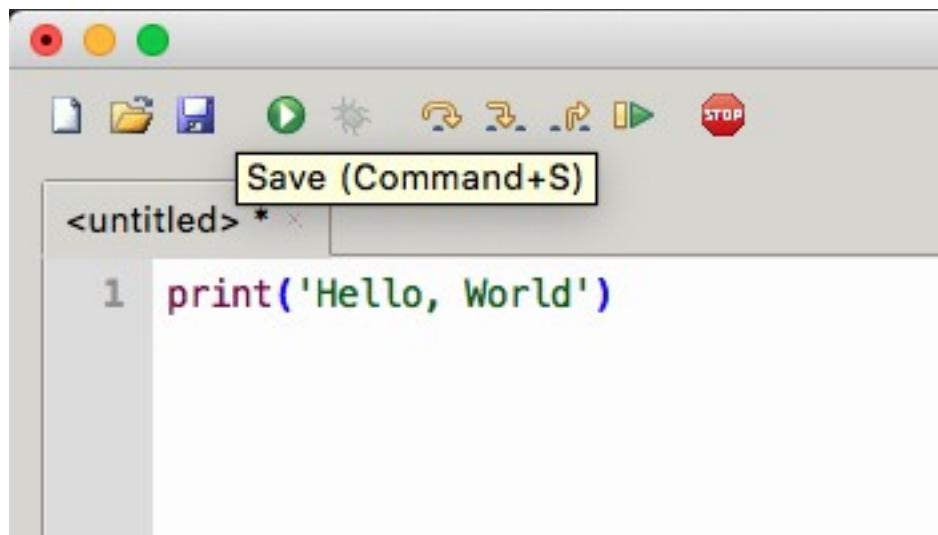


Figura 6: Guardando nuestro primer programa en Python

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 incorporar 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:

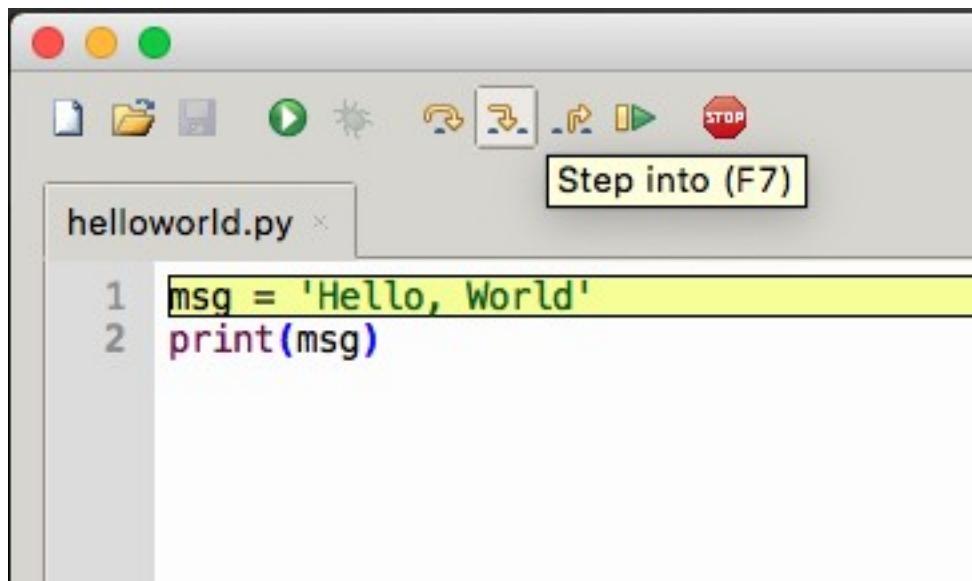


Figura 7: Depurando nuestro primer programa en Python

2. Contexto real



Foto original de portada por [SpaceX](#) en Unsplash

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](#).

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](#)

2.2 Gestión de paquetes

La instalación limpia⁸ 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:

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

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

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.

El paquete 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:

⁸ También llamada «vanilla installation» ya que es la que viene por defecto y no se hace ninguna personalización

```
$ 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
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 > ls sitepackages
__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
- \$ ls -l sitepackages: lista los paquetes instalados en el entorno virtual activo

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 **IDE⁹**. 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)

⁹ Integrated Development Environment.

Editores o IDEs específicos para Python

- [PyCharm](#)
- [Spyder](#)
- [Thonny](#)

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

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 de datos o «machine-learning»¹⁰.

Podemos verlo como un intérprete de Python (contiene un «kernel»¹¹ 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.

2.6 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 para preparar y gestionar el entorno de desarrollo Python. Durante mucho tiempo esto fue difícil de conseguir hasta que *Microsoft* sacó WSL.

¹⁰ Término inglés utilizado para hacer referencia a algoritmos de aprendizaje automático.

¹¹ Proceso específico para un lenguaje de programación que ejecuta instrucciones y actúa como interfaz de entrada/salida.

[WSL](#)¹² 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 aparte para 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 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 18.04 que usaremos:

```
$ Invoke-WebRequest -Uri https://aka.ms/wsl-ubuntu-1804 -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*.

¹² Windows Subsystem for Linux

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.

1. Datos



Foto original de portada por [Alexander Sinn](#) en Unsplash

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.¹

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.
- Un **número de referencias** que rastrea las veces que se usa un objeto.

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:

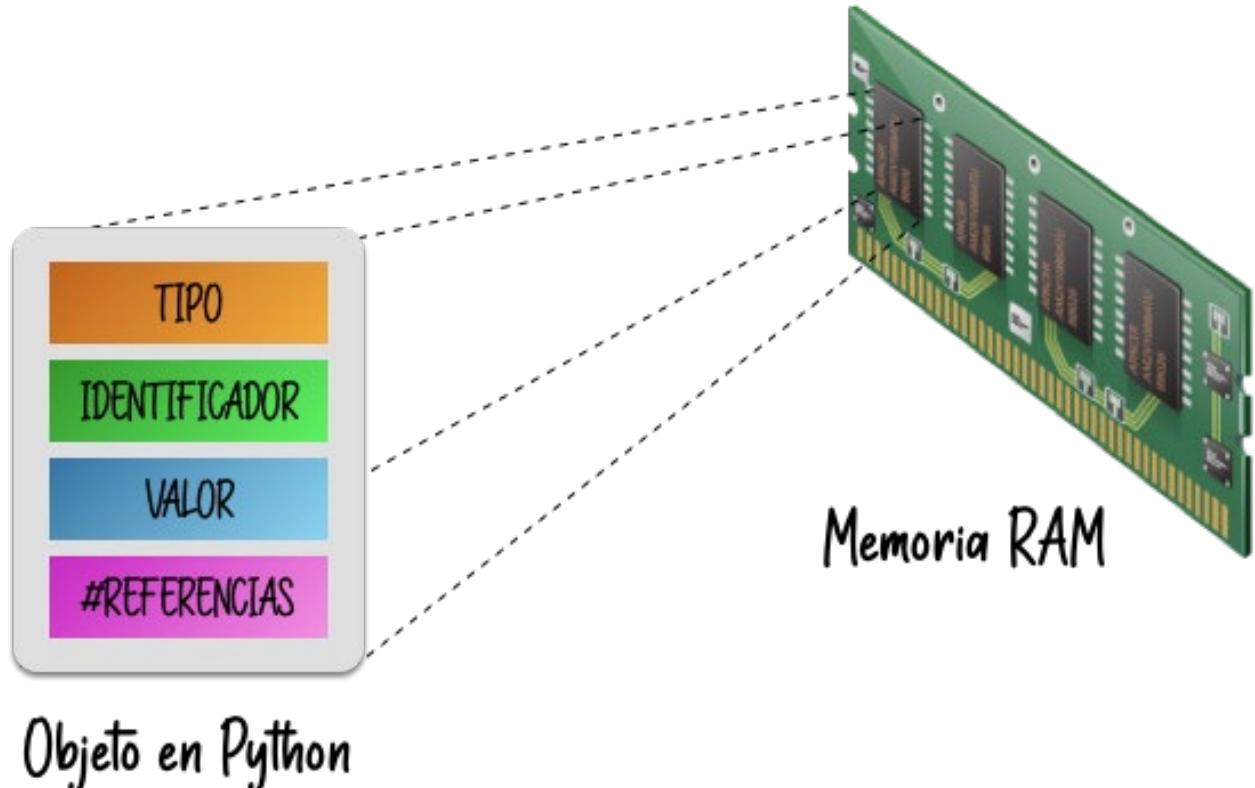


Figura 8: Esquema (*metadatos*) de un objeto en Python

Tabla 5: 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'}

1.2 Variables

Las **variables** son un concepto clave en los lenguajes de programación y permiten definir **nombres** para los **valores** que tenemos en memoria y que vamos a usar en nuestro programa.

nombre = valor

Reglas para nombrar variables

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

- Sólo pueden contener los siguientes caracteres¹³:
 - Letras minúsculas (de la A a la z).
 - Letras mayúsculas (de la A a la Z).
 - Dígitos (del 0 al 9).
 - Guiones bajos (_).
- Son «case-sensitive»¹⁴. Por ejemplo, thing, Thing y THING son nombres diferentes. Deben **empezar con una letra o un guion bajo**, nunca con un dígito.
- No pueden ser una palabra reservada del lenguaje («keywords»).

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
的帮助.

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
```

¹³ 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.

¹⁴ Sensible a cambios en mayúsculas y minúsculas.

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.

Ejemplos de nombres de variables

Veamos a continuación una tabla con nombres de variables:

Tabla 6: 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

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 –.¹⁵ 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](#)¹⁶ 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

¹⁵ Más información sobre convenciones de nombres en [PEP 8](#).

¹⁶ Del término inglés «Python Enhancement Proposals».

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 largas); nos quedaríamos con las otras dos. Si nos 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.

1.3 Asignación

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



Figura 9: 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 5 + 2.0 a la variable result y muestra su tipo.

Ampliar conocimientos

- [Basic Data Types in Python](#)
- [Variables in Python](#)
- [Immutability in Python](#)

2. Números



Foto original de portada por [Brett Jordan](#) en Unsplash.

En esta sección veremos los tipos de datos numéricos que ofrece Python centrándonos en [booleanos](#), [enteros](#) y [flotantes](#).

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.

Advertencia: 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
```

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:

```

>>> 5
5
>>> 0
0
>>> 05
  File "<stdin>", line 1
    05
    ^
SyntaxError: invalid token
>>> 123
123
>>> +123
123
>>> -123
-123
>>> 1000000
1000000
>>> 1_000_000
1000000

```

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 7: Operaciones con enteros en Python

Operador	Descripción	Ejemplo	Resultado
+	Suma	5 + 8	13
-	Resta	90 - 10	80
*	Multiplicación	4 * 7	28
/	División flotante	7 / 2	3.5
//	División entera	7 // 2	3
%	Módulo	7 % 3	1
**	Exponenciación	3 ** 4	81

Veamos algunas pruebas de estos operadores:

```
>>> 5 + 9 + 4
18
>>> 4 ** 4
256
>>> 7 / 3
2.3333333333333335
>>> 7 // 3
2
>>> 5 / 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*.

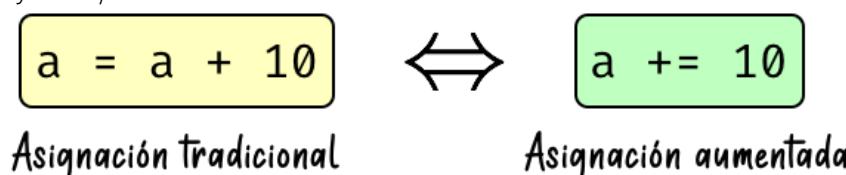


Figura 10: Asignación aumentada en Python

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:

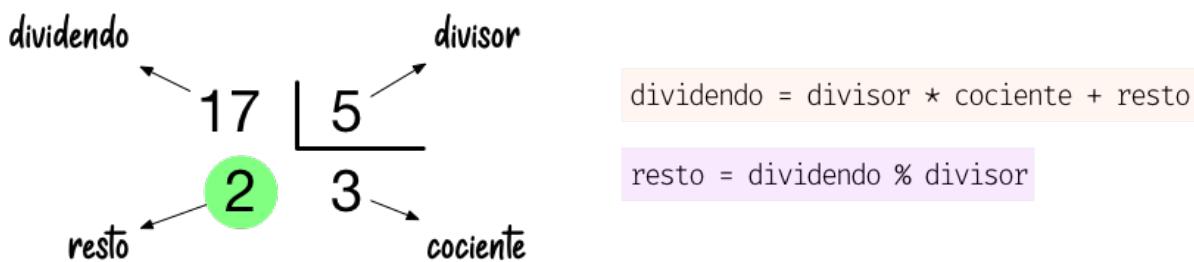


Figura 11: Operador «módulo» en Python

```
>>> 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*¹⁷. Por ejemplo:

$$4^{\frac{1}{2}} = 4^{0,5} = \sqrt{4} = 2$$

Hecho en Python:

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

2.3 Flotantes

Los números en [punto flotante](#)¹⁸ tienen [parte decimal](#). Veamos algunos ejemplos de flotantes en Python.

Lista 3: Distintas formas de escribir el flotante `5.0`

```
>>> 5.0
5.0
>>> 5.
5.0
>>> 05.0
5.0
>>> 05.
5.0
>>> 5e0
5.0
```

2.4 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

¹⁷ No siempre es una raíz cuadrada porque se invierten numerador y denominador

¹⁸ Punto o coma flotante es una [notación científica](#) usada por computadores

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

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(25.5)
25
>>> float(False)
0.0
>>> float(10)
10.0
```

Para poder **comprobar el tipo** que tiene 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'>
```

Ejercicio

Existe una aproximación al seno de un ángulo θ expresado en *grados*.

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

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:

- $\sin(90) = 1,0$
- $\sin(45) = 0,7071067811865475$
- $\sin(50) = 0,766044443118978$

Ampliar conocimientos

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

3. Cadenas de texto



Foto original de portada por [Roman Kraft](#) en Unsplash

Los «strings» o cadenas de texto son el primer ejemplo de *secuencia* en Python. En concreto se trata de una **secuencia de caracteres**. Un carácter es la mínima unidad en un sistema de escritura e incluye letras, dígitos, símbolos, signos de puntuación e incluso espacios en blanco o directivas.

3.1 Creando «strings»

Para escribir una cadena de texto en Python basta con rodear los caracteres con comillas simples:

```
>>> '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:

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

Cadena vacía

La cadena vacía es aquella que no tiene caracteres, pero es perfectamente válida. Aunque a priori no lo pueda parecer, es un recurso importante en cualquier código. Su representación en Python es la siguiente:

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

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'
```

También podemos convertir «strings» a otros tipos de datos:

```
>>> int('7')
7

>>> float('3.77')
3.77

>>> bool('1')
True

>>> bool('')
False
```

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 linea
>>> msg = 'Primera línea\nSegunda línea\nTercera línea'
>>> print(msg)
Primera línea
Segunda línea
Tercera línea

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

# 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: Es cuando utilizamos la función print() que vemos el resultado de utilizar los caracteres escapados.

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.

3.5 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'
>>> age = input('Introduzca su edad: ')
Introduzca su edad: 41
>>> age
'41'
>>> type(age)
<class '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](#).

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 suena agua lleva'  
  
>>> 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 posición propia. Para obtener un único carácter dentro de una cadena de texto es necesario especificar su **índice** dentro de corchetes [...].

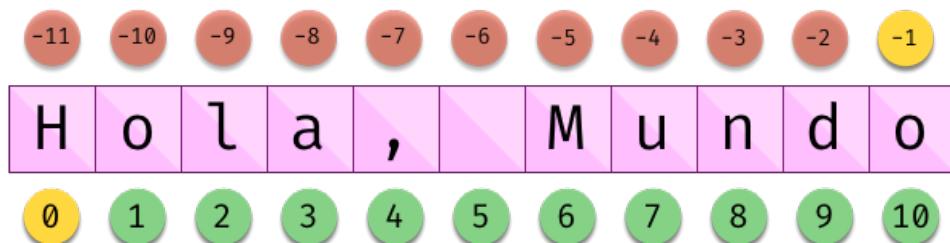


Figura 12: Indexado de una cadena de texto

Veamos algunos ejemplos de acceso a caracteres:

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

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

Nota: 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 para determinados casos.

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

```
>>> sentence[100]
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 que se crea un nuevo objeto con las modificaciones.

Trocear una cadena

Es posible extraer «trozos» («rebanadas») de una cadena de texto¹⁹. Tenemos varias aproximaciones a ello:

¹⁹ El término usado en inglés es *slice*.

[:] 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*.

[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()` que es una de las funciones «built-in»²⁰ que ofrece Python:

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

²⁰ Término inglés para referirse a algo que ya está incorporado por defecto con el lenguaje de programación.

Dividir una cadena

A contrario que len() algunas funciones son específicas de «strings». Para usar una función de cadena es necesario escribir el nombre del «string», un punto y el nombre de la función, pasando cualquier *argumento* necesario:

```
>>> 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.

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*²¹ 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 final del «string». También existen variantes de esta función para aplicarla únicamente al comienzo o 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.

²¹ Se suele utilizar el término inglés «padding» para referirse a estos caracteres.

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

```
>>> serial_number.lstrip() # left strip
'48374983274832    \n\n\t\t \t\t \n'
>>> serial_number.rstrip() # right strip
'\n\t\t \n 48374983274832'
```

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

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

Veamos aquellas funciones que proporciona Python para la búsqueda en cadenas de texto. Vamos a partir de una variable que contiene un trozo de la canción [Mediterráneo](#) de *Joan Manuel Serrat*.

```
>>> 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')
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
```

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 = 'quién a buen árbol se arrima Buena Sombra le cobija'

>>> proverb
'quién a buen árbol se arrima Buena Sombra le cobija'

>>> proverb.capitalize()
'Quién a buen árbol se arrima buena sombra le cobija'

>>> proverb.title()
'Quién A Buen Árbol Se Arrima Buena Sombra Le Cobija'

>>> proverb.upper()
'QUIÉN A BUEN ÁRBOL SE ARRIMA BUENA SOMBRA LE COBIJA'

>>> proverb.lower()
'quién a buen árbol se arrima buena sombra le cobija'

>>> proverb.swapcase()
'QUIÉN A BUEN ÁRBOL SE ARRIMA bUENA sOMBRA LE COBIJA'
```

3.7 «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 a la hora de construir un «string».

Para indicar en Python que una cadena es un «f-string» basta con precederla con 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.

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-strings²²](#)

Ejercicio

Obtenga el número de palabras que contiene la siguiente cadena de texto:

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

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](#)

²² Escrito por Nirant Kasliwal en Medium

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*.

1. Condicionales

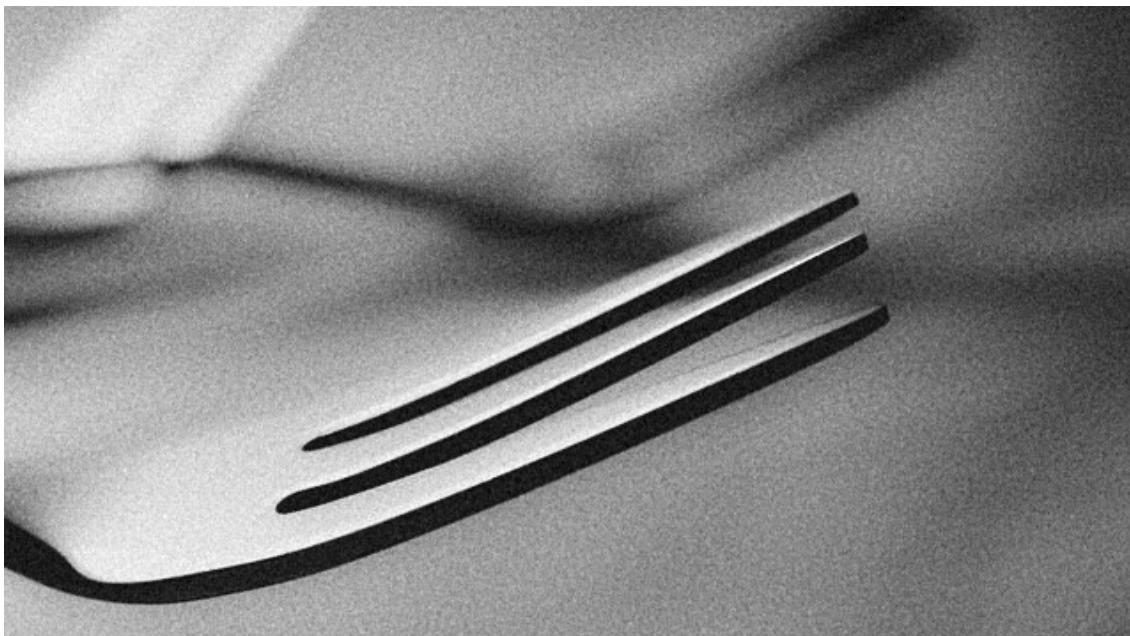


Foto original de portada por [ali nafezarefi](#) en Unsplash

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*.

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**.²³ 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.

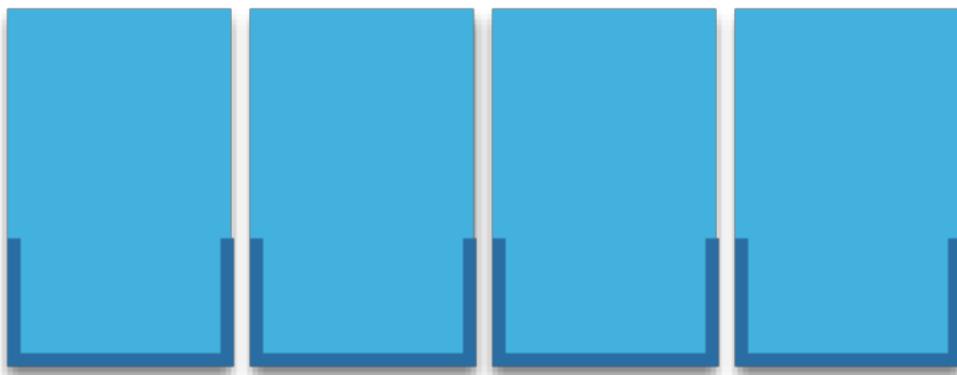


Figura 13: Python recomienda 4 espacios en blanco para indentar

1.2 Comentarios

Un *comentario* es un trozo de texto en tu programa que es ignorado por el intérprete de Python. Se pueden usar para aclarar líneas de código adyacentes, para dejar notas recordatorias o cualquier otro propósito.

Los comentarios se inician con el símbolo almohadilla # y desde ese punto hasta el final de la línea es parte del comentario:

Lista 1: Comentario de bloque

```
# 60 sec/min * 60 min/hr * 24 hr/day
seconds_per_day = 86400
```

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    # Liberar productos adicionales
```

²³ Reglas de indentación definidas en [PEP 8](#)

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:

usar la *barra invertida*\ o usar los *paréntesis* (...). Veamos un ejemplo:

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

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 prioridad.

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:
...     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**²⁴. 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 nos aparecen juntos un else y un if. Podemos sustituirlos por la sentencia elif:

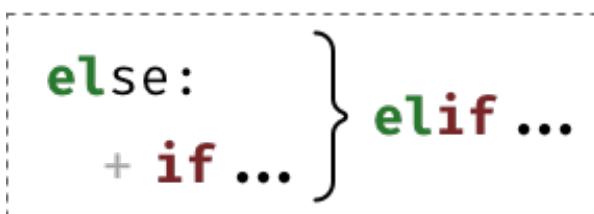


Figura 14: Construcción de la sentencia elif

Apliquemos esta mejora al código del ejemplo anterior:

²⁴ El anidamiento (o «nesting») hace referencia a incorporar sentencias unas dentro de otras mediante la inclusión de diversos niveles de profundidad (indentación).

```
>>> temperature = 28

>>> if temperature < 20:
...     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>

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><</code>
Menor o igual que	<code><=</code>
Mayor que	<code>></code>
Mayor o igual que	<code>>=</code>

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

```
>>> value = 7

>>> value == 5
False

>>> value == 7
True

>>> 5 < value
True

>>> value < 10
True
```

Podemos escribir condiciones más complejas usando los [operadores lógicos](#):

- o and
- o or
- o not

```
>>> (5 < value) or (value > 10)
True

>>> (5 < value) and (not (value > 10))
True

>>> (5 > value) and (value < 10)
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 5 y 10 haríamos:

```
>>> 5 < value < 10
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.

«Booleanos» en condiciones

Cuando queremos preguntar por la *veracidad* de 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:
...     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

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

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. Puede hacer la comprobación en [esta lista de años bisiestos](#).

Ampliar conocimientos

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

2. Bucles



Foto original de portada por [Gary Lopater](#) en Unsplash

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.

2.1 La sentencia while

El mecanismo más sencillo en Python para repetir instrucciones es mediante la sentencia while. El mensaje que podemos interpretar tras esta sentencia es: «Mientras se cumpla la condición haz algo». Veamos un sencillo bucle que muestra por pantalla los números del 1 al 5:

```
>>> count = 1

>>> while count <= 5:
...     print(count)
...     count += 1
...
1
2
3
4
5
```

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

<https://cutt.ly/dfeqTCZ>

La condición del bucle se comprueba en cada nueva repetición. En este caso chequeamos que la variable count sea menor o igual que 5. 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 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.

Bucle infinito

Si no establecemos bien 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
...
^C-----
KeyboardInterrupt                               Traceback (most recent call last)
...last)
<ipython-input-59-f6cb5d82e006> in <module>
      1 while num != 10:
----> 2     num += 2
      3

KeyboardInterrupt:

```

El problema que surge es que la variable num toma los valores 1, 3, 5, 7, 9, 11, ... por lo que nunca se cumple la condición 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.

2.2 La sentencia for

Python permite recorrer aquellos tipos de datos que sean **iterables**, es decir, que admitan *iterar*²⁵ 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.

²⁵ Realizar cierta acción varias veces. En este caso la acción es tomar cada elemento.

A continuación, 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 de 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>

Nota: 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'  
  
>>> 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>

Generar secuencias de números

La función `range()` devuelve un *flujo de números* en el rango especificado, sin necesidad de crear y almacenar previamente una larga estructura de datos. Esto permite generar rangos enormes sin consumir toda la *memoria* del sistema.

El uso de `range()` es similar a los [slices](#): `range(start, stop, step)`. Podemos omitir `start` y el rango empezaría en 0. El único valor requerido es `stop` y el último valor generado será justo el anterior a este. El valor por defecto de `step` es 1, pero se puede ir «hacia detrás» con -1.

`range()` devuelve un *objeto iterable*, así que necesitamos obtener los valores paso a paso con una sentencia `for ... in`²⁶. 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
```

²⁶ O convertir el objeto a una secuencia como una lista

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 viene a denominar **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.

2.3 Bucles anidados

Como ya vimos en las sentencias condicionales, el *anidamiento* es una técnica en 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
```

(continúa en la próxima página)

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
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
9 x 8 = 72
9 x 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/vfyewvj>

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 a 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, ...

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.

1. Listas



Foto original de portada por [Mike Arney](#) en Unsplash.

Las listas son adecuadas para [guardar elementos](#) en un [orden concreto](#). A diferencia, por ejemplo, de las cadenas de texto, las listas son [mutables](#). Podemos modificar, añadir o borrar elementos de una lista. Cabe destacar que un mismo valor puede aparecer más de una vez en una lista.

1.1 Creando listas

Una lista está compuesta por *cero o más elementos*, separados por *comas* y rodeados por *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 de datos muy versátil.

Ejecución [paso a paso](#) a través de *Python Tutor*.

<https://cutt.ly/Ofiiare>

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 acceder a sus elementos.

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()  
[]
```

1.3 Operaciones con listas

Obtener un elemento

Igual que en el caso de las [cadenas de texto](#) podemos extraer un elemento de una lista especificando su **índice**. 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']

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

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

Hacer notar que el uso de *índices inválidos* en el troceado no genera una excepción. Python trata de ajustarse al índice válido más próximo:

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

>>> shopping[10:]
[]

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

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

Nota: 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']

>>> 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

La forma tradicional de añadir elementos al final de una lista es utilizar 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']
```

Añadir en cualquier posición de una lista

La función *append()* sólo permite añadir elementos al final de la lista. Cuando se quiere insertar un elemento en otra posición de una lista debemos usar *insert()* especificando el índice de colocación. También se trata de una función *destructiva*²⁷:

²⁷ Cuando hablamos de que una función/método es «destructiva/o» significa que modifica la lista (objeto) original, no que la destruye.

```
>>> 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 interpretar como la posición *delante* (a la izquierda) de la cual vamos a colocar el nuevo valor en la lista.

No hay que preocuparse por insertar un elemento en desplazamientos no válidos. Si el índice supera el tamaño de la lista, el elemento se insertará al final de la lista. Si el índice es demasiado bajo se insertará al comienzo de la lista. En ninguno de los dos casos vamos a obtener un error debido a esta circunstancia:

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

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

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

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

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

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', 'i', '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 ampliar 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
```

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»²⁸ 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'
>>> shopping
['Agua', 'Huevos', 'Sal']
```

Nota: Si usamos la función 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.

Borrado completo de la lista:

1. Utilizando la función clear():

²⁸ Más adelante veremos el comportamiento de las funciones. Devolver o retornar un valor es el resultado de aplicar una función

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

>>> shopping.clear()

>>> shopping
[]
```

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

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

>>> shopping = []

>>> shopping
[]
```

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

Encontrar un elemento

Para conocer el **índice** que tiene un determinado elemento dentro de una lista podemos hacer uso de la función index():

```
>>> 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
```

Pertenencia de un elemento

La forma **pitónica** de comprobar la existencia de un elemento (valor) dentro de una lista, 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.

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 15: Estructura de llamada a la función `join()`

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

>>> ','.join(shopping)
'Agua,Huevos,Aceite,Sal,Limón'

>>> ' '.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*.

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()

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

Ambos métodos admiten un *parámetro «booleano» reverse* para indicar si queremos que la ordenación se haga en [orden 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 a través de 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/TfuiZ0>

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/lfioiG>

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')]
```

1.4 Cuidado con las copias

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>

1.5 Construyendo una lista

Una forma muy habitual de trabajar con listas es empezar con una vacía e ir añadiendo elementos poco a poco. 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)
...
>>> 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>

1.6 Funciones matemáticas

Python nos ofrece, entre otras²⁹, 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
```

Máximo de todos los valores: Mediante la función max():

```
>>> data = [5, 3, 2, 8, 9, 1]
>>> max(data)
9
```

1.7 Listas de listas

Como ya hemos visto en varias ocasiones, las listas son estructuras de datos que pueden contener elementos heterogéneos. Una de la forma en las que podemos utilizarlas es usando listas como elementos.

Por ejemplo, si pensamos en la variable shopping que hemos estado usando y que representa la *lista de la compra*, la podríamos dividir en sublistas, cada una de ellas indicando los productos que vamos a comprar en las secciones del supermercado:

```
>>> fruit_shop = ['Naranjas', 'Manzanas', 'Melón']
>>> butcher_shop = ['Pollo', 'Hamburguesas', 'Lomo']
>>> delicatessen = ['Jamón', 'Queso', 'Salami', 'Mortadela']
>>> hygiene = ['Jabón', 'Desodorante', 'Crema']
>>> bakery = ['Pan', 'Croissant', 'Magdalenas']
```

Ahora podríamos juntar todo en una única lista de la compra:

²⁹ Existen multitud de paquetes científicos en Python para trabajar con listas o vectores numéricos. Una de las más famosas es la librería [Numpy](#).

```
>>> shopping = []  
  
>>> shopping.append(fruit_shop)  
>>> shopping.append(butcher_shop)  
>>> shopping.append(delicatessen)  
>>> shopping.append(hygiene)  
>>> shopping.append(bakery)  
  
>>> shopping  
[['Naranjas', 'Manzanas', 'Melón'],  
 ['Pollo', 'Hamburguesas', 'Lomo'],  
 ['Jamón', 'Queso', 'Salami', 'Mortadela'],  
 ['Jabón', 'Desodorante', 'Crema'],  
 ['Pan', 'Croissant', 'Magdalenas']]
```

Ejecución [paso a paso](#) a través de *Python Tutor*.

<https://cutt.ly/dfi7e41>

Ejercicio

Lea [desde teclado](#) una cadena de texto con números separados por comas. Sin utilizar las [funciones matemáticas](#), obtenga la media de dichos valores (*muestre el resultado con 2 decimales*).

Ejemplo

- Entrada: '32,56,21,99,12,17'
- Salida: 37.50

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](#)

2. Tuplas



Foto original de portada por [engin akyurt](#) en Unsplash.

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**.

2.1 Creando tuplas

Podemos pensar en crear tuplas tal y como *lo hacíamos con listas*, pero usando **paréntesis** en lugar de *corthetes*:

```
>>> empty_tuple = ()  
>>> tenerife_geoloc = (28.46824, -16.25462)  
>>> three_wise_men = ('Melchor', 'Gaspar', 'Baltasar')
```

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
```

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
```

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()  
()
```

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()

2.5 Desempaquetado de tuplas

El **desempaquetado** es una característica de las tuplas que nos permite *asignar una tupla a variables independientes*.

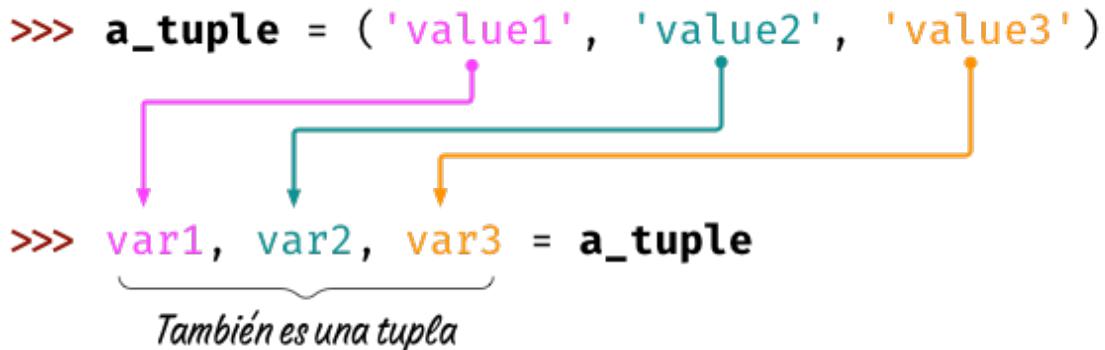


Figura 16: Desempaquetado de tuplas

Veamos un ejemplo con código:

```
>>> three_wise_men = ('Melchor', 'Gaspar', 'Baltasar')

>>> king1, king2, king3 = three_wise_men

>>> king1
'Melchor'
>>> king2
'Gaspar'
>>> king3
'Baltasar'
```

Intercambio de valores

A través del desempaquetado de variables 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.

3. Diccionarios

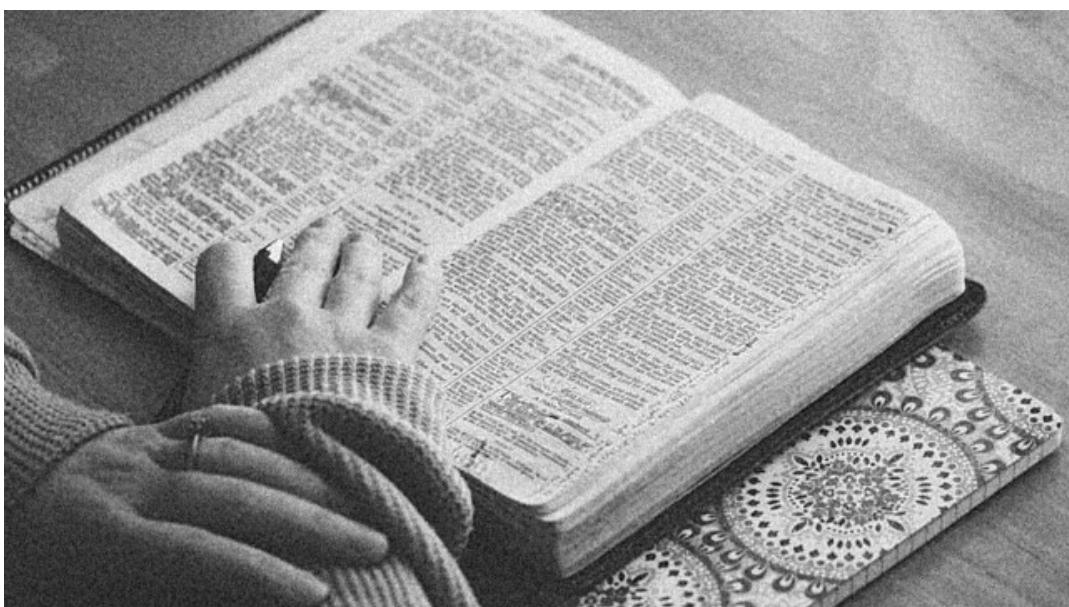


Foto original de portada por [Aaron Burden](#) en Unsplash

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).

Los diccionarios en Python tienen las siguientes *características*:

- Mantienen el **orden** en el que se insertan las claves.³⁰
- 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.³¹

³⁰ 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](#)

³¹ Véase este [análisis de complejidad y rendimiento](#) de distintas estructuras de datos en CPython.

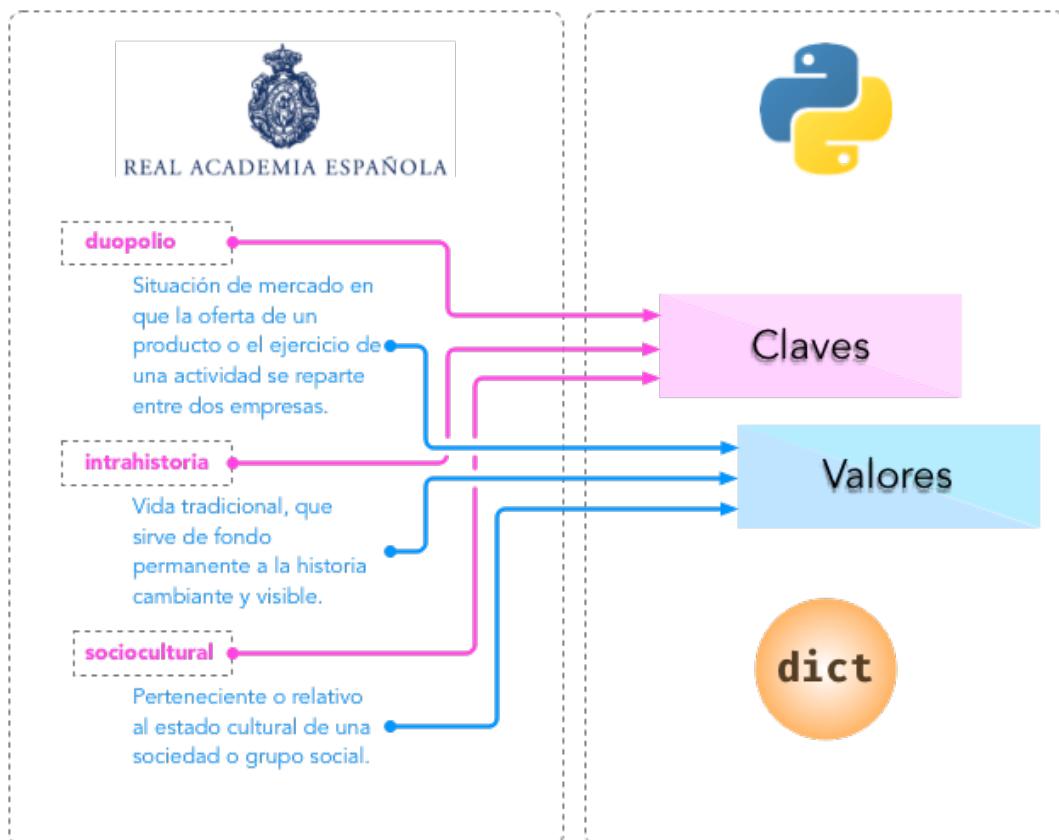


Figura 17: Analogía de un diccionario en Python

Nota: En otros lenguajes de programación, a los diccionarios se les conoce como *arrays asociativos*, «hashes» o «hashmaps».

3.1 Creando diccionarios

Para crear un diccionario basta con usar llaves {} rodeando pares clave: valor separados 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 enteros.

Ejecución [paso a paso](#) a través de *Python Tutor*.

<https://cutt.ly/Sfav2Yw>

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(['ab', 'cd'])
{'a': 'b', 'c': 'd'}

>>> # Diccionario a partir de una tupla de cadenas de texto
>>> dict(('ab', 'cd'))
{'a': 'b', 'c': 'd'}

>>> # Diccionario a partir de una lista de listas (de cadenas de texto)
>>> dict([['a', 'b'], ['c', 'd']])
{'a': 'b', 'c': 'd'}
```

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:

```
>>> 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:

```
>>> 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 None³² 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.³³

Línea 11: La clave buscada no existe y nos devuelve el valor que hemos aportado por defecto.

Añadir o modificar un elemento

Añadir un elemento a un diccionario es sencillo. 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.

³² None es la palabra reservada en Python para la «nada». Más información en [esta web](#).

³³ Realmente no estamos viendo nada en la consola de Python porque la representación en cadena de texto es vacía.

Al contrario que en [las listas](#), no hay que preocuparse por las excepciones (errores) por fuera de rango durante la asignación de elementos a un diccionario:

```
>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     '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'}
```

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
```

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

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.

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.³⁴

³⁴ En este caso «último» hace referencia al diccionario que se encuentra más a la derecha en la expresión.

Python ofrece dos mecanismos para realizar esta combinación. Vamos a partir de los siguientes diccionarios para exemplificar su uso:

```
>>> rae1 = {
...     'bifronte': 'De dos frentes o dos caras',
...     'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'
...
... }

>>> 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'}
```

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',
...     '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.

3.4 Cuidado con las copias

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 la propiedad de *mutabilidad*. 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'

>>> 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'}

```

3.5 Construyendo un diccionario

Una forma muy habitual de trabajar con diccionarios es empezar con 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 utilizado la función `enumerate()` que ya vimos para las listas en el apartado: [Iterar usando enumeración](#).

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}

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](#)

4. Conjuntos



Foto original de portada por [Duy Pham](#) en Unsplash.

Un [conjunto](#) en Python se representa por una serie de [valores únicos](#) y [sin orden establecido](#). Lo podríamos ver como un diccionario al que le hemos quitado los valores y nos hemos quedado sólo con las claves. Mantiene muchas similitudes con el [concepto matemático de conjunto](#)

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
```

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
```

4.2 Conversión

Para convertir otros tipos de datos en un conjunto podemos usar la función set():

```
>>> set('aplatanada')  
{'a', 'd', 'l', 'n', 'p', 't'}  
  
>>> set([1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 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.

4.3 Operaciones con conjuntos

Obtener un elemento

En un conjunto no existe un orden establecido para sus elementos, por lo cual **no podemos acceder a un elemento en concreto**.

De este hecho se deriva igualmente que *no podemos modificar un elemento existente*, ya que no podemos acceder a él. 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 dónde 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:

```
>>> 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>

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
```

4.4 Teoría de conjuntos

Vamos a partir de dos conjuntos $A = \{1, 2\}$ y $B = \{2, 3\}$ para ejemplificar 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}
```

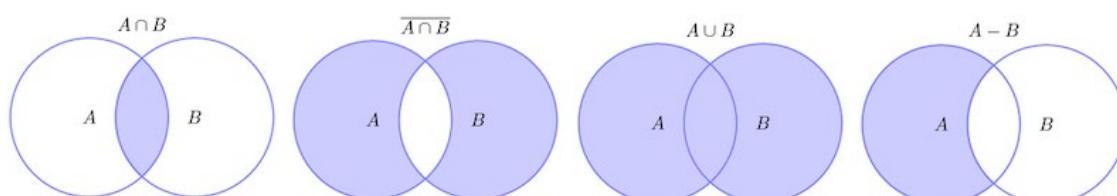


Figura 18: Diagramas de Venn

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}
```

Ampliar conocimientos

- [Sets in Python](#)

5. Ficheros



Foto original de portada por [Maksym Kaharlytskyi](#) en Unsplash

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.

Un **fichero** es una *secuencia de bytes* almacenados en algún *sistema de ficheros* y accesibles por un *nombre de fichero*. Un *directorio* (o carpeta) es una colección de ficheros, y probablemente de otros directorios. Muchos sistemas de ficheros son jerárquicos y a menudo nos referimos a ellos como árbol de ficheros.

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 es posible 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 una vez podríamos ocupar demasiada RAM y reducir el rendimiento de la 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

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.

5.2 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 de que las carpetas hasta llegar a dicho fichero existen. En otro caso obtenemos un error de tipo FileNotFoundError.

Ahora ya podemos hacer uso de la función write() para que 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 Canarias³⁵.

```

1 >>> canary_iata = ("GCFV", "GCHI", "GCLA", "GCLP", "GCGM", "GCRR",
2     ...      "GCTS", "GCXO")
3
4 >>> for code in canary_iata:
5     ...     f.write(code + '\n')
6
7 >>> f.close()
```

Nótese:

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.

³⁵ Fuente: [Smart Drone](#)

Advertencia: Siempre que se abre un fichero en modo escritura utilizando el argumento 'w', el fichero se inicializa, borrando cualquier contenido que pudiera tener.

5.3 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() harán que aparezcan nuevo contenido al final del contenido ya existente en dicho fichero.

5.4 Usando contextos

Python ofrece [gestores de contexto](#) que permiten establecer reglas de entrada y salida al contexto definido. 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.

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.³⁶

En este capítulo veremos las facilidades que nos proporciona Python para trabajar en la línea de modularidad del código.

1. Funciones



Foto original por [Nathan Dumlao](#) en Unsplash.

Hasta ahora todo lo que hemos hecho han sido breves fragmentos de código Python. Esto puede ser razonable para pequeñas tareas, pero nadie quiere reescribir los fragmentos de código cada vez. Necesitamos una manera de organizar nuestro código en piezas manejables.

³⁶ Definición de modularidad en [Wikipedia](#)

El primer paso para la **reutilización de código** es la **función**. Se trata de un trozo de código con nombre y separado del resto. Puede tomar cualquier número y tipo de *parámetros* y devolver cualquier número y tipo de *resultados*.

Básicamente podemos hacer dos cosas con una función:

- Definirla (con cero o más parámetros).
- Invocarla (y obtener cero o más resultados).

1.1 Definir una función

Para definir una función en Python debemos usar la palabra reservada `def` seguida del nombre de la función con paréntesis rodeando a los parámetros de entrada y finalmente dos puntos :

Advertencia: Prestar especial atención a los dos puntos : porque suelen olvidarse en la *definición de la función*.

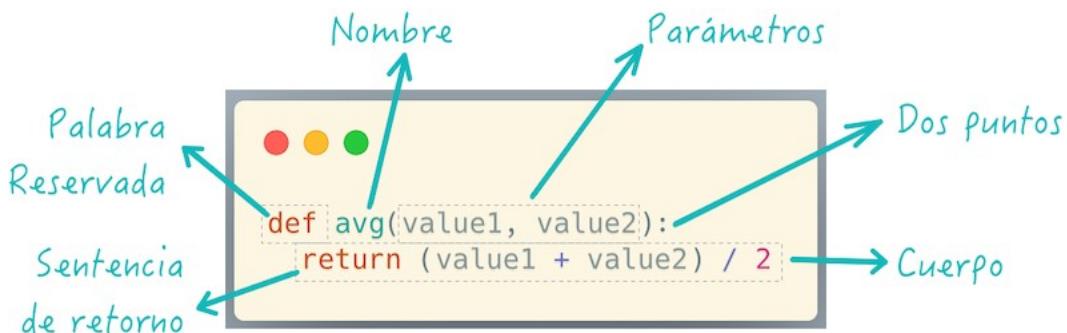


Figura 19: Definición de una función en Python

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 basta con escribir su nombre y utilizar 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 la 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 agree():
...     return True
...
>>> agree()
True
```

Pero no sólo podemos invocar a la función directamente, también la podemos integrar en otras expresiones. Por ejemplo, en condicionales:

```
>>> if agree():
...     print('Trato hecho')
... else:
...     print('Hasta la próxima')
...
Trato hecho
```

Nota: En la sentencia return podemos incluir variables y expresiones, no únicamente literales.

En aquellos casos en los que una función no tenga un return explícito, siempre devolverá None.

```
>>> def foo():
...     x = 'foo'
...
>>> print(foo())
None
```

1.2 Parámetros y argumentos

Vamos a empezar a crear funciones que reciben **parámetros**. En este caso escribiremos una función echo() que recibe el parámetro anything y muestra esa variable dos veces separada por un espacio:

```
>>> def echo(anything):
...     return anything + ' ' + anything
...
>>> echo('Is anybody out there?')
'Is anybody out there? Is anybody out there?'
```

Nota: En este caso, 'Is anybody out there?' 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:

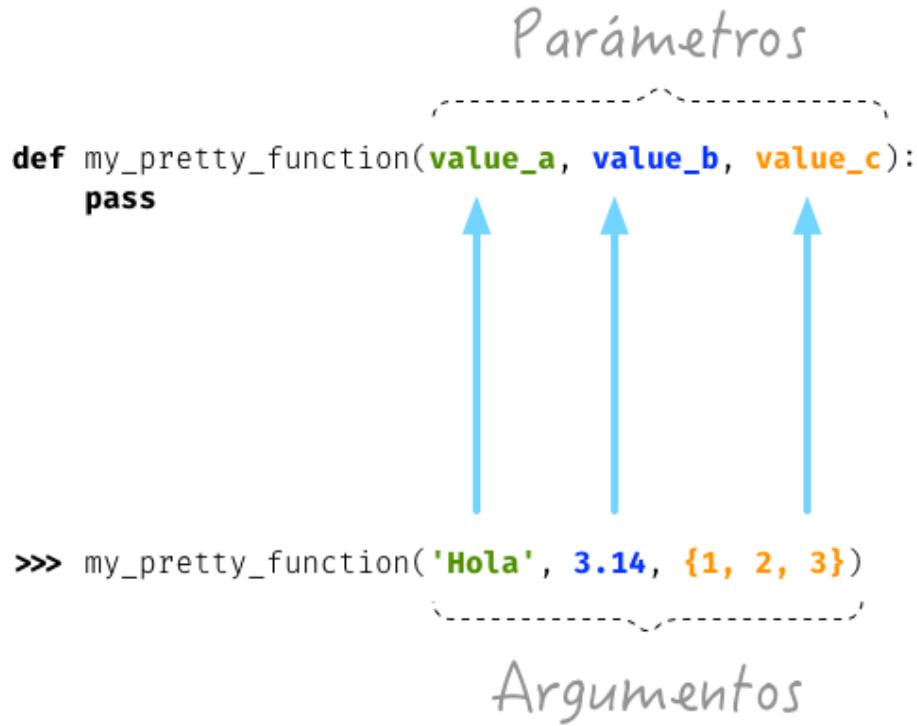


Figura 20: Parámetros y argumentos de una función

Truco: La sentencia `pass` permite «no hacer nada». Es una especie de «placeholder».

Veamos otra función con algo más de lógica de negocio:³⁷

³⁷ Término para identificar el «algoritmo» o secuencia de instrucciones derivadas del procesamiento que corresponda.

```
>>> def fruit_detection(color):
...     if color == 'red':
...         return "It's an apple"
...     elif color == 'yellow':
...         return "It's a banana"
...     elif color == 'green':
...         return "It's a kiwi"
...     else:
...         return f"I don't know about the color {color}"
...
>>> fruit = fruit_detection('green')
>>> fruit
"It's a kiwi"
```

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 y devuelve un diccionario a partir de los argumentos recibidos:

```
>>> def menu(wine, entree, dessert):
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}
... 
```

Una posible llamada a la función con argumentos posicionales sería la siguiente:

```
>>> menu('Flor de Chasna', 'Garbanzas', 'Quesillo')
{'wine': 'Flor de Chasna', 'entree': 'Garbanzas', 'dessert': 'Quesillo'} 
```

Lo que ha sucedido es un **mapeo** directo entre argumentos y parámetros en el mismo orden que estaban definidos:

Argumento	Parámetro
Flor de chasna	wine
Garbanzas	entree
Quesillo	dessert

Nota: Una clara desventaja del uso de argumentos posicionales es que se necesita recordar el significado de cada posición.

Argumentos por nombre

Para evitar la confusión que pueden producir los argumentos posicionales, es posible especificar argumentos **usando el nombre de los correspondientes parámetros**, incluso en un orden distinto a cómo están definidos en la función:

```
>>> menu(entree='Queso asado', dessert='Postre de café', wine='Arautava
... ')
{'wine': 'Arautava', 'entree': 'Queso asado', 'dessert': 'Postre de
... café'}
```

Incluso podemos *mezclar* argumentos posicionales y argumentos por nombre:

```
>>> menu('Marba', dessert='Frangollo', entree='Croquetas')
{'wine': 'Marba', 'entree': 'Croquetas', 'dessert': 'Frangollo'}
```

Nota: Si se llama a una función mezclando argumentos posicionales y por nombre, los argumentos posicionales deben ir primero.

```
>>> menu(dessert='Frangollo', entree='Croquetas', 'Marba')
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. El valor por defecto se usará cuando en la llamada a la función no se haya proporcionado el correspondiente argumento.

Supongamos que nos gusta mucho el *Tiramisú*. Podemos especificar en la definición de la función que, si no se especifica el postre, éste sea siempre *Tiramisú*.

```
>>> def menu(wine, entree, dessert='Tiramisú'):
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}
... 
```

Llamada a la función sin especificar postre:

```
>>> menu('Ignios', 'Ensalada')
{'wine': 'Ignios', 'entree': 'Ensalada', 'dessert': 'Tiramisú'}
```

Llamada a la función indicando un postre concreto:

```
>>> menu('Tajinaste', 'Revuelto de setas', 'Helado')
{'wine': 'Tajinaste', 'entree': 'Revuelto de setas', 'dessert': 'Helado
... '}
```

Importante: Los valores por defecto en los parámetros se calculan cuando se [define](#) la función, no cuando se [ejecuta](#).

1.3 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 echo(anything):
...     'echo returns its input argument'
...     return anything
... 
```

La forma más ortodoxa de escribir un docstring es utilizando *triples comillas*.

```
>>> def print_if_true(thing, check):
...     """
...     Prints the first argument if a second argument is true.
...     The operation is:
...         1. Check whether the *second* argument is true.
...         2. If it is, print the *first* argument.
...     """
...     if check:
...         print(thing) 
```

Para ver el docstring de una función, basta con utilizar `help`:

```
>>> help(print_if_true)

Help on function print_if_true in module __main__:

print_if_true(thing, check)
    Prints the first argument if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument. 
```

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)³⁸ pero vamos a centrarnos en el modelo NumPy/SciPy.

Este modelo se basa en:

- Una primera línea de **descripción de la función**.
- A continuación, especificamos las características de los **parámetros** (incluyendo sus tipos) usando el encabezado Parameters.
- Por último, si la función **retorna un valor**, lo indicamos con el encabezado Returns.

Veamos un ejemplo:

```
>>> def subtract(value1, value2, vabs=False):
...     '''Subtract two values with choice of absolute value
...
...     Parameters
...
...     -----
...     value1 : int
...         First value in subtraction
...     value2 : int
...         Second value in subtraction
...     vabs : bool
...         Indicates if absolute value is performed over the
...         subtraction
...
...     Returns
...     -----
...     int
...         Subtraction of input values
...
...     result = value1 - value2
...     if vabs:
...         result = abs(result)
...     return result
...
...
...
>>> subtract(3, 5)
-2
>>> subtract(3, 5, True)
2
```

³⁸ Véase [Docstring Formats](#)

1.4 Espacios de nombres

Un nombre puede hacer referencia a múltiples cosas, dependiendo de dónde lo estemos usando. Los programas en Python tienen diferentes [espacios de nombres](#), secciones donde un nombre particular es único e independiente del mismo nombre en otros espacios de nombres.

Cada función define su propio espacio de nombres. Si se define una variable x en el programa principal y otra variable x dentro de una función, hacen referencia a cosas diferentes. Dicho esto, también es posible (*aunque desaconsejado*) acceder al espacio de nombres global dentro de las funciones.

En el siguiente ejemplo se define una variable global (*primer nivel*) y luego mostramos su valor directamente y mediante una función:

```
>>> animal = 'tiger'

>>> def print_global():
...     print('inside print_global:', animal)
...

>>> print('at the top level:', animal)
at the top level: tiger

>>> print_global()
inside print_global: tiger
```

Ejecución [paso a paso](#) a través de *Python Tutor*.

<https://cutt.ly/3fMI8de>

Sin embargo, si creamos una variable dentro de la función que también tenga el nombre animal, realmente estaremos creando una nueva variable distinta de la global:

```
>>> animal = 'tiger'

>>> def change_local():
...     animal = 'panther'
...     print('inside change_local:', animal)
...

>>> print('at the top level:', animal)
at the top level: tiger

>>> change_local()
inside change_local: panther
```

Ejecución [paso a paso](#) a través de *Python Tutor*.

<https://cutt.ly/ifMOeYf>

Ejercicio

Primera parte

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

Ampliar conocimientos

- [Defining Your Own Python Function](#)
- [Python args and kwargs: Demystified](#)
- [Documenting Python Code: A Complete Guide](#)
- [Thinking Recursively in Python](#)
- [Writing Comments in Python](#)

2. Módulos



Foto original por [Xavi Cabrera](#) en Unsplash.

Es una certeza que, antes o después, usaremos código Python en más de un fichero. Un **módulo** es simplemente un fichero con código Python. No se necesita hacer nada especial. Cualquier código Python se puede usar como un módulo en código de terceros.

2.1 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 tu 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`

```

def addere(a, b):
    '''Sum of input values'''
    return a + b

def minuas(a, b):
    '''Subtract of input values'''
    return a - b

def pullulate(a, b):
    '''Product of input values'''
    return a * b

def partitus(a, b):
    '''Division of input values'''
    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.

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.

2.2 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 exemplificar 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

```
def et(a, b):
    '''Logic "and" of input values'''
    return a & b

def uel(a, b):
    '''Logic "or" of input values'''
    return a | b

def vel(a, b):
    '''Logic "xor" of input values'''
    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

2.3 Programa principal

Cuando decidimos hacer 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.

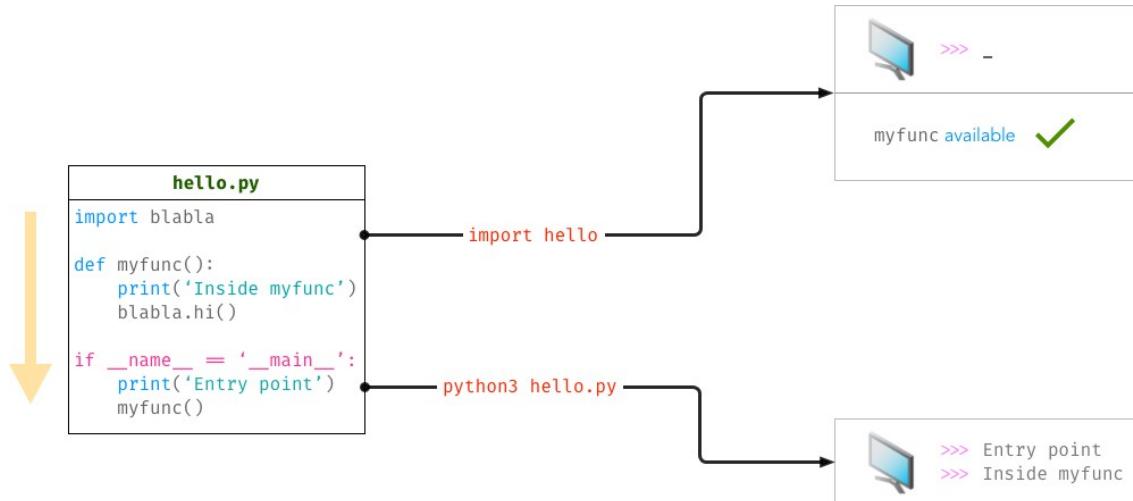


Figura 21: Comportamiento de un programa principal al importarlo o ejecutarlo

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](#)



Gobierno de Canarias

Consejería de Educación,
Universidades, Cultura y Deportes



UNIÓN EUROPEA

UCTICEE



FONDO EUROPEO DE DESARROLLO REGIONAL

Contenidos creados por:

Sergio Delgado Quintero

Maquetado por:

