

CURSO BÁSICO DE PYTHON

Guía

1. Bases de la programación

Para trabajar con cualquier lenguaje de programación, primero debemos entender cuál es el soporte físico en el que se programa, es decir, la computadora u ordenador. Una computadora es una máquina que ejecuta unos comandos y procesa datos de entrada, que serán enviados a los dispositivos de salida. Las computadoras actuales son máquinas electrónicas, digitales y programables, es decir que están formadas por circuitos electrónicos (circuitos eléctricos formados por componentes de control del flujo de electrones, como diodos y transistores y componentes pasivos, como conexiones, resistencias, etc.), que trabajan en base binaria (en lugar de nuestro sistema en base 10) y que puede realizar tareas diferentes en función de las instrucciones que le demos.

Como en toda la electrónica digital, la base de la programación se encuentra en el sistema binario, que representa el paso / no paso de la corriente eléctrica a través de los componentes electrónicos. En la computación se conoce como **bit** a la mínima unidad de información, que puede tener 2 valores, 0 o 1, y **byte** a un conjunto ordenado de 8 bits.

Se puede ver entonces que, en lugar de trabajar con números en base 10, como lo hacemos en nuestra vida diaria, y en los cuales existen los símbolos del 0 al 9, las computadoras utilizan diferentes iteraciones del sistema binario esto es en base 2, que, como ya se ha mencionado, sólo contiene los símbolos 0 y 1.

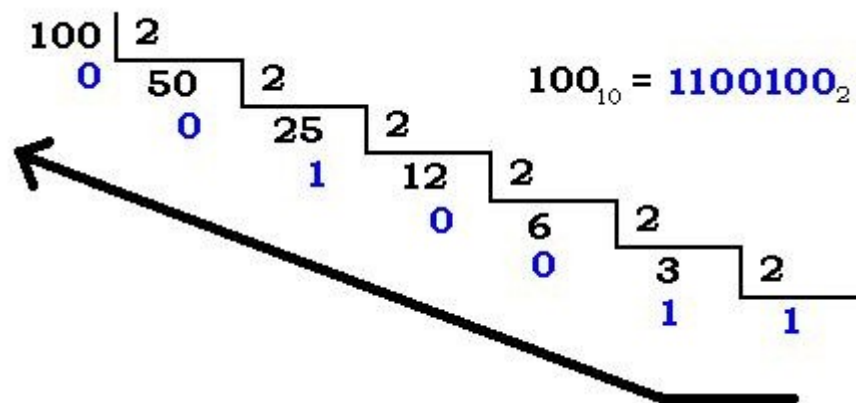
Hay varias formas de representar un número en base 10 en base 2:

- BCD (o binario natural) es la conversión más sencilla, consiste simplemente en una conversión de base. Así pues el número “treinta y siete”, que en base 10 sería el “37”, en binario sería “100101”.

Para pasar de binario a decimal solo hay que multiplicar cada bit por una potencia de 2 correspondiente a su posición (de derecha a izquierda) y empezando por 0:

$$100101 = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 32 + 4 + 1 = 37$$

El proceso contrario sería dividir el número por dos hasta que el resultado sea uno e ir cogiendo los restos de las operaciones anteriores:



- Signo y magnitud: se añade un bit a la izda. del número donde 0 representa que es positivo y 1 que es negativo. así pues, “37” sería “0100101” y “-37”, “1100101”
- Complemento a 2: es una forma más enrevesada a primera vista de obtener números enteros, pero que resulta mucho más fácil a la hora de realizar operaciones y por eso es más utilizada en sistemas informáticos.

Si quisiéramos hallar el complemento a 2 de 37 (esto es -37), la forma más sencilla sería invertir todos los dígitos del número (lo que se conoce como complemento a 1) y luego sumarle 1.

$00100101 \rightarrow 11011010 \rightarrow 11011011$

- Coma flotante: Para representar números reales normalmente se usa una notación científica, para el caso decimal:

$$0.37 = 37 * 10^{-2}$$

La coma flotante en binario se compone de 3 partes: un bit de signo para el número y así diferenciar entre 0.37 y -0.37, el exponente (normalmente un número en BCD), que representa la posición de la coma en binario, y la parte significativa o mantisa (puede ser binario natural o complemento a 2). Puesto que el primer bit de la mantisa siempre será 1 (excepto contadas excepciones) este bit no se suele guardar, aunque luego se trabaja con él en las operaciones (pasa de ser implícito a explícito).

Finalmente hay que comentar otro sistema de notación muy utilizado, el hexadecimal, o base 16, en el cual se representan con un solo símbolo los números del 1 al 15.

decimal	0	1	2	3	4	5	6	7
hex	0	1	2	3	4	5	6	7

BCD	0000	0001	0010	0011	0100	0101	0110	0111
decimal	8	9	10	11	12	13	14	15
hex	8	9	A	B	C	D	E	F
BCD	1000	1001	1010	1011	1100	1101	1110	1111

2. Introducción

Python fue creado a finales de los ochenta por [Guido van Rossum](#) en el Centro para las Matemáticas y la Informática (CWI, [Centrum Wiskunde & Informatica](#)), en los [Países Bajos](#), como un sucesor del [lenguaje de programación ABC](#), capaz de [manejar excepciones](#) e interactuar con el [sistema operativo Amoeba](#).⁵

El nombre del lenguaje proviene de la afición de su creador por los humoristas británicos [Monty Python](#).

Los usuarios de Python se refieren a menudo a la filosofía de Python que es bastante análoga a la filosofía de [Unix](#). El código que siga los principios de Python se dice que es "pythonico". Estos principios fueron descritos por el desarrollador de Python [Tim Peters](#) en [El Zen de Python](#)

- 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.
- Disperso es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son tan especiales como para quebrantar las reglas.
- Lo práctico gana a lo puro.
- Los errores nunca deberían dejarse pasar silenciosamente,
- a menos que hayan sido silenciados explícitamente.
- Frente a la ambigüedad, rechaza la opción de adivinar.
- Debería haber una (y preferiblemente solo una) forma de hacerlo,
- Aunque esa manera no sea obvia de hacerlo, a menos que sea usted holandés.
- "Ahora" es mejor que "nunca",
- Aunque "nunca" es mejor que "ahora mismo".
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, probablemente sea una buena idea
- Los espacios de nombre (namespace) son una gran idea. Hagamos más de esas cosas.

En resumen, Python es un lenguaje de alto nivel con gramática sencilla, clara y muy legible. Cuenta con tipado dinámico y fuerte, es orientado a objetos e interpretado. Fácil de aprender, versátil y open source.

3. Instalación del entorno

a. Windows

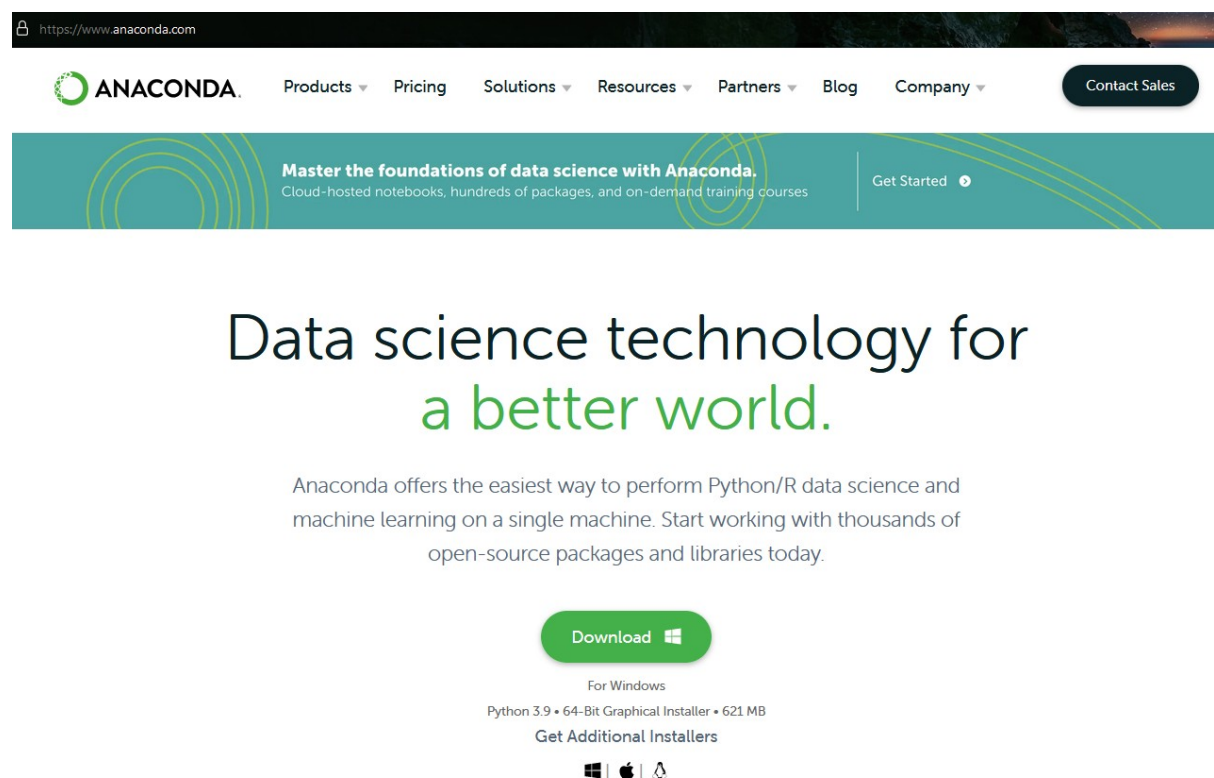
Instalación de Anaconda

Anaconda es una distribución gratuita de Python (así como del lenguaje R), que contiene una GUI (Graphical User Interface), llamada Anaconda Navigator, que nos permite acceder a diferentes aplicaciones de programación (como Spyder, JupyterLab, Visual Studio Code, etc.), así como acceder a la ventana de comandos (la CLI o Command-Line Interface) de anaconda, pudiendo crear diferentes entornos de python donde trabajar dependiendo de nuestras necesidades.

Instalación de Anaconda Navigator

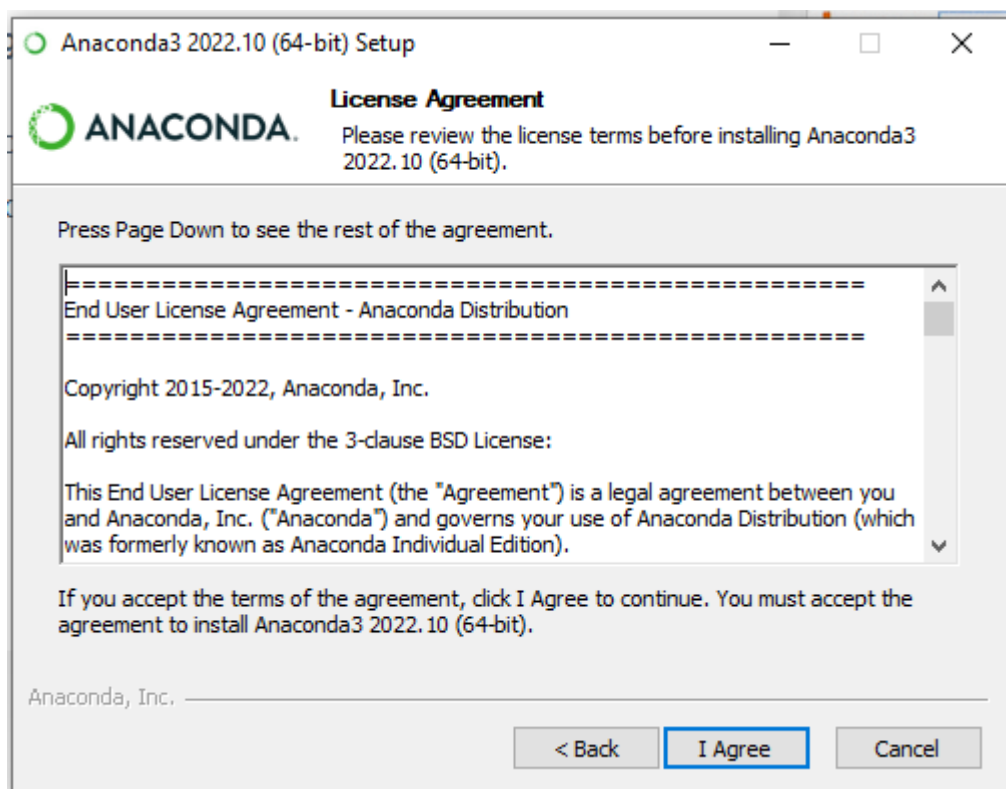
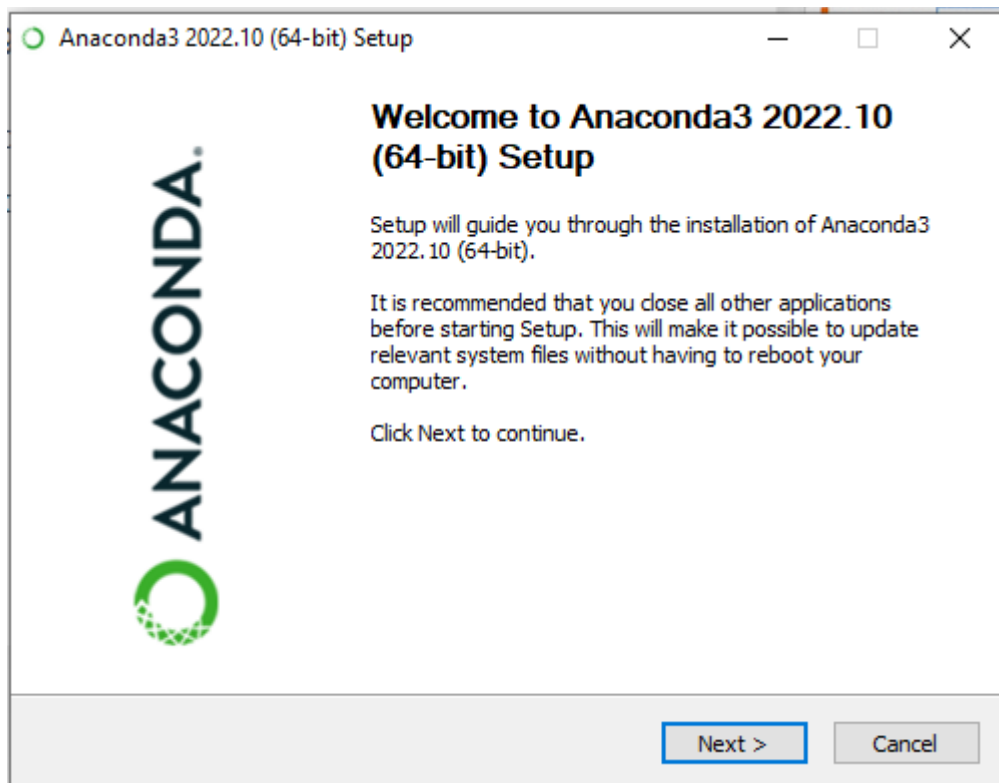
Para descargar el entorno, navegaremos aquí:

<https://www.anaconda.com/>

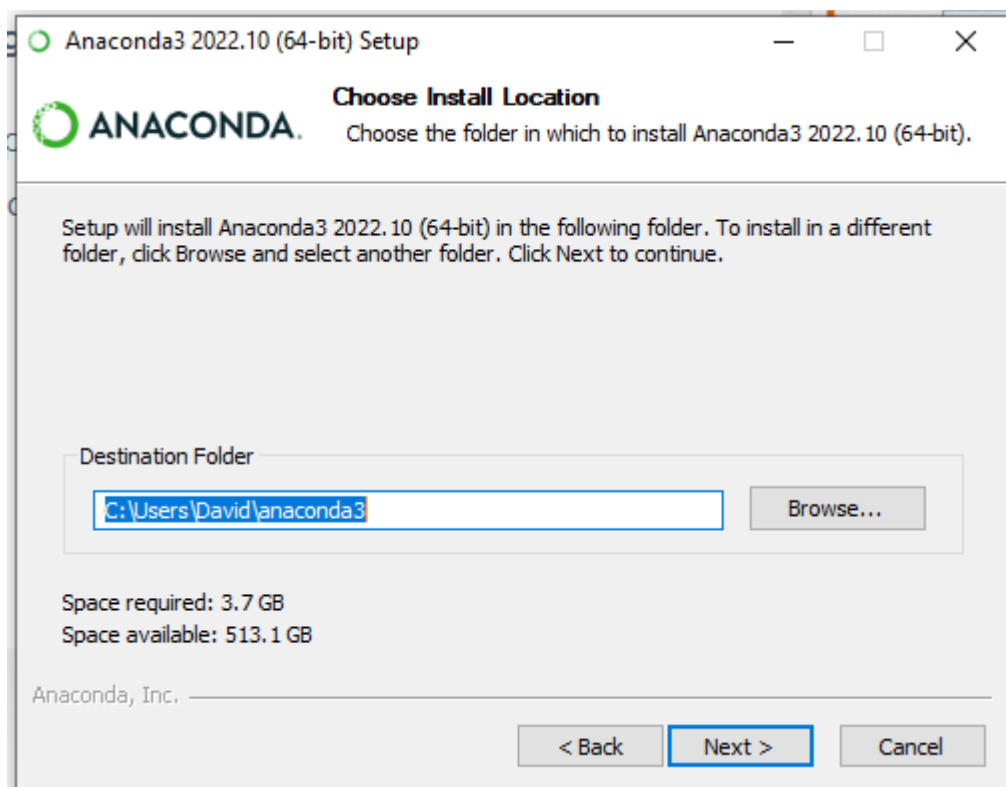
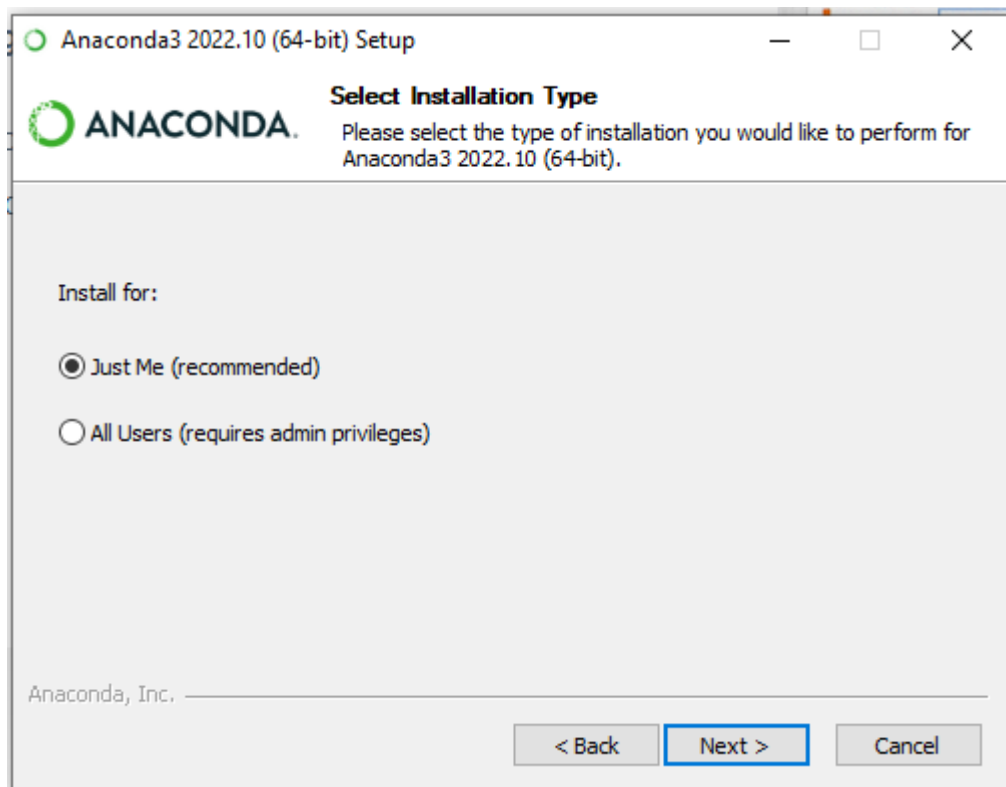


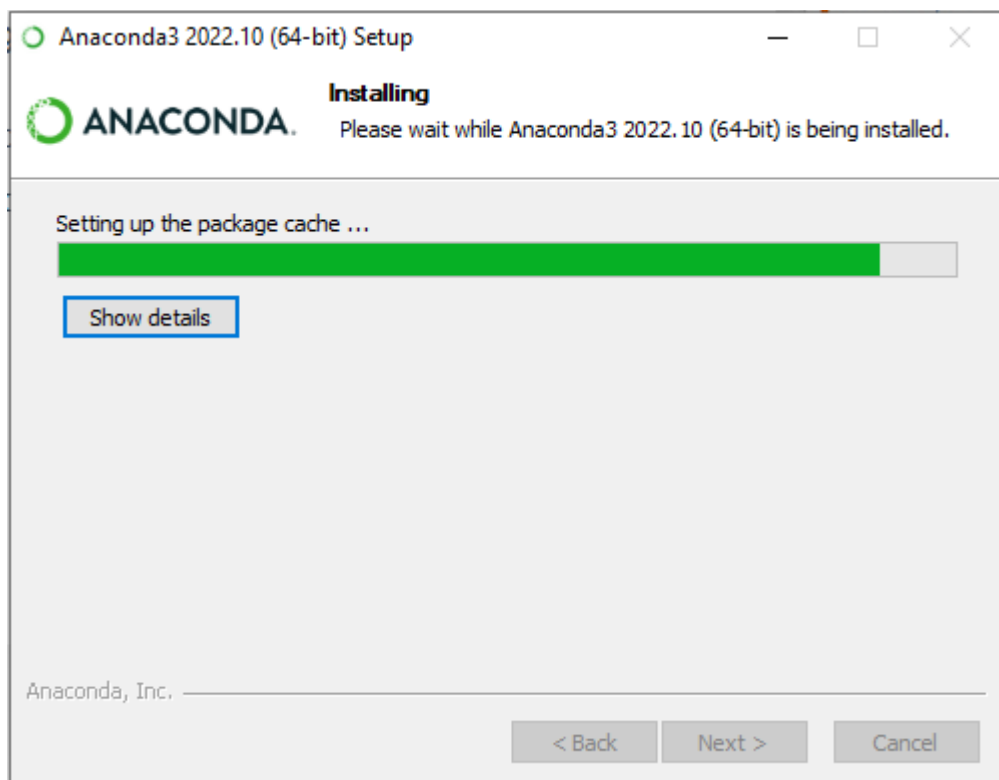
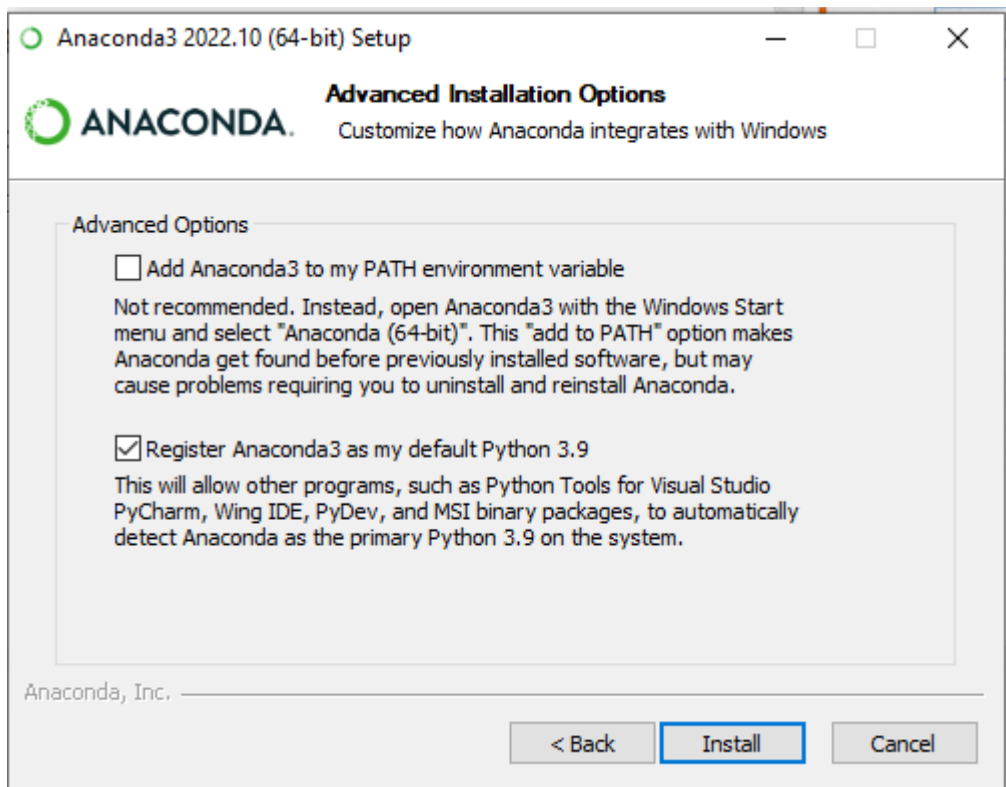
Haremos click en download y comenzaremos el proceso de instalación.

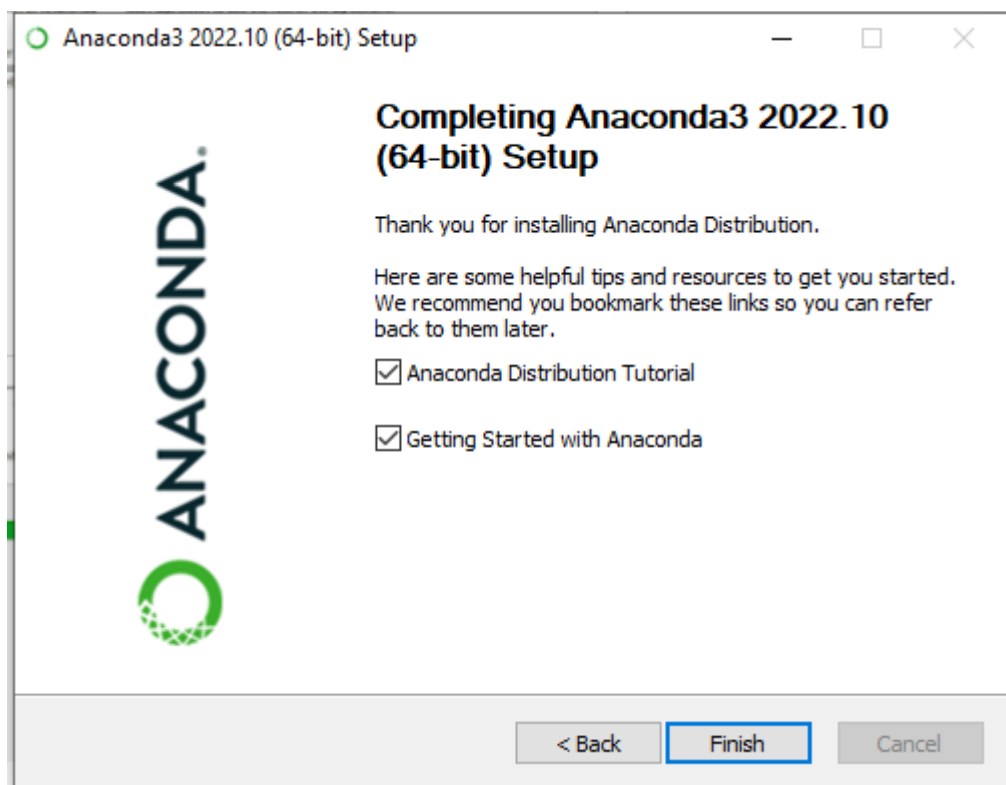
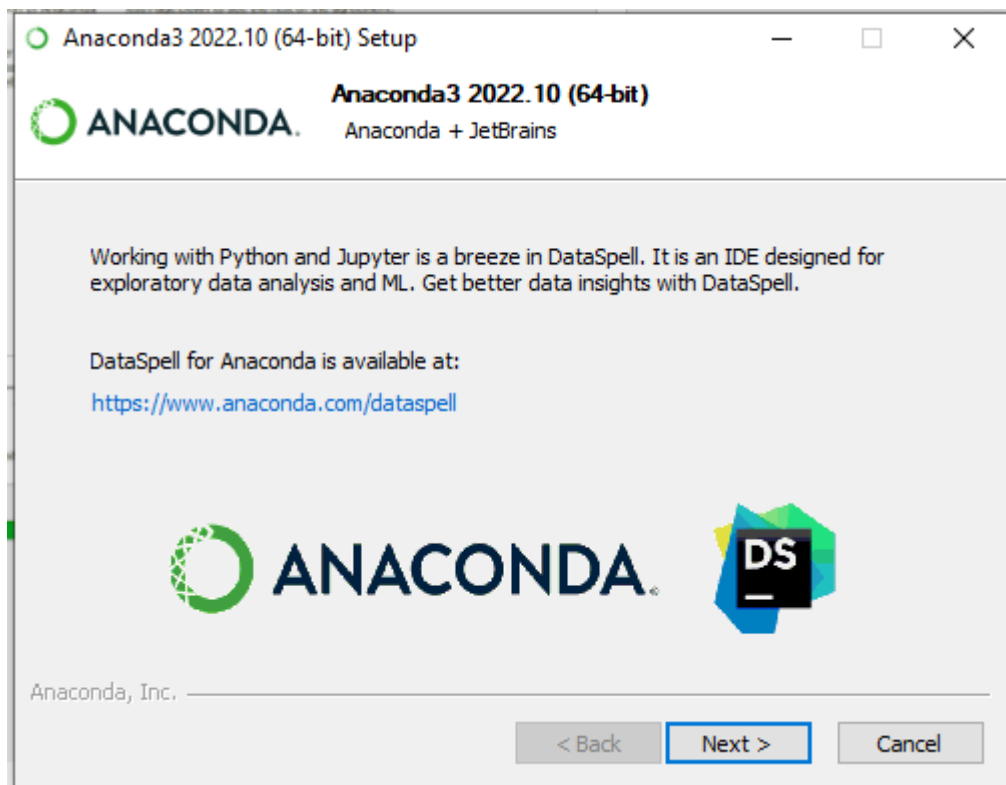
El proceso en sí es sencillo, podéis dejar las opciones por defecto o cambiarlas a vuestro gusto, igualmente aquí hay una pequeña guía visual del proceso:



Términos y condiciones de uso.

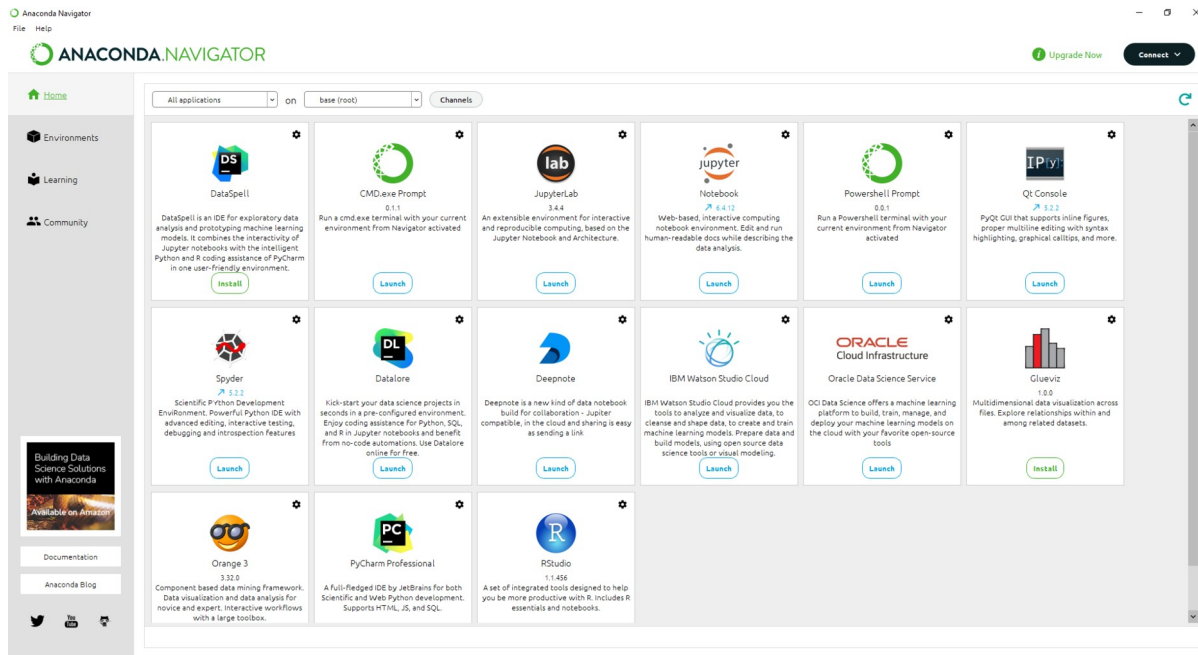






Después accederemos al navegador de anaconda. Si no hemos creado un acceso directo en el escritorio, lo podemos encontrar desde la barra de búsqueda de windows.

Una vez abierto, nos encontraremos con esta pantalla:



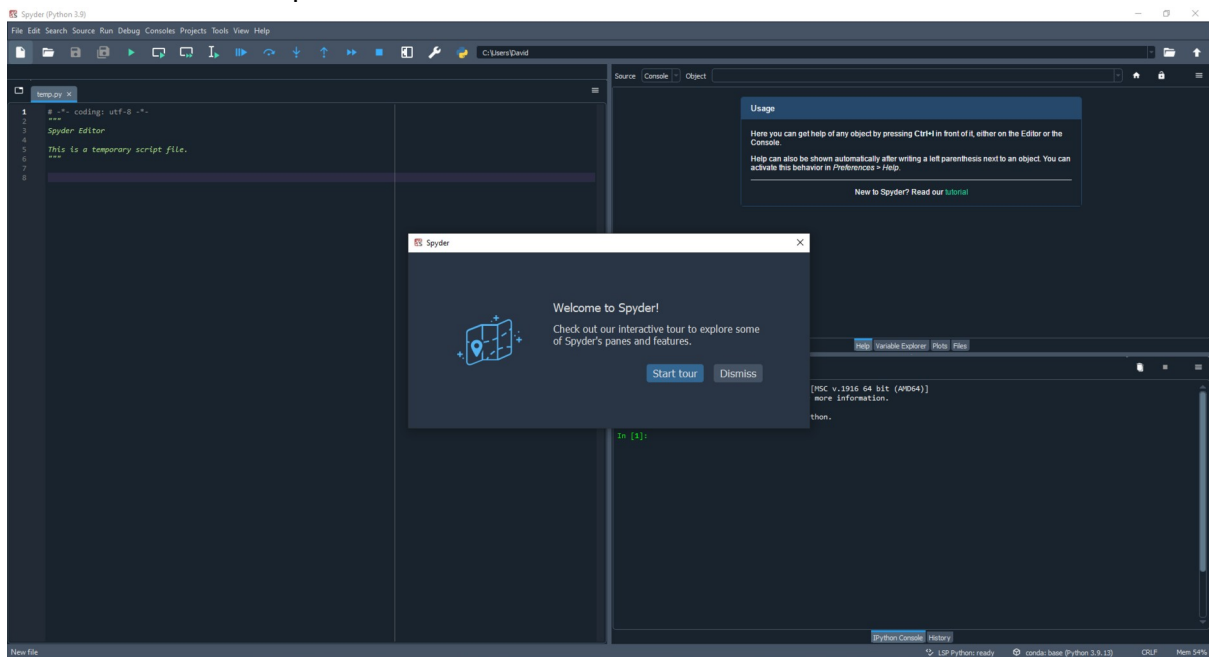
A la izquierda de la pantalla nos encontramos con un menú que contiene:

- **Home:** visualiza las aplicaciones instaladas para el actual entorno
- **Environments:** aquí podemos crear, modificar y borrar entornos de Python. Viene con uno creado por defecto. Estos entornos nos permiten encapsular los paquetes y distribuciones de python que necesitamos para diferentes formas de programación y así evitar conflictos, ya sea por versiones de los paquetes o el propio Python
- **Learning:** contiene una amplia serie de tutoriales
- **Community:** accesos directos a las principales páginas de ayuda en python (como StackOverflow)

Lanzando las aplicaciones

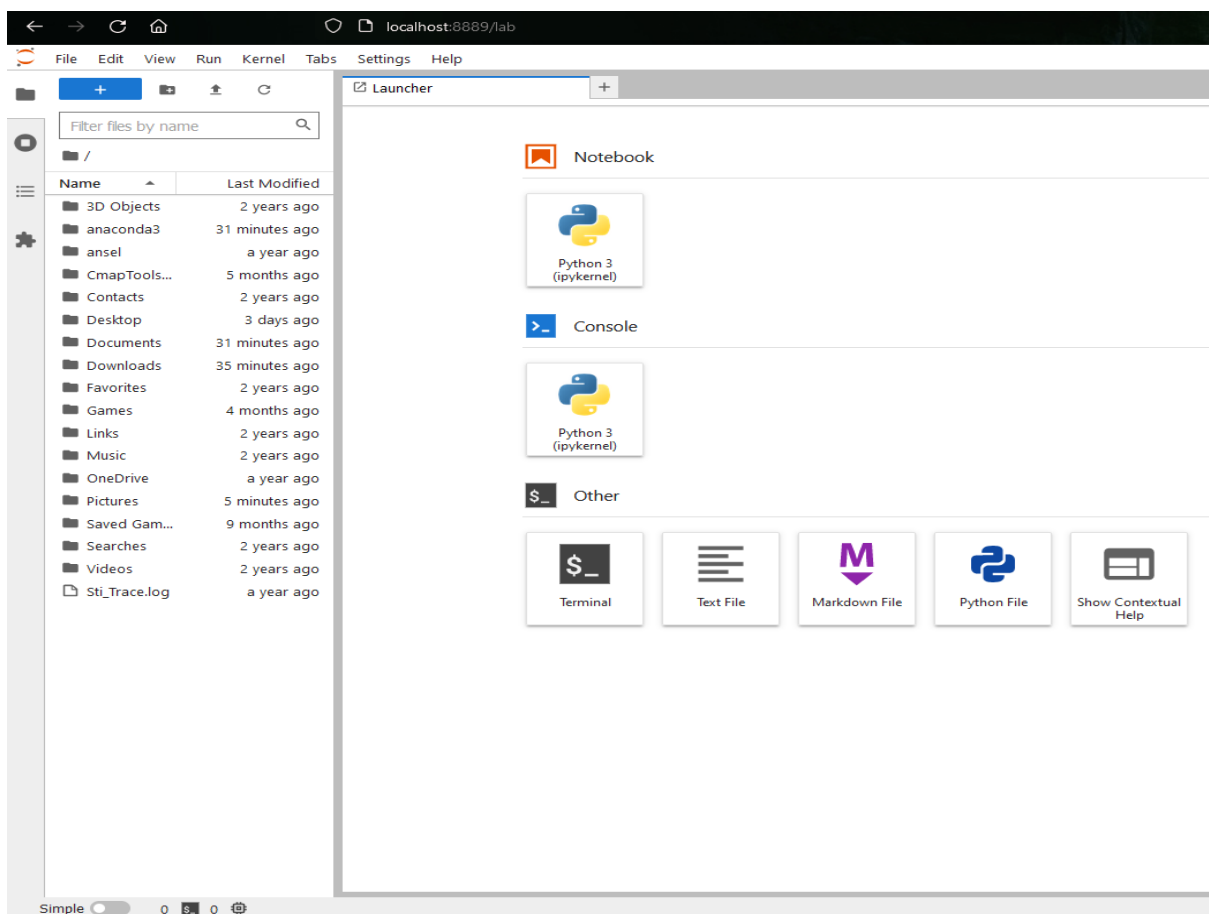
Por último, para comprobar que todo se ha instalado correctamente, lanzaremos tanto el Spyder como el JupyterLab (no confundir con Jupyter Notebook o simplemente llamado Notebook)

Al abrir el Spyder podremos acceder a un tutorial que nos mostrará las diferentes características de la aplicación.



Es muy recomendable seguir el tutorial para dominar las bases de Spyder

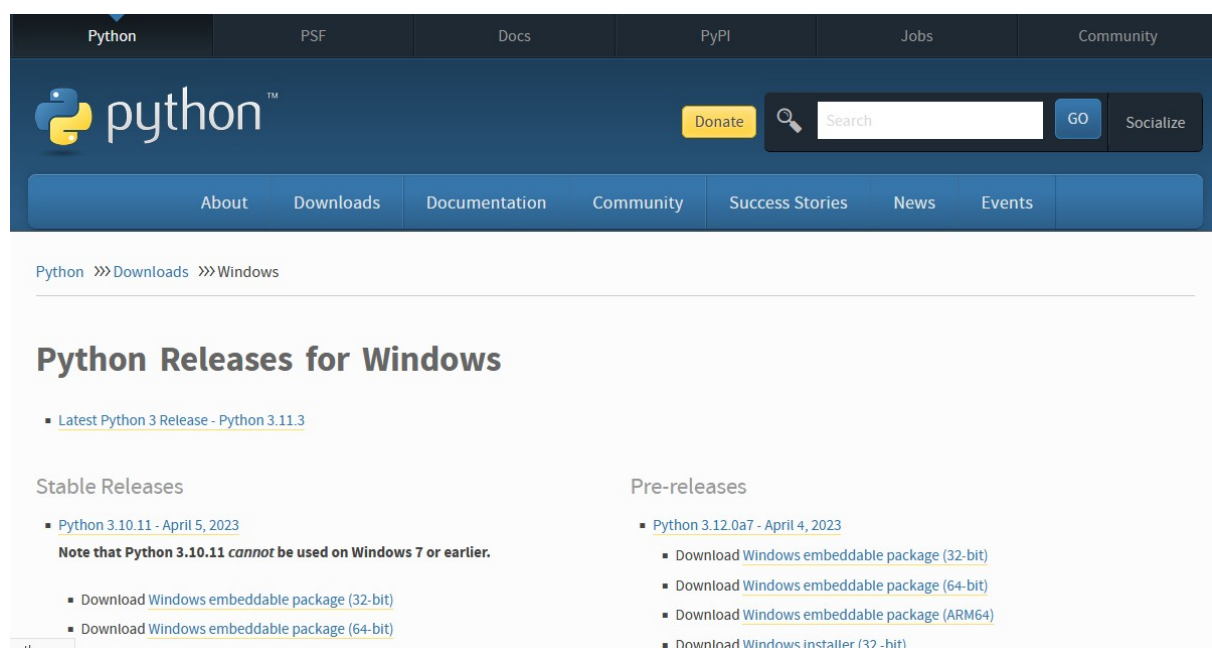
Por el contrario, cuando abrimos JupyterLab se nos abrirá una nueva pestaña en nuestro navegador por defecto.



Si nos fijamos, JupyterLab trabajará desde nuestra carpeta de usuario, por lo que es recomendable crear una carpeta donde guardar nuestros proyectos. Esto se puede hacer creando una carpeta directamente desde windows o creando desde el JupyterLab, haciendo click derecho en la ventana que nos muestra los directorios y archivos o, justo encima de la barra de búsqueda que se encuentra aquí, haciendo click en el símbolo de la carpeta con un “+” superpuesto.

Instalación de Python

Si quisiéramos trabajar con Python directamente desde nuestro sistema operativo Windows, podemos instalarlo sin necesidad de utilizar frameworks como Anaconda. Para ello iremos a la [página oficial de Python](#) y buscaremos la versión del lenguaje que más se adapte a nuestras necesidades.



The screenshot shows the Python.org website. The top navigation bar includes links for Python, PSF, Docs, PyPI, Jobs, and Community. Below this is a search bar and a 'Donate' button. The main content area is titled 'Python Releases for Windows'. It lists the latest Python 3 release as Python 3.11.3. Under 'Stable Releases', it shows Python 3.10.11 from April 5, 2023, with a note that it cannot be used on Windows 7 or earlier. Download links are provided for Windows embeddable packages (32-bit and 64-bit) and a Windows installer (32-bit). Under 'Pre-releases', it shows Python 3.12.0a7 from April 4, 2023, with download links for Windows embeddable packages (32-bit, 64-bit, and ARM64) and a Windows installer (32-bit).

Navegamos hasta la sección “Archivos” de la página y elegimos el ejecutable que queremos descargar para nuestro sistema operativo (seguramente uno de los dos últimos)

Files

Version	Operating System	Description	MDS Sum	File Size	PGP	Sigstore
Gzipped source tarball	Source release		7e25e2f158b1259e271a45a249cb24bb	26085141	SIG	.sigstore
XZ compressed source tarball	Source release		1bf8481a683e0881e14d52e0f23633a6	19640792	SIG	.sigstore
macOS 64-bit universal2 installer	macOS	for macOS 10.9 and later	f5f791f8e8bfb829f23860ab08712005	41017419	SIG	.sigstore
Windows embeddable package (32-bit)	Windows		fee70dae06c25c60cbe825d6a1bfda57	7650388	SIG	.sigstore
Windows embeddable package (64-bit)	Windows		f1c0538b060e03cbb697ab3581cb73bc	8629277	SIG	.sigstore
Windows help file	Windows		52ff1d6ab5f300679889d3a93a8d50bb	9403229	SIG	.sigstore
Windows installer (32-bit)	Windows		83a67e1c4f6f1472bf75dd9681491bf1	27865760	SIG	.sigstore
Windows installer (64-bit)	Windows	Recommended	a55e9c1e6421c84a4bd8b4be41492f51	29037240	SIG	.sigstore

Ejecutamos el programa de instalación:



IMPOTANTE: chequear la casilla “Add python.exe to PATH” o habrá que añadir python al path de windows manualmente.

Después seguís las instrucciones y, una vez terminada la instalación, podemos

- b. IOS
- c. Linux

4. Bases del lenguaje

Cuando abrimos una shell de python, podemos empezar a introducir instrucciones, por ejemplo:

```
(base) C:\Users\User>python
Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 2
4
>>>
```

Aquí hemos sumado $2 + 2$, esta instrucción es lo que se conoce como **expresión**, que consta de **valores** y **operadores**.

En el caso de Python, encontramos diferentes tipos de operadores, muy similares a los de otros lenguajes.

OPERADORES ARITMÉTICOS			
Símbolo	Nombre	Explicación	Ejemplo

+	suma		2 + 3 (=5)
-	resta		4 - 5 (=-1)
*	multiplicación		2 * 5 (=10)
/	división		10 / 4 (=2.5)
%	resto		8 % 3 (=2)
**	potencia		2**3 (=8)
//	división entera		8 // 3 (=2)

OPERADORES COMPARATIVOS

Símbolo	Nombre	Explicación	Ejemplo
==	Igualdad o equivalencia	Etor operadores devuelven un booleano (True / False) de la expresión que comparan. Por ejemplo: 3 == 2 >>> False 3 != 2 >>> True 3 < 2 >>> False	
!=	distinto		
>	mayor que		
<	menor que		
>=	mayor o igual que		
<=	menor o igual que		

OPERADORES LÓGICOS

Símbolo	Nombre	Explicación	Ejemplo
and	y	Son operadores lógicos que trabajan con variables booleanas, que normalmente son el resultado de expresiones con operadores comparativos. Por ejemplo: (x > 10) and (y == 20)	
or	o		
not	negado o no		

OPERADORES DE PERTENENCIA

Símbolo	Nombre	Explicación	Ejemplo
is		Devuelve True si ambos objetos son el mismo	
is not		Devuelve True si sendos objetos son diferentes	

OPERADORES DE MEMBRESÍA

Símbolo	Nombre	Explicación	Ejemplo
in		Devuelve True si el valor especificado está en el objeto	3 in x
not in		Devuelve True si el valor especificado no está en el objeto	3 not in x

OPERADORES DE ASIGNACIÓN			
Símbolo	Explicación	Ejemplo	Equivalencia
=	Operador asignación	x = 3	
+=		x += 3	x = x + 3
-=		x -= 3	x = x - 3
*=		x *= 3	x = x * 3
/=		x /= 3	x = x / 3
%=		x %= 3	x = x % 3
**=		x **= 3	x = x ** 3
//=		x //= 3	x = x // 3
&=	and (bitwise)	x &= 3	x = x & 3
=	or (bitwise)	x = 3	x = x 3
^=	xor	x ^= 3	x = x ^ 3
>>=	desplazamiento de bits a la dcha.	x >>= 3	x = x >> 3
<<=	desplazamiento de bits a la izda.	x <<= 3	x = x << 3

Además de los operadores, Python contiene una lista de caracteres especiales, sobre todo usados en las variables tipo string, y unas palabras reservadas que representan la base del lenguaje.

Caracter	Explicación	Ejemplo
\n	salto de línea	
\t	tabulación horizontal	
\r	vuelta de carro	
\b	borrar (backspace)	
\f	página siguiente (form feed)	

\'	comilla simple	
\"	comilla doble	
\\	backslash	
\v	tabulación vertical	
\UN	Caracter en Unicode, donde N es el número	'\U000000D1' es 'Ñ'
\NNN	Caracter donde NNN son dígitos en base octal	'\151' es 'i'
\xN	Caracter donde N son dígitos en base hexadecimal	'\x69' es 'i'
\a	sonido de campana del sistema	

así mismo, las palabras reservadas, donde también se cuentan algunos operadores, son:

Palabra	Significado	Palabra	Significado
False		global	para modificar una variable global dentro de una función
None		if	para hacer una declaración condicional, junto con 'else' y 'elif'
True		import	importar módulos completos
and	operador 'y'	in	
as	reasigna un objeto con un nuevo nombre	is	
assert	Utilizado para debugging, similar a la función print()	lambda	función anónima que puede tener cualquier número de argumentos, pero solo una expresión
break	Para salir de un bucle	nonlocal	para funciones dentro de funciones, declara que algo no es local.
class	para definir clases	not	
continue	saltar a la siguiente iteración del bucle	or	
def	definir función	pass	declaración nula, para añadir código en el futuro
del	borrar un objeto de la memoria	raise	obliga a lanzar un error
elif	equivalente a "else if"	return	salir de una función o método
else	acción alternativa a 'if' y 'elif'	try	parte del código que se intenta ejecutar
except	qué hacer en caso de excepción	while	declaración de un bucle while

	(Error, warning, etc.)		
finally	parte del (try/except), que obliga a ejecutar esta parte del código	with	se utiliza para simplificar el manejo de excepciones
for	definición del bucle for	yield	finaliza una función y devuelve el iterador
from	para incluir una parte específica de un módulo		

Con esto, podemos también ahora utilizar los tipos de dato predefinidos en Python

Categoría	Tipo	Explicación	Ejemplo
Texto	str	string o cadena de caracteres	x = 'hola' x = "hola"
Numérico	int	entero	x = 14
	float	coma flotante	x = 14.
	complex	número complejo (donde "j" es el imaginario)	x = 2 + 3j
Secuencia	list	lista (puede modificarse) de objetos que pueden ser de varios tipos	x = [2, 4, 9] x = ["uno", "dos"]
	tuple	tuple (no puede modificarse)	x = (4., 5.3, 7.)
	range	secuencia inmutable de números	x = range(10)
Mapeado	dict	guardan datos en parejas de "key": "value"	x = {"manzanas": 3, "peras": 17}
Set	set	guardan varios datos en una variable, pero los datos no pueden repetirse	x = {"peras", "manzanas", "piñas"}
	frozenset	set inmutable	
Booleano	bool	puede tener dos valores: True o False	x = True x = False
Binario	bytes	secuencia de enteros (de 0 a 255) inmutable	
	bytearray	igual que bytes pero admite cambios	
	memoryview	permite acceder el buffer interno de python de forma segura	
None	NoneType	Tipo nulo, sin valor	x = None

Es destacable también que los nombres de los tipos de dato sirven a su vez para hacer conversiones y definiciones de los mismos, es decir, que podemos definir, por ejemplo, una lista de las siguientes formas:


```
x = list()
x = []
```

En ambos casos crearía una lista vacía, pero podemos concatenar la llamada de funciones:

```
x = [0, 1, 2, 3, 4, 5]
x = list(range(6))
```

Finalmente, para dejar comentarios en python tenemos dos opciones:

- `#` para comentarios de línea
- `"""(...)"""` para un bloque de comentario

Por ejemplo:

```
# Este comentario es de una línea
x = 37 # se pueden añadir comentarios después de una sentencia

""" Este es un comentario de bloque,
que tiene varias líneas,
pero no acaba hasta que se cierre la triple comilla"""
```

a. Control de flujo

Antes de hablar del control de flujo per se, hay comentar la importancia de la indentación y la definición de bloques de código en Python.

Un bloque de código es un conjunto de sentencias en un lenguaje de programación, que a su vez puede contener otros bloques de código dentro de ellos, es decir, que se pueden anidar bloques de código. En Python, tanto las clases, como el cuerpo de las funciones, como los módulos ejecutados o un archivo tipo `.py` se consideran bloques de código, es decir cualquier conjunto de sentencias bien definidas y delimitadas.

Además, a diferencia de en otros lenguajes, los bloques de código vienen dados por el número de tabulaciones que se encuentran antes de la sentencia. Por ejemplo:

```
x = 10
if x <= 10:
    x += 1
    print(x)
else:
    print("El numero es mayor que 10")
```

En este caso, un bloque de código sería la parte que se ejecuta si el número “x” es menor o igual que 10, otra parte si no lo es y total del código en su conjunto.

Dicho esto, podemos empezar con el control de flujo propiamente dicho en Python.

Python, al ser un lenguaje de programación secuencial, ejecutará de forma sistemática las sentencias que se le ordenen, pero a veces queremos que solamente se ejecuten ciertas partes dependiendo de las circunstancias, así pues, tenemos las instancias **if**, **elif** y **else**.

Una instancia **if** ejecutará el bloque de código contenida en su interior si la instancia de la que depende tiene como resultado un booleano de valor **True** (aunque a veces con la existencia de un tipo, que el valor sea distinto de 0 o con que este contenga al menos un elemento valdrá).

```
if (x > 10) and (y == 4):  
    (...) # esto solamente se ejecutará si x es mayor que 10 e y  
    es igual a 4
```

Por otro lado, a veces hay una dicotomía, en la que hay que elegir ejecutar una parte del código u otra, aquí usaremos el **else**:

```
if x == 25:  
    (...) # esto solamente se ejecutará si x es igual a 25  
else:  
    (...) # esto se ejecutará si x es distinto de 25
```

Finalmente, en ciertas ocasiones hay que elegir entre más de dos opciones, es ahí donde podemos usar la sentencia **elif** (abreviatura de **else if**).

```
if x == 25:  
    (...) # esto solamente se ejecutará si x es igual a 25  
elif x > 50:  
    (...) # esto se ejecutará si x es mayor que 50  
else:  
    (...) # esto se ejecutará si x es distinto de 25 y menor o  
    igual a 50
```

Es importante destacar, que en caso de cumplirse dos condiciones a la vez, se ejecutará solamente el código de la primera escrita, ya que python recorre las condiciones de forma secuencial.

Por otro lado, tenemos los bucles, que ejecutarán un bloque de código mientras sean ciertas las condiciones de su definición.

En python tenemos dos tipos de bucles: el bucle **while** y el **for**.

El bucle **while** se define de la siguiente manera:

```
while condition:  
    (...)
```

donde “condition” es una condición lógica que debe ser cierta para que se ejecute la siguiente iteración del bucle.

Por otro lado el bucle for itera sobre los elementos de una secuencia (una lista, tupla, diccionario, set o string) y se define:

```
for element in sequence:
    (...)
```

Donde “element” es cada elemento de la secuencia “sequence”.

Es importante destacar que, tanto los bucles como las instancias lógicas pueden estar anidados. Por ejemplo:

```
for i in range(10):
    if i < 5:
        for j in range(10):
            print(j)
    else:
        print("asd")
```

Por último tenemos dos sentencias de control en los bucles: **break** y **continue**. Break sirve para salir del bucle actual, mientras que continue salta directamente a la siguiente iteración sin ejecutar el resto de código de la iteración actual.

```
for i in range(1000):
    if i == 10:
        print("Numero 10")
        continue
    print(i)
    if i > 100:
        break
```

También podemos hacer uso de pass si queremos dejar un bucle vacío para ser rellenado más adelante para evitar que salte un error:

```
for elem in element_list:
    pass
```

b. Listas, diccionarios y strings

Dentro de los tipos secuenciales, las listas, los diccionarios y las string son los más utilizados.

Tanto estos tipos secuenciales como el reto no mencionados en esta sección tienen acceso a una serie de funciones y operadores comunes, así como una lógica de acceso a sus elementos internos similar. Pongamos, por ejemplo la cadena:

```
x = 'hola mundo'
x = "hola mundo"
```

En este caso, el índice de los elementos sería:

x	h	o	l	a		m	u	n	d	o
índice incremental	0	1	2	3	4	5	6	7	8	9
índice decremental	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Así pues, podemos acceder a los elementos de la cadena utilizando los corchetes “[” y “]”

```
x[0] # accede al primer elemento
x[-1] # accede al último elemento
x[i:j] # accede a los elementos de i a j-1
x[i:j:n] # accede a los elementos de i a j-1 de n en n
x[:j] # accede a los elementos desde el primero hasta el anterior a j
x[j:] # accede a los elementos desde el j hasta el final
x[-k:] # accede a los últimos k elementos
x[-k:-1] # accede a los elementos desde -k hasta el anterior a -1
```

Un operador utilizado frecuentemente es el de concatenación, es decir, “+” y “+=”.

```
x += ' que tal?' daría la cadena 'hola mundo que tal?'
```

Algunas funciones que comúnmente se utilizan son **len()** e **index()**.

len(x) da la longitud del objeto según su número de elementos.

x.index(elem) es un método que devuelve el índice del elemento elem

Como ya se ha mencionado anteriormente, un string es una cadena de caracteres, es decir, un conjunto de varios caracteres alfanuméricos dentro del estándar Unicode. Podemos inicializar o definir un string por medio de las comillas simples o dobles, pues Python no realiza distinción alguna entre ambas.

```
x = 'hola mundo'
x = "hola mundo"
```

A su vez, la función str() permite convertir un tipo de dato a una string

```
n = 17
x = str(n)
```

En cuanto a los métodos propios de la clase string, nos encontramos con algunos muy típicos:

método	explicación	ejemplo
capitalize()	pone la primera letra de la string en	str.capitalize()

	mayúscula	
casefold()	convierte la string a minúsculas	
count()	número de veces que una string aparece en otra	
endswith()	devuelve True si la string acaba con los caracteres especificados	
find() index()	devuelven la posición de un elemento, que no se sabe si existe en la cadena en el caso de find()	
join()	une los elementos de un iterable según la cadena	' '.join(list_of_words)
strip() lstrip() rstrip()	quita los caracteres especificados de la cadena (por defecto el caracter espacio), en el caso de lstrip() y rstrip() los quita de l principio y del final de la cadena respectivamente	
split() splitlines()	divide la cadena de caracteres por el espacio (u otro caracter) o el salto de línea ('\n') respectivamente	

Para ver el resto de métodos que corresponden a la clase string:

<https://docs.python.org/3/library/stdtypes.html#string-methods>

https://www.w3schools.com/python/python_strings_methods.asp

Las listas, por otro lado, se definen de la siguiente forma:

```
x = [1, 2, 3]
x = list(1, 2, 3)
```

Una lista puede contener elementos de cualquier tipo, incluso otras listas, lo que formaría una matriz. en este caso se puede acceder a cada elemento de la siguiente forma:

```
x[i] # accede al elemento tipo lista en la posición i
x[i][j] # accede al elemento j de la lista i
```

Se pueden aplicar las reglas de indexación explicadas anteriormente:

```
x[i:j][k:l] accede a los elementos de k a l de las listas de i a j
```

Es importante denotar que, al igual que no todos los elementos de una lista tienen por qué tener la misma longitud, tampoco tienen por qué tener la misma longitud, por lo que es importante tener cuidado para no acceder a elementos inexistentes:

```
x = [[1, 2, 3], [4, 5], [6, 7, 8, 9, 10]]
```

Las listas, a su vez, tienen una serie de métodos propios de su clase:

Método	Explicación	Ejemplo
append()	añade un elemento a la lista	
clear()	borra todos los elementos de la lista	
copy()	devuelve una copia de la lista	
count()	devuelve el número de elementos	
extend()	añade varios elementos a la lista, es equivalente a + o +=	lst_1.extend(lst_2) lst_1 += lst_2
index()	devuelve el índice de un elemento	
insert()	inserta un elemento en la posición especificada	
pop()	elimina el elemento de la posición especificada	
remove()	elimina el elemento que coincide con lo que se le pasa	
reverse()	da la vuelta a la lista	
sort()	ordena la lista según un criterio	

Por último, los diccionarios son un tipo de dato complejo que contienen información en parejas de **key:value**, esto es, de clave:valor. Por ejemplo:

```
alumno = {"nombre": "Paco",  
          "apellido": "García",  
          "edad": 20}
```

Para crear y definir un diccionario podemos hacerlo directamente o ir añadiendo sus campos uno a uno:

```
alumno = {} # o bien alumno = dict()  
alumno['nombre'] = "Paco"  
alumno['apellido'] = "García"  
alumno['edad'] = 20
```

Indirectamente, podemos ver aquí la forma de acceder a los elementos de un diccionario, que, en lugar de con un número entero, es con el nombre de la clave, mientras que, para ir añadiendo campos nuevos al diccionario solamente tenemos que definirlos con una clave nueva.

Así pues, tenemos también una lista de métodos de diccionario que podemos utilizar:

Método	Explicación	Ejemplo
clear()	borra todos los elementos del diccionario	
copy()	devuelve una copia del diccionario	
fromkeys()	devuelve un diccionario con las claves especificadas y sus valores	
get()	devuelve el valor de la clave especificada	
items()	devuelve una lista de tuplas de la forma (key, value)	
keys()	devuelve una lista de las claves	
pop()	elimina el elemento de la clave especificada	
popitem()	elimina la última pareja de clave y valor insertada	
setdefault()	devuelve el valor de la clave especificada, si no existe, la crea con el valor especificado	
update()	actualiza el diccionario con las claves y valores dados	
values()	devuelve una lista de los valores del diccionario	

Es muy común, a su vez, iterar sobre listas, diccionarios o string, lo cual normalmente se hace por medio de un bucle for. Por ejemplo, para sacar por pantalla los elementos de una lista, podemos:

```
for i in range(len(elem_list)):  
    print(elem_list[i])
```

Pero, teniendo en cuenta que podemos iterar sobre una lista, es más simple hacer:

```
for elem in elem_list:  
    print(elem)
```

Sin embargo, si queremos trabajar con los elementos de una lista, podemos usar:

```
for elem in elem_list:
    if elem % 2 == 0:
        even_list.append(elem)
```

Sin embargo, Python permite trabajar con listas en una sola línea, lo cual, en aras de la legibilidad, puede ser preferible. Para el ejemplo anterior, sería:

```
even_list = [elem for elem in elem_list if elem % 2 == 0]
```

Es importante saber que la sintaxis de una línea para un condicional if y para un if/else son diferentes.

```
out_lst = [value for elem in in_lst if condition]
out_lst = [value if condition else default_value for elem in in_lst]
```

Por ejemplo:

```
in_lst = list(range(10))
out_lst_1 = [elem for elem in in_lst if elem % 2 == 0]
out_lst_1 = ['par' if elem % 2 == 0 else 'impar' for elem in in_lst]
```

Esto viene a su vez de la capacidad de poner expresiones condicionales en una sola línea:

```
n = 10
if n % 2 == 0:
    m = 'par'
else:
    m = 'impar'
```

Sería equivalente a:

```
m = 'par' if n % 2 == 0 else 'impar'
```

c. Funciones

Las funciones son bloques de código que solamente se ejecutan cuando son llamadas explícitamente.

Además de las funciones que contiene Python, nosotros podemos definir nuestras propias funciones utilizando **def**.

```
def funcion():
    print("Soy una funcion")
```

Para llamar a dicha función tenemos que ejecutar la sentencia:

```
funcion()
```


Además, a una función podrán pasársele diferentes tipos de argumentos o parámetros para que trabaje con ellos.

Los argumentos de palabra clave son aquellos de los cuales estamos obligados a pasar un valor para la ejecución de la función:

```
def funcion(arg1, arg2):  
    print(arg1, "y", arg2)  
  
funcion(4, "blabla")
```

Estos argumentos, a su vez, pueden tener valores por defecto para el caso de que no queramos pasarle siempre un valor o haya un valor que se use comúnmente.

```
def funcion(arg1, arg2 = "hola"):  
    print(arg1, "y", arg2)  
  
funcion(4)
```

Los argumentos con valores por defecto deberán ir siempre detrás de aquellos que no lo tienen.

Por otro lado, si desconocemos el número de parámetros que se le va a pasar a una función, podemos hacer uso de los ***args** y ****kwargs**. Los ***args** son los argumentos arbitrarios (arbitrary arguments) y, con ellos, la función recibirá una tupla de argumentos, mientras que con los ****kwargs**, o argumentos arbitrarios de palabras clave (keywords arbitrary arguments) la función recibe un diccionario de argumentos con sus palabras clave)

```
def receta(*ingredientes):  
    for elem in ingredientes:  
        print(elem)  
receta("pasta", "tomate", "queso")  
  
def nombre_completo(**kwargs):  
    print("apellido:" kwargs['apellido'])  
nombre_completo(nombre = 'Paco', apellido = 'García', edad = 20)
```

También es importante el uso de la palabra reservada **return**, por el cual la función devolverá uno o varios elementos, ya sean valores concretos o expresiones.

```
def suma_3(x):  
    return x + 3  
  
print(suma3(2))
```

También se pueden devolver expresiones lógicas o varios argumentos, por ejemplo:

```
def paridad(n):  
    return 'par' if n % 2 == 0 else 'impar'
```

```
def mayores_que_100(x, y):  
    return x > 100 and y > 100
```

Es necesario aclarar que, aunque la expresión del return no es necesaria para la correcta definición de una función, sí que puede ayudar a delimitar el código mientras se escribe y no da error devolver nada

Una función podrá llamarse a sí misma en lo que es conocido como recursión, pero es una estrategia arriesgada, pues es fácil caer en bucles infinitos.

```
def funcion(i):  
    if i > 10:  
        return  
    i += 1  
    print(i)  
    funcion(i)  
funcion(0)
```

Es, a su vez, una buena práctica el intentar evitar hacer demasiadas anidaciones, para que así el código quede lo más limpio posible.

Es posible definir una función dentro de otra función, lo que se conoce como funciones anidadas. Esta función interna estará dentro del namespace de la función superior, por lo que solamente podrá ser llamada desde dentro de la misma.

```
def funcion_1():  
    def funcion_2():  
        ...  
        return  
  
    funcion_2()  
    return  
  
funcion_1()  
funcion_2() # Dará error, pues no está definida fuera del  
            namespace de funcion_1
```

5. Tratamiento de archivos

Un archivo es un conjunto de datos guardados en una serie de posiciones dentro de una memoria no volátil, que puede ser creado, accedido, leído y editado si se cumplen ciertas condiciones.

Para trabajar con archivos, Python proporciona una serie de funciones que permiten crearlos, abrirlos, leerlos, editarlos, cerrarlos y borrarlos.

Podemos abrir (o crear) un archivo por medio de la función `open()`, la cual admite 3 argumentos principalmente, el primero es el nombre del archivo, el segundo el modo de apertura y el tercero la codificación del archivo (por defecto 'utf-8').

```
open(file, mode='r', buffering=-1, encoding=None, errors=None,
      newline=None, closefd=True, opener=None)
```

El nombre del archivo también deberá contener su path en caso de que este esté en una o directorio diferente al actual.

Modo	Nombre	explicación	ejemplo
"r"	read	Valor por defecto. Abre el archivo para leerlo	f = open("example.txt") f = open("example.txt", 'r')
"a"	append	Abre para añadir al final, si no existe, crea el archivo	f = open("example.txt", 'a')
"w"	write	Abre para escribir, si no existe, crea el archivo	f = open("example.txt", 'w')
"x"	create	Crea el archivo, da error si ya existe	f = open("example.txt", 'x')
"t"	text	Modo texto (para archivos de texto). Añadido a 'r', 'a', 'w', 'x'	f = open("example.txt", 'rt')
"b"	binary	Modo binario (para otro tipo de archivos). Añadido a 'r', 'a', 'w', 'x'	f = open("example.png", 'rb')
"+"	update	Abre un archivo para ser actualizado	f = open("example.png", 'r+b')

Es importante que, si se abre un archivo, se cierre después de realizar las operaciones necesarias con él. Esto puede hacerse explícitamente por medio del método **close()** de la clase archivo, o implícitamente por medio de la palabra reservada **with**:

```
f = open("example.txt", 'r')
(...)
f.close()
```

```
with open("example.txt", 'r') as f:
    (...)
```

En ambos casos se hará lo mismo.

Además, la clase file (archivo) posee una serie de métodos que permiten modificar el contenido del mismo.

Método	Explicación
--------	-------------

close()	cierra el archivo
detach()	devuelve un raw stream separado para el buffer
fileno()	devuelve un número que representa el buffer interno, desde el punto de vista del sistema operativo
flush()	limpia el buffer interno
isatty()	devuelve si el archivo es interactivo o no
read()	devuelve el contenido del archivo
readable()	devuelve si el archivo es legible o no
readline()	lee una línea del archivo
readlines()	crea una lista con las líneas del archivo
seek()	cambia la posición del puntero del archivo
seekable()	devuelve si es posible cambiar la posición de dicho puntero
tell()	devuelve la posición actual del puntero en el archivo
truncate()	redimensiona el archivo a un tamaño determinado
writable()	devuelve si se puede escribir en el archivo
write()	escribe una string en el archivo
writelines()	escribe una lista de strings en el archivo

Veamos tres ejemplos de cómo trabajar con la lectura, escritura y añadido de datos a los archivos:

```
with open("file1.txt", 'rt') as f:
    lines = f.readlines()
for i, line in enumerate(lines):
    print(i, line)

lines = ["line1\n", "line2\n", "line3\n"]
line_extra = "extra line"
with open("file2.txt", 'wt') as f:
    f.writelines(lines)
    f.write(line_extra)

with open("file3.txt", 'a+') as f:
    f.write("\nNEW LINE\n")
```

6. Clases y objetos

a. Introducción

La programación orientada a objetos (OOP) es un paradigma de programación que se basa en los “objetos”. Un objeto consta de un estado (datos que almacena) y de un comportamiento (tareas o funciones que puede realizar), así como de una identidad que lo diferencia de otros objetos.

Algunas de sus propiedades principales son:

- **Abstracción:** no es necesario conocer el funcionamiento interno de un objeto para utilizar sus métodos.
- **Encapsulamiento:** toda la información relevante sobre un objeto está dentro del mismo.
- **Polimorfismo:** puede utilizarse el mismo nombre para llamar a comportamientos distintos en objetos distintos (por ejemplo, sobreescritura de operadores).
- **Herencia:** una clase puede heredar las propiedades de otra clase superior.
- **Modularidad:** la aplicación puede dividirse en partes más pequeñas tan independientes como sea posible del resto
- **Principio de ocultación:** cada objeto está aislado del resto y se relacionan a través de interfaces, evitándose así modificaciones por parte de usuarios no autorizados
- **Recolección de basura:** los objetos se destruyen automáticamente y liberan memoria.

b. Namespaces

Antes de hablar de clases y objetos, debemos definir dos conceptos en Python: los namespace y el scope.

Es bien sabido que Python es un lenguaje de programación orientado a objetos y en el cual todo lo que utilizamos es un objeto de una clase, por ejemplo, los tipos de datos en Python son en realidad clases, que contienen una serie de atributos y métodos.

Un **nombre**, o **identificador**, es aquello con lo que llamamos a un objeto en concreto. Por ejemplo, si tenemos la simple sentencia:

```
a = 2
```

En este caso, “2” sería el objeto creado y “a” el nombre asociado con él. Para poder observar su posición en memoria utilizaremos la función **id()**, de forma que si aplicamos dicha función a “a” y a “2” deberíamos obtener el mismo resultado, ergo ambas se refieren al mismo objeto.

```
print(id(a))  
print(id(2))
```

Pongamos ahora una serie de sentencias:

```
a = 2
```

```

print(id(a))
print(id(2))

a = a + 1
print(id(a))
print(id(3))

b = 2
print(id(b))
print(id(2))

```

Así podremos observar una serie de cambios a lo largo de las tres sentencias principales.

- En “a = 2”, se crea el objeto “2” y se le asocia como nombre “a”.
- En “a = a + 1”, se crea un nuevo objeto, “3”, y a pasa “a” ser su nombre, es decir que cambia de ser el nombre de “2” a ser el nombre de “3”.
- Finalmente, 2 no se ha eliminado de memoria, sigue por ahí flotando, y le asociamos un nuevo nombre, “b”. De forma que “a” sigue siendo el nombre de “3” y “b” es ahora el nombre de “2”.

Pero no solo los objetos de datos se benefician de esta propiedad de Python, las funciones que definimos son, a su vez, objetos también, de forma que puede asignárseles un nombre nuevo.

```

def funcion(n):
    print(f"pasado {n}, devolviendo {n**2}")
    return n**2
a = funcion
y = a(4)
print(y)

```

Así pues, un **namespace** es el conjunto de nombres definidos para objetos concretos. En Python pueden coexistir diferentes namespaces, aunque están aislados unos de otros. Cuando iniciamos el intérprete de Python, se crea un primer namespace general, que es el que permite que tengamos acceso a las diferentes utilidades propias del lenguaje (como usar funciones básicas del estilo `print()`). Por otro lado, cada módulo contiene su propio namespace, que importamos a un script al importar el módulo. Es cada script definimos, a su vez, nuestro propio namespace, así como namespaces locales dentro de las utilidades e nuestro script, como en las funciones.

Sin embargo, aunque tengamos todos estos namespaces dentro de nuestro script, no siempre podremos acceder a ellos desde ciertas partes del programa. En esencia, un scope es una parte de un programa desde la que se puede acceder a los nombres de un namespace sin necesidad de prefijos.

Por ejemplo:

```

def funcion_1():
    a = 20

```

```

def funcion_2():
    a = 30
    print("a =", a)
    funcion_2()
    print("a =", a)

a = 10
funcion_1()
print("a =", a)

```

En esta función, hay 3 variables llamadas "a" que pertenecen a 3 namespaces diferentes:

1. a = 10: pertenece al namespace global del programa
2. a = 20: pertenece al namespace de funcion_1
3. a = 30: pertenece al namespace de funcion_2

Sin embargo, si en cada caso utilizamos la palabra reservada "global", podemos acceder a los namespaces de niveles superiores:

```

def funcion_1():
    global a
    a = 20
    def funcion_2():
        global a
        a = 30
        print("a =", a)
    funcion_2()
    print("a =", a)

a = 10
funcion_1()
print("a =", a)

```

En este caso, las tres "a" tendrán el mismo valor, pues estamos usando el nombre de una variable de un namespace superior dentro de cada una de las funciones.

c. Clases y objetos

Volviendo al tema que nos incumbe. Un **objeto** puede contener dos tipos de **atributos**: **métodos** y **propiedades**. Las propiedades son variables internas del objeto, mientras que los métodos son funciones propias del objeto.

```

class Five:
    x = 5
    def suma(self, y):
        return self.x + y

obj = Five()

```

```
print(obj.suma(3))
```

En Python, el constructor de la clase se realiza con el método `__init__()` que tienen todos los objetos. Si lo declaramos explícitamente podemos además modificarlo para que inicialice el objeto de la forma que nosotros queremos.

```
class Persona:
    especie = 'homo sapiens'

    def __init__(self, edad, nombre, apellido, apodo = ""):
        self.edad = edad
        self.nombre = nombre
        self.apellido = apellido
        self.apodo = apodo
    def saludar(self):
        print("hola!")

per1 = Persona(30, 'Jorge', 'Jiménez', 'el pelos')
per2 = Persona(20, 'Gustavo', 'Gutiérrez')
```

Si queremos que al llamar directamente al objeto, nos salga por pantalla cierta información del mismo, podemos hacer uso de la función `__str__()`. Para el caso anterior, dentro de la definición de la clase pondríamos:

```
def __str__(self):
    return f"{self.nombre} {self.apellido} {self.edad}"
```

De forma que al hacer:

```
print(per1)
```

Salga por pantalla:

```
Jorge Jiménez 30
```

Estas funciones con doble barra baja son lo que se conoce como los Métodos Especiales de Python, a veces también llamados métodos dunder (de “double underscore”). Estos métodos pueden invocar sintaxis especiales de Python y son la forma que tiene este lenguaje de sobrecargar operadores. Por ejemplo, pongamos que definimos una clase similar a una lista, para poder acceder a cada uno de sus elementos, podemos sobrescribir la función `__getitem__`, pudiendo usar así la sintaxis `objeto[i]`. Viendo un ejemplo más visual, las dos siguientes sintaxis serían equivalentes:

```
x = list(...)
print(x[index])
print(x.__getitem__(index))
```


Para sobrescribir un operador en python y poderlo utilizar de la forma que queramos con nuestra clase, tendremos que acceder al método especial del operador en la definición de la clase. Por ejemplo, si definimos una clase vector y luego queremos sumar dos vectores de esa clase, crearíamos algo así:

```
class vector():
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def __add__(self, other):
        if isinstance(other, int) or isinstance(other, float):
            # Si other es un int o un float lo sumamos a cada
            # miembro
            x = self.x + other
            y = self.y + other
            z = self.z + other
            # devolvemos un nuevo vector
            return vector(x, y, z)
        if isinstance(other, vector):
            # si other es de la clase vector sumamos self y
            # other término a término
            x = self.x + other.x
            y = self.y + other.y
            z = self.z + other.z
            # devolvemos un nuevo vector
            return vector(x, y, z)
```

Así, ahora podemos usar:

```
a = vector(1,2,3)
b = vector(4,5,6)
print(a + b)
```

Todos los métodos especiales se encuentran en la documentación de Python:

<https://docs.python.org/3/reference/datamodel.html#special-method-names>

Se pueden crear además clases que heredan las propiedades de otra. Continuando con el ejemplo de los estudiantes, podemos definir una nueva clase “estudiante”, que hereda de persona. En una clase heredada podemos cambiar los métodos de la clase madre, así como añadir unos nuevos.

```
class Estudiante(Persona):
    universidad = "Complutense"
    def __init__(self, edad, nombre, apellidos, matricula):
        Persona.__init__(self, edad, nombre, apellidos)
```

```

        self.matricula = matricula
    def saludar(self):
        print("Ahora no tengo tiempo, que tengo exámenes")
    def despedirse(self):
        print("Adios!")

```

Para llamar a los métodos de la clase madre, podemos usar la sentencia `super`, de forma que, por ejemplo, el constructor quedase:

```

def __init__(self, edad, nombre, apellidos, matricula):
    super().__init__(edad, nombre, apellidos)
    self.matricula = matricula

```

Se puede dar el caso de herencia múltiple, en la cual una clase herede de más de otra clase, ya sea porque a su vez la madre es una clase heredada de otra, o porque se hereda directamente de varias clases a la vez.

```

class clase_1():
    (...)
class clase_2(clase_1):
    (...)
class clase_3(clase_2):
    (...)

class clase_4():
    (...)
class clase_5():
    (...)
class clase_6(clase_4, clase_5):
    (...)

```

Si existe una superposición de métodos llamados de la misma forma, la incongruencia se soluciona por el MRO (Method Resolution Order), el cual dicta que un método se resolverá primero según la clase actual, y si no, en la definición de las múltiples clases de izda. a dcha. o, si no, en niveles de herencia superiores.

En cuanto al polimorfismo, lo hemos visto ya indirectamente en Python, por ejemplo, en funciones que pueden admitir diferente número de argumentos o la longitud de los mismos.

El polimorfismo es la capacidad de utilizar un mismo nombre para referirse a procesos diferentes, puede aplicarse tanto a clases, como funciones y métodos, como variables. Lo cierto es que en Python el polimorfismo está definido implícitamente, aunque podemos verlo con algunos ejemplos:

```

print(len("hola"))
print(len([1, 2, 3]))

```

En este caso la función `len()` actúa polimórficamente sobre dos tipos de objetos de datos diferentes, un string y una lista.

Podemos verlo también:

```
class apple():
    def color():
        print("Red")
class pear():
    def color():
        print("Green")

def show_color(obj):
    obj.color()

obj_apple = apple()
obj_pear = pear()

show_color(obj_apple)
show_color(obj_pear)
```

d. Atributos públicos, protegidos y privados

Un atributo es público si es accesible desde fuera de la clase y no solamente desde sus métodos internos. Por ejemplo:

```
class persona():
    def saludo(self):
        print("Hola")

paco = persona()
paco.saludo()
```

Podemos acceder al método `saludo()` del objeto "paco" llamándolo directamente como `paco.saludo()`. Esto es lo que se conoce como **atributo público**, aquel que puede ser accedido desde cualquier parte dentro y fuera de la clase.

Por otro lado, si queremos que, por seguridad o privacidad, ciertos métodos y propiedades no sean accesibles directamente, podemos declararlos como protegidos o privados.

Un **atributo privado** es aquel que solamente puede ser accedido desde la propia clase en la que está definido. Por ejemplo:

```
class persona():
    def __init__(self, nombre):
        self.__nombre = nombre
```

```
paco = persona("Paco")
print(paco.__nombre) # Dará ERROR
```

Ahora, si intentamos acceder al atributo privado `__nombre` directamente, obtendremos un error. Habremos de acceder a él indirectamente por medio de un método público, por ejemplo:

```
class persona():
    def __init__(self, nombre):
        self.__nombre = nombre
    def mostrar_nombre(self):
        return self.__nombre

paco = persona("Paco")
print(paco.mostrar_nombre()) # Esto sí funcionará
```

Los atributos privados heredados de una clase superior deben ser accedidos a su vez llamando a algún método que devuelva su valor desde la clase padre.

```
class persona():
    def __init__(self, nombre):
        self.__nombre = nombre
    def mostrar_nombre(self):
        return self.__nombre

class estudiante(persona):
    def __init__(self, nombre):
        super().__init__(nombre)
    def mostrar_nombre_estudiante(self):
        return super().mostrar_nombre()
    # NO PODEMOS hacer:
    # return self.__nombre
```

```
paco = estudiante("Paco")

# Al ser ambas métodos públicos ambas funcionarán
print(paco.mostrar_nombre())
print(paco.mostrar_nombre_estudiante())
```

Si queremos que un atributo sea accedido desde las clases heredadas, podemos declararlo como atributo protegido.

```
class persona():
    def __init__(self, nombre):
        self._nombre = nombre
    def mostrar_nombre(self):
```

```

        return self._nombre

class estudiante(persona):
    def __init__(self, nombre):
        super().__init__(nombre)
    def mostrar_nombre_estudiante(self):
        # Ahora accedemos directamente al atributo protegido
        return self._nombre

paco = estudiante("Paco")

# Al ser ambas métodos públicos ambas funcionarán
print(paco.mostrar_nombre())
print(paco.mostrar_nombre_estudiante())

```

En resumen:

tipo de atributo	sintaxis	acceso desde la clase	acceso desde sub-clases	acceso desde todas partes
público	nombre	Sí	Sí	Sí
protegido	_nombre	Sí	Sí	NO*
privado	__nombre	Sí	NO*	NO*

* El acceso se realiza por medio de métodos públicos de la clase y la subclase

7. Librerías y módulos

Una librería o módulo es un archivo o conjunto de archivos que contienen definiciones de funcionalidades (como clases o funciones) que pueden importarse a otros scripts o programas para ser utilizadas.

La principal utilidad de un módulo es la de poder utilizar lo que está definido en él en varios scripts diferentes, ahorrándonos así el tener que definirlos localmente en cada uno de ellos.

Por ejemplo, supongamos que definimos una función "funcion()" que va a ser utilizada en 3 scripts (Ejemplo1.py, Ejemplo2.py y Ejemplo3.py). Pues en lugar de definir funcion() en cada uno de ellos, podemos definirla en un script de python aparte (lo que será nuestra librería) e importar ese script al resto.

Así pues, tendremos en la librería (la cual llamamos libreria.py, por ejemplo) lo siguiente:

```

def funcion():
    (...) # implementacion
    return

```

Mientras que al principio de cada uno de los otros script importaremos la librería. Por ejemplo, en Ejemplo1.py:

```
from libreria import funcion

# Podemos llamar a funcion como si la hubiéramos definido aquí
funcion()
```

Hay varias formas de importar módulos, dependiendo además de si queremos importarlos por completo o solo algunas partes.

Si queremos importar el módulo completo podemos hacerlo de dos formas:

```
import libreria

# Aquí tendremos que tratar las funciones del módulo como métodos de
la clase libreria
libreria.funcion()
```

O bien:

```
# El asterisco representa que importamos todo lo que haya en la
librería
from libreria import *

# Llamamos a la función como si la hubiéramos definido aquí
funcion()
```

Pero si solo queremos importar una función podemos hacer lo siguiente

```
from libreria import funcion

funcion()
```

También podemos cambiarle el nombre a dicha función, muy útil si en varias librerías hay funciones con el mismo nombre pero diferente implementación.

```
from libreria import funcion as nuevo_nombre

nuevo_nombre() # Ejecutará funcion(), pero se llama nuevo_nombre
```

Siguiendo con esta lógica podemos importar varias funciones concretas, ya sea en una línea o varias:

```
from libreria import funcion1, funcion2
from libreria import funcion3
```

importar el módulo como una clase	<code>import module_name</code>
importar todas las funciones y clases	<code>from module_name import *</code>
importar solo una función o clase	<code>from module_name import x</code>
importar solo una función o clase y cambiarle el nombre	<code>from module_name import x as new_name</code>
importar funciones o clases concretas	<code>from module_name import x, y, z</code> <code>from module_name import a</code> <code>from module_name import b</code> <code>from module_name import c</code>

Ejemplo práctico con el módulo `numpy`:

```
import numpy
vector = numpy.array([...])
```

```
import numpy as np
vector = np.array([...])
```

```
from numpy import array
vector = array([...])
```

Por otro lado, un **paquete** o **package** es una carpeta en la cual guardamos diversos módulos. Esto se hace por organización dentro de una aplicación.

Si el script de nuestra aplicación se encuentra en su directorio (al cual nos referiremos por su path relativo), un paquete sería una carpeta en el mismo directorio en el cual se guardan los módulos. Si dicha carpeta (`./pck/`) contiene dos módulos (`mod1.py` y `mod2.py`) podemos acceder a ellos de la siguiente forma:

```
import pck.mod1
from pck.mod2 import funcion
```

Lo trataremos de la misma forma que si fuese un módulo. Pudiendo hacer lo siguiente:

```
import package_name
from package_name import module1
from package_name.module1 import x
```

Las librerías o módulos que podemos usar en Python son muchas y además existen multitud de módulos que podemos instalar con facilidad por medio de **pip** o **conda**.

Para instalar un módulo en nuestra shell de python, podemos abrir una terminal del sistema operativo (o una terminal de Anaconda) para hacer lo siguiente:

```
pip install module_name
```

Para una versión concreta:

```
pip install module_name==version
```

Si queremos ver los módulos instalados:

```
pip list
```

Y si tenemos dudas con los comandos:

```
pip --help
pip -h
pip command_name --help
```

finalmente para desinstalar un paquete:

```
pip uninstall module_name
```

Por otro lado, para Anaconda, podemos ver los comandos aquí:

https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf

Si queremos instalar las dependencias e un repositorio de Github, podemos hacerlo con pip

```
pip install git+https://bitbucket.org/<project_owner>/<project_name>
pip install git+ssh://git@bitbucket.org/<project_owner>/<project_name>.git/
```

8. Decoradores

Un decorador es una forma simple de extender las funcionalidades de una función. En sí, un decorador no es más que una función a la que se le pasa otra función como argumento, pero que tiene una sintaxis propia en Python.

```
function = decorator(function) # Sintaxis tradicional

# Sintaxis en python
@decorator
def function(...):
    ...
```



```
function()
```

Tanto las funciones como los métodos de una clase son elementos considerados “llamables”, es decir, que se les puede llamar para su ejecución desde otra parte del código. Al ser llamables, ambos contienen el método `__call__()`, de esta forma, podemos definir un decorador como una función que devuelve un elemento al que se le puede llamar, esto es, otra función.

La estructura común de un decorador es la siguiente, se define el decorador, al cual se le pasa el nombre de una función, dentro del mismo se define otra función, normalmente llamada “wrapper”, de argumentos genéricos, en el cual se hace la ampliación de funcionalidad de la función que se ha decorado y finalmente se devuelve :

```
def decorador(func):
    def wrap_func(*args, **kwargs):
        (...)
        return func(*args, **kwargs)
    return wrap_func

@decorador
def funcion_de_ejemplo(...):
    (...)

funcion_de_ejemplo()
```

Así pues, hemos decorado `funcion_de_ejemplo`, de forma que el namespace de esta función se pasará como un argumento al decorador, que a su vez pasará los argumentos de dicha función a la función de wrapper, `wrap_func`, que trabajará con ellos si es necesario, devolviendo la función `funcion_de_ejemplo` con sus argumentos y, finalmente, el decorador devolverá `wrap_func`.

En realidad, los decoradores no son, si no, llamadas a una función que añade una funcionalidad a otra sin modificar la misma. De una forma más tradicional podríamos llamar al decorador de la siguiente forma:

```
nombre = decorador(funcion_de_ejemplo )
nombre()
```

Y esto ejecutaría el código de la misma forma.

Podemos, además, encadenar varios decoradores. Los cuales se ejecutarán de arriba a abajo:

```
def decorador1(func):
    def wrapper(*args, **kwargs):
```

```

        print("Decorador 1")
        return func(*args, **kwargs)
    return wrapper

def decorador2(func):
    def wrapper(*args, **kwargs):
        print("Decorador 2")
        return func(*args, **kwargs)
    return wrapper

@decorador1
@decorador2
def funcion(x):
    print(f"{x} ** 2 = {x**2}")

funcion(5)

```

Haciendo esto, obtendremos por pantalla lo siguiente:

```

Decorador 1
Decorador 2
5 ** 2 = 25

```

Por último, si queremos decorar un decorador, tendremos realmente que decorar la función wrapper del decorador de la siguiente forma:

```

def decorador1(func):
    def wrapper(*args, **kwargs):
        print("Decorador 1")
        return func(*args, **kwargs)
    return wrapper

def decorador2(func):
    @decorador1
    def wrapper(*args, **kwargs):
        print("Decorador 2")
        return func(*args, **kwargs)
    return wrapper

```

9. Tratamiento de excepciones y el try/except

En Python tenemos muchas excepciones que van desde simples advertencias (o warnings) a errores fatales. Si queremos manejar dichas excepciones para evitar la interrupción de la ejecución del programa, o para poder controlar mejor nuestro código, podemos hacer uso de la sentencia **try/except**:

```
try:
    (...) # intentamos ejecutar este bloque de código
except:
    (...) # si el try no funciona, hacemos esto
```

Pero qué pasa si queremos que una parte del código se ejecute sí o sí. Para ellos podemos hacer uso de **finally**:

```
try:
    (...) # intentamos ejecutar este bloque de código
except:
    (...) # si el try no funciona, hacemos esto
finally:
    (...) # esta parte siempre se ejecuta
```

Si en el bloque del try no ha saltado ninguna excepción podemos hacer uso del **else**:

```
try:
    (...) # intentamos ejecutar este bloque de código
except:
    (...) # si el try no funciona, hacemos esto
else:
    (...) # si no ha habido excepciones en el try, hacemos esto
finally:
    (...) # esta parte siempre se ejecuta
```

Para poder visualizar la excepción concreta que ha saltado, podemos modificar la parte del except de esta forma:

```
except Exception as e:
    (...)
    print(e)
    (...)
```

A su vez, un try/except puede hacer uso de varios except.

Finalmente, para obligar a hacer que salte una excepción, podemos hacer uso de la palabra reservada **raise**

```
raise exception_name
```

10. Los módulos NumPy, Matplotlib y Pandas

a. Matplotlib

b. NumPy

Numpy es un módulo de Python que trata sobre todos con datos similares a las listas, pero de forma mucho más rápida y eficiente, debido a que los datos de numpy se acumulan de forma continua en memoria.

Para importar el módulo, como siempre:

```
pip install numpy
```

Para importar el módulo, es común llamarlo np:

```
import numpy
import numpy as np
```

Numpy trabaja con un tipo de dato propio llamado ndarray, que se crea con el método `array()` y es similar a las listas. Este método también sirve para convertir tipos de datos como listas, tuplas, o cualquier otro que sea similar a ndarray.

```
import numpy as np
```

```
arr = np.array([2, 4, 5,...])
```

Debemos ahora hablar de las dimensiones de los arrays. Un array puede tener de 0 a N dimensiones. La forma más común de referirse a ellos según las dimensiones es:

Dim.	Nombre	Forma
0	escalar (número)	<code>np.array(42)</code>
1	array o vector	<code>np.array([1, 2, 3])</code>
2	matriz (o matriz bidimensional)	<code>np.array([[1, 2, 3], [4, 5, 6]])</code>
3	matriz tridimensional	<code>np.array([[[...], [...], ...], [...], [...], ...], ...])</code>
4 o más	tensor o matriz de N dimensiones	

Podemos ver que realmente se pueden traducir los arrays multidimensionales como arrays de arrays de arrays hasta llegar a números. Así por ejemplo, una matriz bidimensional sería un array de array de números.

Además de convertir listas, podemos crear nuestros propios vectores:

```

arr_zeros = np.zeros(10) # array 1D de 10 ceros
mat_zeros = np.zeros((10,10)) # matriz de 10x10 de ceros

arr = np.array(range(10, 50)) # vector que contiene números del 10
al 49

arr_rand = np.random.random(10) # array de 10 números aleatorios
mat_rand = np.random.random((3, 3, 3)) # matriz tridimensional de
num aleatorios

```

También podemos conocer las dimensiones del array y cambiarlas:

```

arr = np.array(...)
print(arr.shape) # muestra las dimensiones

new_shape = (3, 3) # int o tupla de ints
np.reshape(arr, new_shape) # Cambio de dimensiones usando una
función
arr = arr.resize(new_size) # cambio de dimensiones usando un método
de la clase array

```

Podemos encontrar un compendio de los comandos más utilizados aquí:

https://assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf

c. Pandas

Para realizar las operaciones matemáticas propias del Machine Learning existen multitud de módulos y librerías más óptimos que el código básico del lenguaje, este es el caso para Numpy y Pandas.

Pandas es una herramienta de análisis de datos

11. Tratamiento de imágenes y PDF

a. Módulos os, sys y shutil

Aunque ya hemos visto cómo abrir, leer y editar archivos, es, a su vez, importante conocer la forma de poder comunicarse con el sistema operativo desde un script.

El módulo **os** permite utilizar funcionalidades propias del sistema operativo, mientras que su parte **os.path** ayuda de forma sencilla a trabajar con directorios.

El módulo **sys** permite trabajar con el intérprete de Python y con sus variables desde dentro de un script.

El módulo **shutil** provee funcionalidades de alto nivel para el trabajo con archivos y colecciones de archivos.

Se puede visitar la documentación de estos tres módulos aquí:

Módulo os	Módulo sys	Módulo shutil
---------------------------	----------------------------	-------------------------------

Cuando trabajamos con direcciones de archivos y carpetas, esto es, **paths**, debemos tener cuidado con su formato, sobre todo si trabajamos en Windows, pues el caracter especial de separación en un path es la **backslash** “\”, la cual tiene otras utilidades dentro de una string.

Pudiera darse el caso, por ejemplo, de que copiemos directamente un path desde el explorador de Windows y lo copiemos en una variable string:

```
path = 'C:\path\to\directory'
```

Entonces el intérprete de python nos diría que hemos cometido un error. Para solventar dicho error hay dos formas de hacerlo. Si el path no acaba en “\”, podemos transformar la cadena de caracteres en una string raw, que interpreta todos literalmente, poniendo una “r” antes de abrir comillas. Por otro lado, tanto si el path termina en “\” como si no, siempre podemos sustituir las backslash simples por backslash doble, que Python interpretará entonces como el caracter “\”.

```
path = r'C:\path\to\directory'
path = 'C:\\path\\to\\directory'
```

Veamos una serie de ejemplos simples para trabajar con estos módulos.

Si queremos saber desde qué directorio estamos trabajando y cambiar dicho, podemos hacer:

```
os.getcwd()      # para saber el directorio actual
os.chdir(path)   # para cambiar el directorio a path
```

Pero cómo asegurarnos de que un directorio o un archivo existen, pues para ello el módulo **os.path** nos proporciona las funciones:

```
os.path.exists(path) # Devuelve True si existe el archivo o
directorio
os.path.isdir(path_to_dir) # Devuelve True si existe y es un
directorio
os.path.isfile(path_to_file) # Devuelve True si existe y es un
archivo
```

También podemos crear directorios:

```
os.mkdir(path)
```

Y ver qué tienen en su interior:

```
os.listdir(path)
```

Por otro lado, `shutil` nos proporciona multitud de herramientas de manejo de archivos y directorios. Con él podemos, por ejemplo, borrar un directorio y todos sus contenidos, moverlos o copiarlos:

```
shutil.rmtree(path) # borra todos los contenidos de una carpeta
shutil.copytree(source_path, dest_path) # copia una carpeta
shutil.move(source_path, dest_path) # mueve una carpeta
```

También podemos comprimir una carpeta y sus contenidos:

```
shutil.make_archive(file_to_create, extension, root_dir, base_dir)
```

Donde:

- `file_to_create`: path al archivo a crear, sin la extensión
- `extension`: extensión del archivo
- `root_dir`: directorio base del archivo, por defecto el actual
- `base_dir`: desde donde se empieza a comprimir, por defecto el directorio actual. Si el comando trabaja desde `root_dir` (y hace de este el directorio base del archivo) `base_dir` es la carpeta dentro de `root_dir` que comprimiremos.

b. PIL e imágenes en Python

PIL es el módulo que instalamos con el paquete Pillow:

```
pip install Pillow
```

Con este módulo podemos abrir, modificar y guardar imágenes. Para abrir y cerrar el archivo lo tratamos como cualquier otro archivo:

```
from PIL import Image

img = Image.open(filename)
(...)
img.close()

with Image.open(filename) as img:
    (...)
```

NOTA: en una imagen de PIL las coordenadas tienen su origen en la esquina superior izda.

Algunas de las funciones más comunes de la librería pueden ser:

Uso del método	Método o atributo	Explicación
Procesado de imágenes	<code>alpha_composite(im1, im2)</code>	hace un compositado alfa de img2 sobre img1, devolviendo una imagen
	<code>blend(im1, im2, alpha)</code>	
	<code>composite(image1, image2, mask)</code>	
	<code>merge(mode, bands)</code>	
Creación	<code>new(mode, size, color=0)</code>	crea una nueva imagen
	<code>fromarray(obj, mode=None)</code>	conversión a través de un array de numpy
Efectos sobre la imagen	<code>effect_mandelbrot(size, extent, quality)</code>	
	<code>effect_noise(size, sigma)</code>	
	<code>linear_gradient(mode)</code>	
	<code>radial_gradient(mode)</code>	
Métodos de la clase Image (PIL.Image, Image)	<code>Image.alpha_composite(im, dest=(0, 0), source=(0, 0))</code>	Compone una imagen (im) sobre nuestra imagen
	<code>Image.apply_transparency()</code>	Aplica transparencia a la paleta de la imagen
	<code>Image.copy()</code>	
	<code>Image.crop(box=None)</code>	corta la imagen, box es una tupla de coordenadas de 4 elementos: (izda., arriba, dcha., abajo)
	<code>Image.paste(im, box=None, mask=None)</code>	Pega una imagen sobre nuestra imagen. box puede ser una tupla (izda., arriba) o (izda., arriba, dcha., abajo) y es donde se pega im
	<code>Image.resize(size, resample=None, box=None, reducing_gap=None)</code>	redimensiona la imagen, donde size es una tupla (width, height)
	<code>Image.rotate(angle, resample=Resampling.NEAREST, expand=0, center=None, translate=None, fillcolor=None)</code>	rota la imagen en sentido antihorario, con un ángulo dado en grados sexagesimales
	<code>Image.save(fp, format=None, **params)</code>	guarda la imagen, donde fp es el nombre del archivo y format la extensión ('PNG', 'JPEG',...)
	<code>Image.show(title=None)</code>	muestra la imagen
	<code>Image.transpose(method)</code>	gira la imagen 90°

	<code>Image.close()</code>	cierra la imagen
Atributos de la clase <code>Image</code>	<code>Image.filename: str</code>	nombre del archivo
	<code>Image.format: optional str</code>	formato de la imagen
	<code>Image.width: int</code>	ancho en píxeles
	<code>Image.height: int</code>	alto en píxeles
	<code>Image.palette: optional PIL.ImagePalette.ImagePalette</code>	
	<code>Image.info: dict</code>	
	<code>Image.is_animated: bool</code>	
	<code>Image.n_frames: int</code>	

Además de `Image`, PIL contiene otros módulos, como `ImageDraw`, que sirve para dibujar en una imagen.

La documentación del módulo PIL la podemos encontrar aquí:

<https://pillow.readthedocs.io/en/stable/reference/index.html>

c. Trabajando con PDF

Un archivo `.pdf`, por sus siglas “Portable Document Format”, es un archivo de texto e imagen creado por Adobe, aunque ahora es administrado por la ISO (“International Organization for Standardization”).

Para trabajar con pdfs en Python existe una plétora de módulos, aunque los más comunes son `PyPDF2` y `PyPDF3`, que derivan de otro módulo más antiguo, `PyPDF`. En general, `PyPDF2` es la más utilizada en la documentación disponible, siendo `PyPDF3` la más moderna. Además, tenemos el módulo `PyFPDF`, especializado en la generación de pdfs a través de Python.

```
pip install pypdf2
pip install pypdf3
pip install fpdf
```

Usando `PyPDF2` podemos, por ejemplo, leer el contenido de un pdf:

```
import PyPDF2

pdf = open(filename, 'rb')
pdf_reader = PyPDF2.PdfFileReader(pdf)
```

```

for i in range(pdf_reader.numPages):
    page = pdfReader.getPage(i)
    print(page.extractText())

pdf.close()

```

Un objeto PyPDF2.PdfFileReader puede darnos información el archivo:

```

print(pdf_reader.numPages)
print(pdf_reader.documentInfo)

```

Para escribir un nuevo archivo, podemos:

```

pdf_writer = PyPDF2.PdfFileWriter()
(...)
pdf_writer.addPage(pageObj)
(...)
new_pdf = open(newfile, 'wb')
pdf_writer.write(new_pdf)
new_pdf.close()

```

Aunque también podemos hacer uso de las funcionalidades de FPDF:

```

from fpdf import FPDF

pag_orientation = 'P' # P: portrait(vertical)      L:
landscape(horizontal)
pag_format = 'A4' # NOTA: fpdf da las dimensiones en mm

pdf = FPDF(orientation = pag_orientation, unit = 'mm', format =
pag_format)

pdf.add_page()
pdf.set_font("Arial", size=12) # font and textsize
pdf.cell(200, 10, txt="your text", ln=1, align="L")
pdf.output(path_pdf, "F")

```

Podemos ver todas las funcionalidades de FPDF aquí:

<https://pyfpdf.readthedocs.io/en/latest/index.html>

12. Bases de datos MongoDB

Una base de datos es una recopilación organizada de información o de datos estructurados, controlada por un sistema de gestión de bases de datos.

MongoDB es una base de datos que, a diferencia de otras más comunes, como SQL, organiza sus datos en formato de archivos json, organizados en colecciones de documentos. Una base de datos de MongoDB puede tener varias colecciones.

Los archivos json (JavaScript Object Notation) son fácilmente interpretables gracias a su módulo homónimo. Así pues, se proporciona una traducción directa de los tipo de dato:

json	python	json	python
object	dict	number(real)	float
array	list	true	True
string	str	false	False
number(int)	int	null	None

Aunque dicho módulo es extenso, sus dos principales funciones son las de crear y leer un json, utilizando **json.dumps()** y **json.loads()** respectivamente.

```
import json

dict_example = {'name': 'José', 'age': 45}

# pasamos el diccionario a json
json_example = json.dumps(dict_example)

# pasamos el json a diccionario
dict_decoded = json.loads(json_example )
```

Por supuesto, ambas funciones tienen muchos más argumentos, tal y como se ve en su definición:

```
json.dumps(obj, *, skipkeys=False, ensure_ascii=True,
check_circular=True, allow_nan=True, cls=None, indent=None,
separators=None, default=None, sort_keys=False, **kw)

json.loads(s, *, cls=None, object_hook=None, parse_float=None,
parse_int=None, parse_constant=None, object_pairs_hook=None, **kw)
```

También existen las funciones `json.load()` y `json.dump()`, pero estas leen un json para convertirlo en un “file-like object” o viceversa, en lugar de a un tipo de dato str, bytes o bytearray.

Lo primero, como siempre, es instalar las librerías. En ambos casos puede utilizarse pip:

```
pip install pymongo
pip install json
```

Una vez hecho esto, podemos empezar a trabajar con MongoDB.

Lo primero, es crear en nuestro código un cliente que nos permita acceder a la base de datos. Podemos hacer que este cliente se conecte al host y el port por defecto, podemos especificarlo o podemos conectarnos a través de una url.

```
from pymongo import MongoClient

client = MongoClient()
client = MongoClient('localhost', 27017)
client = MongoClient('mongodb://localhost:27017/')
```

Ahora podemos acceder a nuestra base de datos, lo cual podemos hacer de las siguientes formas:

```
mydb = client.my_database
mydb = client['my-database']
```

Dentro de nuestra base de datos, podemos acceder a sus colecciones de forma similar:

```
mycol= mydb.my_collection
mycol= mydb['my-collection']
```

Cuidado en ambas notaciones con la diferencia entre guión (“-”) y la barra baja (“_”).

Ahora ya estamos listos para realizar operaciones con los json de nuestra base de datos.

Método	Explicación
mycol.find({"hello": "world"})	Busca todos los documentos que cumplan con el filtro especificado. Si ese filtro se deja vacío ({}), buscará todos los elementos.
mycol.find_one(search_json = my_search_json)	Busca un json en la base de datos
mycol.find_one_and_update(search_json = my_search_json, new_json = my_new_json)	Busca un json y lo cambia por uno nuevo
mycol.insert_one(new_json = my_new_json)	Inserta un nuevo json en la colección
mycol.insert_many(array_jsons = my_array_jsons)	Inserta todos los elemento de una lista de json en la colección
mycol.update_one(search_json = my_search_json, changes_json = my_changes_json)	Cambia un json en la colección por uno nuevo:
mycol.remove({filter_dict})	Elimina todos los elementos que concuerden con el filtro. CUIDADO: los datos eliminados no se pueden

	recuperar.
--	------------

Un ejemplo de `update_one()`:

```
client = MongoClient()
mydb = client['my-database']
mycol = mydb['my-collection']

search_json = {"key1": value1, "key2": value2}
update_q = {"$set": {"key_to_update": value_to_update}}
mycol.update_one(search_json, update_q)
```

13. Bases de la Inteligencia Artificial

Se define grosso modo como **Inteligencia Artificial** (IA en español o AI en inglés, de “Artificial Intelligence”) a un subcampo de las ciencias de la computación que pretende emular la inteligencia de los sistemas biológicos y su toma de decisiones. Es un amplio campo de definición laxa y en muchos casos ligado al Minado de Datos para obtener modelos y predicciones.

Dentro de la inteligencia artificial se encuentra el campo del Aprendizaje Automático o Aprendizaje de Máquinas (más conocido por su terminología inglesa **Machile Learning**), que es la forma de hacer que un sistema de inteligencia artificial aprenda a hacer predicciones más o menos exactas.

Una de las formas más comunes de crear una IA es a través de los diferentes tipos de **máquinas neurales**, o redes neuronales o neurales (NN por sus siglas en inglés, “Neural Networks”). Una máquina neural son modelos computacionales que se basan en un modelo simplificado de las células nerviosas. En esta se tiene una serie de **neuronas** agrupadas en **capas** y unidas por **conexiones**. En general, las conexiones poseen una propiedad llamada **peso**, que indica lo fuerte que es esa conexión o, dicho de otro modo, lo importante que es esa conexión para la neurona en cuestión.

Una neurona obtendrá los pesos de todas sus conexiones de input y los sumará, dicha suma se llama **activación** y normalmente se le añade un término más denominado “**bias**”. Esta suma se pasará por una función, normalmente no lineal, denominada **función de activación**, que producirá el output que pasará a las neuronas de la siguiente capa.

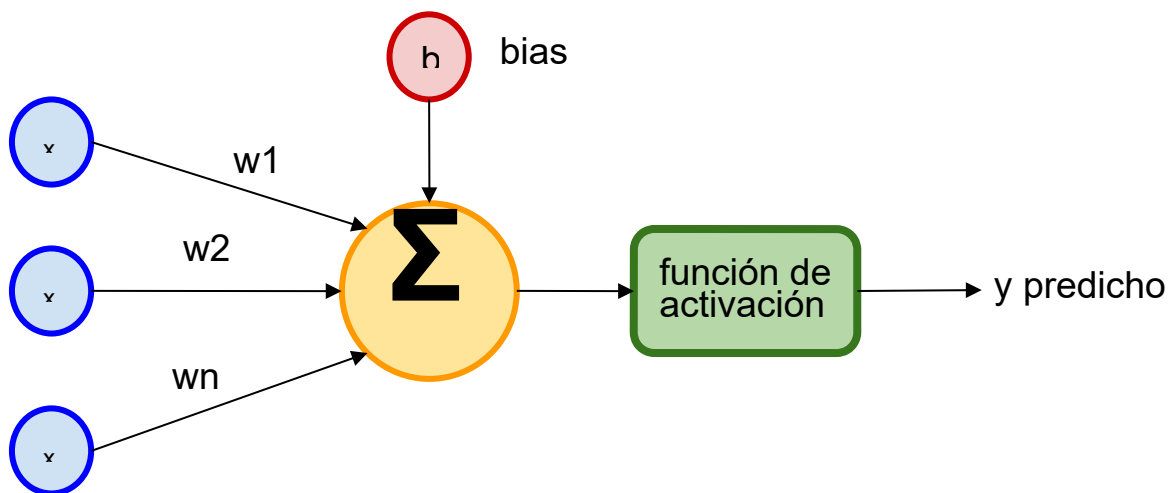
En una red neural la primera capa, o input, son los datos que queremos procesar, mientras que la última nos da los resultados de la predicción, por ejemplo, si en la imagen de input hay caras humanas.

Una vez creado nuestro sistema de inteligencia artificial, es necesario entrenarlo para que sea capaz de hacer algo. Para esto existen varias estrategias, aunque las dos más populares son:

- **Aprendizaje supervisado:** se le da a la IA una serie de ejemplos resueltos y esta ajustará sus parámetros (pesos, bias, etc.) en función de la diferencia entre la solución predicha y la real que nosotros le hemos pasado.
- **Aprendizaje no supervisado:** en este caso se le pasa una serie de inputs sin decir la solución y la IA se entrena para encontrar patrones comunes en ellos.

a. Principios de las redes neuronales

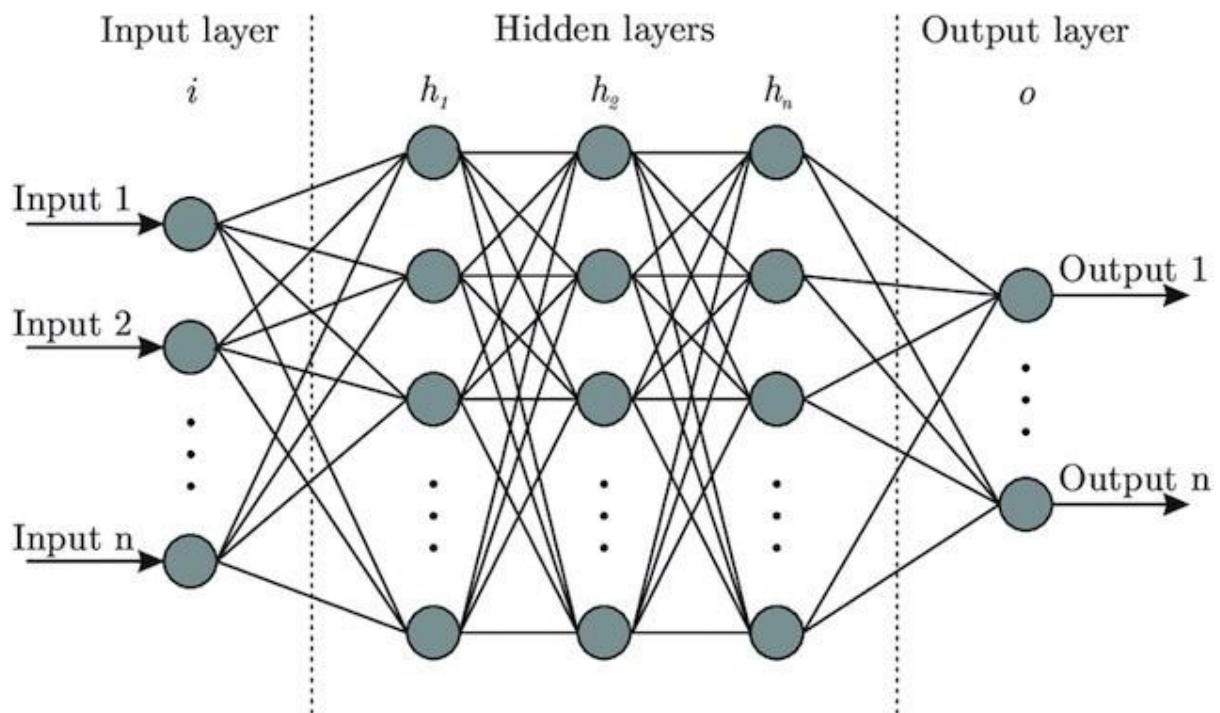
Como ya se ha mencionado, una máquina neural se compone de neuronas y de conexiones entre ellas.



En este caso los outputs de las neuronas de la capa anterior (x_i) se multiplican por los pesos correspondientes (w_i), se suman entre ellos y con la bias (b), creando la activación. Finalmente, la activación se pasa por la función de activación para generar el output de nuestra neurona.

Para una red neuronal convencional, las neuronas se agrupan en capas, estando las capas de entrada y de salida, por donde pasan los datos y salen las predicciones, respectivamente, y las capas ocultas, que son todas aquellas que se encuentran entre las dos anteriores.

Una red neuronal convencional tendrá pues esta estructura:



b. Entrenamiento e hiperparámetros

Una vez definida la estructura de nuestra red neuronal, debemos definir un algoritmo que se encargue de entrenarla. Este algoritmo estará controlado por unos parámetros que controlarán todos los aspectos del aprendizaje. A diferencia de otros parámetros, como los pesos, los hiperparámetros afectan por igual a toda la estructura de la red.

Uno de los parámetros más comunes es el ratio de aprendizaje, o **“learning-rate”**, que indica lo rápido o lento que nuestras neuronas cambian sus parámetros con cada entrenamiento. Resulta antiintuitivo que un learning-rate muy alto pueda evitar que una NN se entrene adecuadamente, pues podría pasar que el cambio tan repentino de los parámetros de la red evite que esta produzca un modelo satisfactorio. Por otro lado, una red con un learning-rate muy bajo sería incapaz de aprender adecuadamente antes de terminar el entrenamiento.

En cada etapa del entrenamiento será necesario calcular el error entre el resultado predicho y el valor actual (en un entrenamiento supervisado), para poder hacer las correcciones necesarias. Este error se conoce como pérdida o **“loss”** y viene calculado por su **“loss function”**. Existen muchas funciones que calculan la pérdida, aunque la más simple es a través del gradiente.

Una vez calculado el error, antes de la nueva etapa de entrenamiento, se calculan los nuevos pesos de las conexiones por un método conocido como **“backpropagation”**, que permite obtener todos estos pesos rápidamente.

También hay hiperparámetro que afectan directamente a los sets de entrenamiento, como, por ejemplo, la proporción entre las partes de test y train del dataset, el tamaño del batch, el número de épocas que deberá durar el entrenamiento, etc.

En general, el objetivo de un entrenamiento es el de disminuir al mínimo el error a la vez que se maximiza la capacidad de predicción de la red.

A la hora de entrenar, nosotros dividiremos nuestro dataset en varias partes: train, valid y test. La parte de train se encarga de entrenar la red y cambiar los pesos, la parte de valid de cambiar los hiperparámetros necesarios para la siguiente época de entrenamiento y la parte de test comprueba los resultados de cada época. A la hora de realizar el dataset debemos hacerlo de forma que quede una parte representativa de cada tipo de input en todas las partes, si no, el entrenamiento no podrá ser tan bueno como deseamos.

Una vez creado nuestro dataset de entrenamiento, comenzaremos con el entrenamiento en sí. Un entrenamiento supervisado consiste en pasar reiteradamente el dataset separado en conjuntos de inputs (el tamaño de este conjunto se conoce como **batch-size**), para finalmente hacer las correcciones correspondientes debido al error calculado entre la predicción y la realidad. Cada etapa del entrenamiento se denomina **época**.

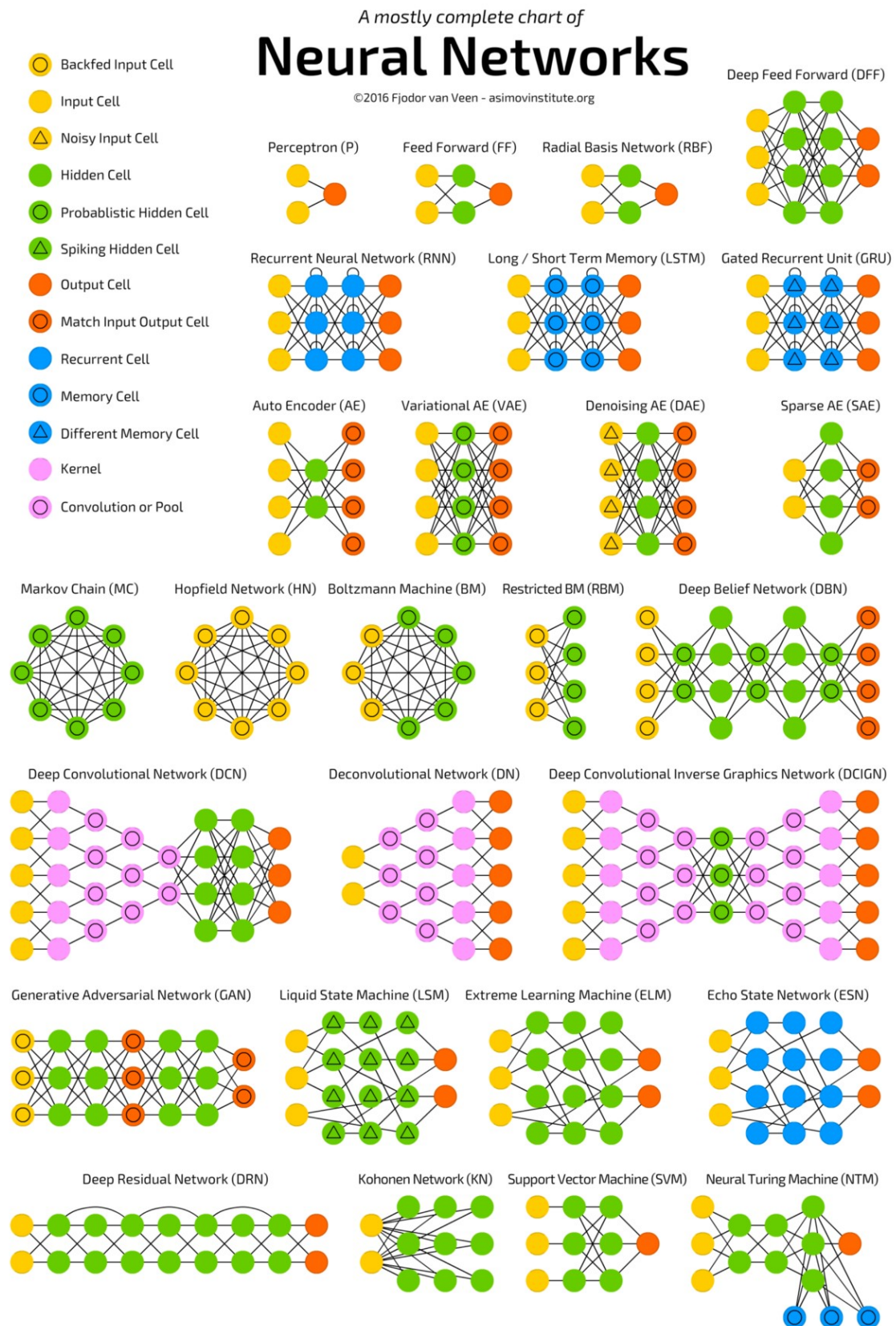
En cualquier caso, hay que tener en cuenta que una red neuronal producirá predicciones que nunca serán acertadas al 100%. Mediante el manejo de los hiperparámetros podemos obtener modelos que obtengan unas muy buenas predicciones, pero es normal toparse con dos problemas al diseñar dicho entrenamiento:

- **Undertraining:** los resultados pueden mejorarse porque el entrenamiento no ha sido lo suficientemente largo para los hiperparámetros dados.
- **Overtraining:** este puede resultar menos intuitivo. Si se deja un modelo entrenar durante demasiado tiempo, adecuara la red casi a la perfección para el dataset de entrenamiento (habrá bajado su loss al mínimo posible), sin embargo, será incapaz de realizar predicciones de cualquier cosa fuera del dataset de entrenamiento.

También podemos encontrar problemas a la hora de entrenar pero debido a la estructura de nuestra red:

- **Underfitting:** nuestra red es demasiado simple para los inputs
- **Overfitting:** nuestra red es demasiado compleja y no puede ser entrenada satisfactoriamente para la cantidad de inputs que tenemos.

c. Tipos de redes neuronales



[the mostly complete chart of neural networks](#)

d. Machine Learning in Python

e. Los módulos Numpy, Pandas y Matplotlib

f. Keras y TensorFlow

Keras y TensorFlow son dos módulos de Machine Learning para Python muy relacionados y ampliamente usados y que suelen usarse a la vez debido a su estabilidad y eficiencia para desarrollar Inteligencias Artificiales.

g. Redes Neurales Convolucionales

14. Google Colab, Yolo, Roboflow y Labelling

Yolo ("You Only Look Once") es un modelo de detección de objetos que divide las imágenes en una cuadrícula, haciendo una detección de objetos común sobre cada parte de esta.

Existen diferentes versiones del modelo Yolo, nosotros trabajaremos con YOLOv5. Para instalarlo en nuestro proyecto de colab, haremos:

```
git clone https://github.com/ultralytics/yolov5.git
```

Después deberemos ir al directorio de YOLOv5 y allí instalar el archivo de requerimientos:

```
cd yolov5  
pip install -r requirements.txt
```

Una vez hecho esto podemos instalar nuestro dataset de imágenes y label: