

Sesión 1

✓ Introducción

✓ Objetivos del Módulo

Los objetivos de este modulo son **introducir Python y sus aplicaciones en Machine Learning** y abordar los **conceptos claves de Python para afrontar las diferentes prácticas que tendréis que realizar a lo largo del Máster**. Para abordar de forma correcta el taller, os dejo una serie de consejos que os pueden ser útiles:

- No tengáis prisa, aunque el modulo es corto y se explican muchos conceptos, la clave es ir repasando poco a poco la información a lo largo del Máster.
- Orientad el módulo como una guía o manual, aquí encontraréis prácticamente todo lo que necesitas durante el Máster, pero es difícil abordarlo todo desde el primer momento. Intentad ir incorporar poco a poco los conocimientos que vamos a ver para luego ampliar la base por tu cuenta.
- Intentad consultar la **documentación oficial**, aquí dejaré un resumen de la mayoría de los conceptos, pero es importante acudir a las guías oficiales para resolver dudas específicas.
- **Escribid código, no os limiteis a leer lo que aparece**. A lo largo del documento encontraréis cuadros en los que escribir tu código y probar todo lo visto.
- **Sed realistas con los objetivos**, programar es algo complejo, se requiere de tiempo y práctica, si no se tiene experiencia previa es necesario un tiempo amplio para adquirir un mínimo de conocimientos.
- Apoyaos en las herramientas de ayuda de código como ChatGPT o Copilot para ir ganando experiencia

Es importante volver de vez en cuando a las explicaciones dadas en el módulo. Muchas de las dudas que os surjan serán cuando os enfrentéis a los problemas que se os propongan, así que es importante repasar asiduamente todo lo visto en esta guía para afianzar conocimientos y resolver dudas.

Algunos de los recursos más importantes sobre Python son:

- [Documentación Oficial de Python](#)
- [Guía de Python](#)
- [Tutorial de Python en Español](#)



Colaboratory

Colaboratory es un entorno basado en [Jupyter Notebook](#) que permite ejecutar código en Python sin ningún tipo de requerimiento ni instalaciones previas.

Con Colaboratory puedes escribir y ejecutar código, guardar y visualizar las analíticas y acceder a recursos computacionales desde el navegador.

El documento que estás leyendo es un *Jupyter notebook* alojado en Colaboratory. No es una página estática, es un entorno dinámico que permite escribir y ejecutar código en *Python 3*.

Por ejemplo aquí tenemos una **celda de código** con un pequeño script en *Python* que realiza una operación aritmética y guarda el resultado en una variable, imprimiendo el resultado en pantalla, lo único que hay que hacer es pulsar entre los dos paréntesis []

```
# My first program
seconds_in_a_minute = 60 * 1 * 1
print(seconds_in_a_minute)
```

 60

También encontrarás celdas vacías donde podrás probar tu código para ver los resultados, es importante usarlas ya que la mejor forma de aprender a programar es programando. Aparecerán con un comentario con la palabra reservada `TODO`.

✓ Ejercicio

Escribe un script que enseñe los minutos que hay en un día

```
# TODO Write a program to display minutes in a day
minutes = 24 * 60
print(minutes)
```

➞ 1440

✓ Solución

Haz click para ver el resultado

```
day = 1
hours = 24
minutes = 60
minutes_in_day = day * hours * minutes
print(minutes_in_day)
```

```
#-----
print(24 * 60 * 1)
```

➞ 1440
1440

✓ Python

Python es un lenguaje de propósito general, interpretado, interactivo, orientado objetos y de alto nivel. Fue creado por Guido van Rossum entre 1985 y 1990. Está amparado dentro de la GNU General Public License (GPL). Y fue llamado así por los Monty Python (no por la serpiente pitón).

Es importante explicar que un lenguaje de programación es un lenguaje formal que proporciona una serie de instrucciones que permiten a un programador escribir secuencias de órdenes y algoritmos a modo de controlar el comportamiento lógico de una computadora.

Python se ha hecho muy popular debido a su sintaxis clara, la facilidad como lenguaje para crear *scripts* y la cantidad de librerías populares que permiten solventar casi cualquier problema con el que el programador se enfrente. Por ello es el lenguaje de preferencia para el análisis de datos (pese a que la mayoría de librerías se creen en *C* para estar muy optimizadas y luego se abstraigan con *Python*)

Otra de las ventajas de *Python* es que se encuentra presente en Windows, macOS y la mayoría de distribuciones de Linux. Para poder utilizar este lenguaje es necesario primero hacer una instalación previa, pero como hemos comentado previamente, al utilizar un *Jupyter Notebook* podremos ejecutar aquí código sin necesidad de esa configuración.

Python 3 vs Python 2

Python tiene un problema de versionado que **ya ha terminado**. A finales de 2019 se acabó el soporte de *Python 2* versión que todavía está presente en muchas Librerías y Frameworks. Esto ha supuesto un problema durante mucho tiempo.

Python 3 nació con el objetivo de mejorar ciertas características de *Python 2*, con el problema de que perdía retrocompatibilidad, así hizo que la comunidad se dividiese entre estos dos lenguajes de programación, con una base en común pero con diferencias importantes.

En la actualidad es importante centrarse en *Python 3* ya que su antecesor ya no tiene soporte, pero por desgracia hay todavía mucho código que se mantiene en *Python 2* por lo que es importante tener conocimiento de su existencia y saber algunas de sus diferencias.

✓ Mi primer programa

✓ Hola Mundo

Python tiene muchas similitudes a otros lenguajes como Perl, C o Java. Pese a ello hay unas cuantas diferencias esenciales que le hacen únicos. Al ser un lenguaje *Interpretado* no hará falta compilarlo antes de ejecutarlo, así evitamos cualquier declaración previa y podemos crear programas de una sola línea de código.

```
# First program
print("Hello World")
```


 Hello World

Aquí podemos ver nuestro primer programa en Python. Tan sencillo como escribir una línea de código para que se imprima en pantalla el mensaje que queramos. Al ser un lenguaje interpretado, permite realizar la traducción del lenguaje al código de máquina a medida que sea necesario, típicamente instrucción por instrucción, por lo que permite ejecutarse casi inmediatamente. Ahora te toca a ti, modifica el programa para que diga "Hola, soy [tu nombre]"

✓ Ejercicio

Escribe un script que asigne a una variable tu nombre para mostrarla en pantalla con una frase


```
# TODO First program that holds a variable with your name and displays it with a sentence
nombre_mio = "Martin Arroyo"
print (nombre_mio)
```

 Martin Arroyo

✓ Solución

Haz click para ver el resultado

```
name = "Lucas Fernandez"
print("Hello, my name is " + name)
print(f"Hello, my name is {name}")
print("Hello, my name is {}".format(name))
```

 Hello, my name is Lucas Fernandez
Hello, my name is Lucas Fernandez
Hello, my name is Lucas Fernandez

✓ Análisis léxico

✓ Identificadores

Los identificadores son nombres usados para identificar una variable, función, clase, módulo u otro objeto. Los identificadores pueden empezar con letras (A-Za-z), números (0-9) o una barra baja (_).

Python es un lenguaje que distingue entre mayúsculas y minúsculas, por lo que **Variable** y **variable** son dos identificadores diferentes en *Python*.

Hay una serie de convenciones para los identificadores en *Python*:

- Nombres de Clases empiezan en mayúsculas. Los demás identificadores comienzan con minúsculas
- Python usa snake case name_surname
- Empezar un identificador con una barra baja (_variable) significa que el identificador es privado
- Empezar con dos barras bajas (__variable) indica un identificador privado con fuerte referencia
- Si el identificador termina encima con otras dos barras bajas (__variable__) significa que es un [magic method](#)

Machine learning

Vamos a ver algunos conceptos de ML como ejemplos de identificadores:

- [Features](#) -> La entrada o dato que vamos a introducir en nuestro modelo para generar una respuesta.
- [Label](#) -> La predicción del modelo que vamos a entrenar.
- [Example](#) -> El conjunto de feature y label, puede estar etiquetado si tiene label o no.

```
# Identifiers
```

```
city_feature = "San Francisco"
median_income_label = 1000000
wrongFormedLabel = 10 # wrong
_private_example = ["San Francisco", 10000]
print(_private_example)
```

```
if __name__ == "__main__":
    print("This is a magic method")
```

```
→ ['San Francisco', 10000]
   This is a magic method
```

✓ Palabras reservadas

Hay una serie de palabras reservadas por el interprete que no deben ser asignadas como variables ya que crearían conflictos. Todas estas palabras reservadas son en minúsculas y tienen una función específica dentro del lenguaje.

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

✓ Lineas e indentación

Python no utiliza corchetes ({} para indicar bloques de código dentro de las clases y funciones o control de flujo. En este lenguaje se utiliza la indentación para indicar bloques, y se aplica sin ningún tipo de excepción.

El número de espacios puede variar dependiendo del intérprete (se suele elegir 2 o 4) pero una vez elegido debe respetarse, por ello este ejemplo no produce ningún error:

```
if True:
    print("Success")
else:
    print("Failed")
```

```
→ Success
```

Mientras que si se cambian los espacios se producirá un error, como indicará el intérprete en este trozo de código:

```
if True:
    print("Sucess")
else:
    print("Failed")
```

```
→ Sucess
```

No es necesario que entiendas la lógica del programa en este momento, solo la importancia de la indentación y como puede variar. Hay excepciones que se pueden aplicar.

Por mejorar la legibilidad del código se permite utilizar el carácter de continuación de línea (\) para encadenar una misma línea:

```
total = 1 + \
    2 + \
    3
print(total)
```

↩ 6

Además, hay algunos elementos como listas o diccionarios que pueden partirse sin necesidad de este caracter especial:

```
days_week = ['Monday', 'Tuesday',
              'Wednesday', 'Thursday', 'Friday']
print(days_week)
```

↩ ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']

✓ Comentarios

El símbolo de la almohadilla (#) representa un comentario en Python, toda cadena que precede a este símbolo en una misma linea es considerado un comentario en Python y es tratado así por el interprete.

```
# This is a comment

test = "string" # You can place a comment after a line of code

# You can also concatenate multiple lines
# of comments like this

print (test)
```

↩ string

Además de esto, *Python* tiene reservado las triples comillas (") para interpretarlo como un comentario de un número indefinido de lineas

```
'''
This is a multiline comment
You can write multiple statements between the quotes
```

```
And leave spaces and special characters
'''
```

✓ Variables y Tipos

Una variable es un espacio de memoria reservado dentro de una computadora para almacenar valores. Significa que cada vez que creas una variable estás reservando un poco de espacio en una zona específica de la memoria.

Dependiendo del tipo de dato, el interprete reserva determinadas zonas de memoria, dependiendo la longitud del tipo que se declara.

Las variables en *Python* no necesitan ser declaradas para reservar memoria. Esta declaración ocurre automáticamente cuando se asigna un valor a una variable.

El simbolo *igual* (=) se utiliza para asignar valores a variables, siendo el item de la izquierda el nombre de la variable y el item de la derecha el valor que se va a asignar.

Machine Learning

Podemos indicar la [función de perdida](#) como un string dentro de nuestro modelo, veremos en un futuro como declararlo.

```
# Variable assignment
l0_weight = 100
loss = 'mean_squared_error'
tensor_data = [10, 20, 30]

# Output variables
print(l0_weight)
print(loss)
print(tensor_data)
```

```

100
mean_squared_error
[10, 20, 30]

```

▼ Ejercicio

Escribe una variable pública y otra privada según el convenio seguido en Python. Imprime ambos resultados por pantalla.

```

# TODO Write one public variable and one private variable and display both
prueba1 = "Hola"
_prueba = "Bien"
print (prueba1)
print (_prueba)

```

```

Hola
Bien

```

▼ Solución

Haz click para ver el resultado

```

public_variable = 40
_private_variable = "secret"
print(public_variable)
print(_private_variable)

```

```

40
secret

```

▼ Números

El primer tipo de dato en *Python* que vamos a ver son números. Los números son objetos creados cuando asignas un valor numérico a una variable.

Python soporta tres tipos de números:

- int: Enteros con signo (Python 3 aún a int y long)
- float: Números reales con coma flotante
- complex: Números complejos

Number examples

```

# Integer
int_number = 10
int_number_2 = -400
int_number_3 = 0x69
int_number_4 = -0x259

```

```

# Float
float_number = 12.40
float_number_2 = -25.3
float_number_3 = 45.2e18
float_number_4 = 70.2E12

```

```

# Complex
complex_number = 3.14j
complex_number_2 = 9.322e-36j

```

Algunas de las operaciones más importantes se pueden ver en su [documentación oficial](#)

▼ Ejercicio

Escribe algunas variables con enteros, con coma flotante y complejos. Convierte los números con coma flotante y complejos en entero e imprime por pantalla.

```
# TODO Write some variables with int, float and complex numbers and cast complex into integer
penarol = 1891
pi = 3.1416

print (penarol)
print (pi)
```

```
1891
3.1416
```

✓ Solución

Haz click para ver el resultado

```
int_number = 300
float_number = 42.30
complex_number = 9.322e-36j

float_int_number = int(float_number)
complex_int_number = int(complex_number.real)
print(float_int_number)
print(complex_int_number)
```

```
42
0
```

✓ Cadenas

Las cadenas en Python se definen como una concatenación de caracteres representados entre comillas. Dentro de *Python* se pueden usar comillas simples (') o dobles (") para representar cadenas.

```
# String examples
name_person = 'Hello World'
char = "H"
str_2 = "Hello World again"

print(char)
print(str_2)
print(name_person)
```

```
H
Hello World again
Hello World
```

Una diferencia importante con otros lenguajes de programación es que *Python* es que no existe el tipo de *Caracter*, estos son tratados como *Cadenas* de longitud 1.

Otra propiedad importante de las *Cadenas* en *Python* es que tienen una forma sencilla de acceder a cadenas más sencillas o sub-cadenas.

Substrings

```
str1 = 'Hello World'
str2 = 'Hello World again'

print(str1[0]) # Prints first char
print(str2[2:6]) # Prints substring from second to sixth character
print(str2[1:-1]) # Prints substring from second to the end

str3 = str1[-1]
print(str3)
```

```
H
llo
ello World agai
d
```

Hay una serie de identificadores reservados que añaden función a las cadenas

```
# Concatenation
str_concat = "Hola" + ", que tal"
print(str_concat)

# Repetition
str_rep = str_concat * 2
print(str_rep)

# Slice
str_one = str_concat[3]
print(str_one)

# Range Slice
str_slice = str_concat[3:]
print(str_slice)

# Equality
str_equal = "Hola" == str_concat
print(str_equal)

# Membership
str_member = "Hola" in str_concat # return True or False
print(str_member)

# Negative Membership
str_not_member = "Hola" not in str_concat # return True or False
print(str_not_member)
```

```
⇒ Hola, que tal
   Hola, que talHola, que tal
   a
   a, que tal
   False
   True
   False
```

Es posible dar formato al texto de diferentes formas, algunas incorporadas en las últimas versiones de python.

```
# First form interpolation
interpolation1 = "Hello world I'm %s and I'm %d years old" % ("Javier", 32)
print(interpolation1)

# Second form interpolation
interpolation2 = "Hello world I'm {0} and I'm {1} years old".format("Javier", 32)
print(interpolation2)

# Third form interpolation (Python 3.6)
name = "Javier"
age = 32
interpolation3 = f"Hello world I'm {name} and I'm {age} years old"
print(interpolation3)
```

```
⇒ Hello world I'm Javier and I'm 32 years old
   Hello world I'm Javier and I'm 32 years old
   Hello world I'm Javier and I'm 32 years old
```

Artículo sobre los distintos métodos de interpolación:

<https://realpython.com/python-f-strings/>

Dentro de las cadenas hay caracteres especiales para dar formato al texto;

- Caracter de escape: \
- Comillas simples y comillas dobles: \' \"
- Salto de línea: \n
- Tabulación: \t

✓ Ejercicio

En esta ocasión vamos a hacer varios ejercicios:

- Escribir varias variables que sean cadenas de caracteres
- Concatenar varias de ellas con interpolación de cadenas
- Coger la sub-cadena de una de las cadenas principales

- Ver si la sub-cadena está en la cadena principal
- Comparar si una cadena (inventada, sin relación con la cadena principal) está contenida en la cadena principal

```
# TODO Write some variables with string
golero = "Rochet"
defensa = "Araujo"
mediocampista = "Valverde"
eq_defensivo = golero + defensa
print (eq_defensivo)
```

```
# Concatenate few of them with interpolation
defensa2 = "Caceres"
latderecho = "Nandez"
latizq = "Olivera"
linea_de_4 = f"La linea de 4 se conformó con {latderecho} y {latizq} como laterales"
print (linea_de_4)
```

```
# Get a substring from one of the main strings
defensa5 = defensa2[0:2]
print (defensa5)
```

```
# Check if this stirng is in the main string
garra = "Nandez" in linea_de_4
print (garra)
```

```
# Check if other random string is in the main string
leyenda = "Araujo" in linea_de_4
print (leyenda)
```

```
# Print the followin text with the given result
```

```
# Slice this String by the given character: "."
```

```
→ RochetAraujo
La línea de 4 se conformó con Nandez y Olivera como laterales
Ca
True
False
```

▼ Solución

Haz click para ver el resultado

```
# TODO Write some variables with string
first_string = "Hello World, I'm Lucas"
second_string = "Hello again, I'm coding"

# Concatenate few of them with interpolation
mix_string = f"{first_string} || {second_string}"
other_interpolation = "First string: {}".format(first_string)
print(mix_string)
print(other_interpolation)

# Get a substring from one of the main strings
sub_first_string = first_string[1:5]
print(sub_first_string)

# Check if this stirng is in the main string
check_string = sub_first_string in first_string
print(check_string)

# Check if other random string is in the main string
check_rand_string = "fasfd asdf" in first_string
print(check_rand_string)

# Print the followin text with the given result
'''
Hola mundo, empiezo a escribir.
    He dejado dos espacios tabulados en esta frase.
    Me despido.
'''
print("Hola mundo, empiezo a escribir.\n\t\tHe dejado dos espacios tabulados en esta frase.\n\tMe despido.")
```

✓ Listas

Las listas son colecciones ordenadas de elementos que pueden no ser del mismo tipo. En python las listas se definen con valores separados por comas contenidos entre corchetes.

```
list_example_strings = ["Hello", "World", "I'm", "John"]
print(list_example_strings)
list_example_number = [1,2,3,4,5,6]
print(list_example_number)
list_example_mix = ["Hello", [2, 3], "Number", 16]
print(list_example_mix)
```

```
→ ['Hello', 'World', 'I'm', 'John']
   [1, 2, 3, 4, 5, 6]
   ['Hello', [2, 3], 'Number', 16]
```

Acceder a valores

Para acceder al valor de una lista, llamamos al nombre de la variable seguido con el índice entre corchetes que queremos obtener.

```
element1 = list_example_strings[0]
print(element1)
# You can also access range of element generating sublist
sub_list = list_example_number[0:-2]
print(sub_list)
sub_list_2 = list_example_number[1:]
print(sub_list_2)
# You can access to the value of sublist
sub_list_value = list_example_mix[1][1]
print(sub_list_value)
```

```
→ Hello
   [1, 2, 3, 4]
   [2, 3, 4, 5, 6]
   3
```

Si accedes a un índice superior al conjunto de la lista se produce un error.

```
# Error
element_fail = list_example_number [3]
print(element_fail)
```

↻ 4

Añadir un elemento

Para añadir un elemento dentro de una lista en Python podremos utilizar varias funciones, dependiendo de si queremos añadirlo al final de la cola, en una posición determinada o si queremos agregar otra lista a la anterior.

```
first_list = ["Hello", "World"]
first_list.append("Again")
print(first_list)
print("=====1=====")
second_list = ["Hello", "Again"]
second_list.insert(0, "World")
print(second_list)
print("=====2=====")
first_list_fail = first_list[:]
first_list_fail.append(second_list)
print(first_list_fail)
print("=====3=====")
first_list.extend(second_list)
print(first_list)
```

```
↻ ["Hello", "World", "Again"]
=====1=====
["World", "Hello", "Again"]
=====2=====
["Hello", "World", "Again", ["World", "Hello", "Again"]]
=====3=====
["Hello", "World", "Again", "World", "Hello", "Again"]
```

```
list_1 = [1, 2, 3]
list_2 = [4, 5, 6]
#list_1.extend(list_2)
print(list_1 + list_2)
```

```
↻ [1, 2, 3, 4, 5, 6]
```

Eliminar un elemento

Para eliminar un elemento, puedes usar la palabra reservada *del* para eliminar por posición o el método *remove()* para eliminar el objeto.

```
first_list_del = ["Hello",
                  "World",
                  "Again"]
first_list_del.remove("Hello")
print(first_list_del)
second_list_del = ["Hello", "World", "Again"]
del second_list_del[2]
print(second_list_del)
```

```
↻ ["World", "Again"]
["Hello", "World"]
```

Actualizar un elemento

Accediendo a la posición de una lista podemos actualizar sus elementos

```
# First list
updating_list = ["Hello", "World", 2017]
print(updating_list)
# Updating list
updating_list[2] = 2023
print(updating_list)
```

```
↻ ["Hello", "World", 2017]
["Hello", "World", 2023]
```

Operadores elementales

Hay ciertos operadores elementales que pueden modificar el comportamiento de las listas en Python

```
list_string = ["Hello", "World", "Again"]
list_numbers = [1, 2, 3]

# Get length
print(len(list_string))
# print(list_string[len(list_string)]) -> Mal
#print(list_string[len(list_string) - 1])

# Concatenate 2 list
print(list_string + list_numbers)

# Repetition of list
print(list_string * 3)

# Check membership
print("Hello" in list_string)
print(5 in list_numbers)

# Iteration
for x in list_string:
    print(x)
```

```
↪ 3
['Hello', 'World', 'Again', 1, 2, 3]
['Hello', 'World', 'Again', 'Hello', 'World', 'Again', 'Hello', 'World', 'Again']
True
False
Hello
World
Again
```

Slicing, indices y matrices

Las listas son secuencias, al igual que las tuplas y las cadenas de texto, por lo que las operaciones de indexación y *slicing* son las mismas

```
list_example = ["Hello", "World", "I'm", "John"]
```

```
# Simple indexing
simple_index = list_example[1]
print(simple_index)

# Negative indexing
negative_index = list_example[-1]
print(negative_index)
```

```
# Slicing
slicing = list_example[:2]
print(slicing)
```

```
↪ World
John
['Hello', 'World']
```

✓ Ejercicio

```
# TODO write some variables holding lists, both empty and populated
peñarol = []
print (peñarol)

# Add few elements to the empty list
peñarol.append(1891)
print (peñarol)
peñarol.insert(2,"Campeon del siglo")
print(peñarol)
peñarol.insert(1,"El")
print(peñarol)

# Check if the empty list is now empty (returning boolean)
print (peñarol == [])

# Get a sublist of one of the non-empty lists
nacional = peñarol[1:]
print (nacional)

# Concatenate the result with the empty list
print (peñarol+nacional)

# Check if one of the followin elements are in the concatenated list: 1, "hello", "world"
print (1 in peñarol)

# Create a sublist with the concatenated list elements that match these elements: 0, 1, 2, 3, "hello", "world"
```

```
↗ []
[1891]
[1891, 'Campeon del siglo']
[1891, 'El', 'Campeon del siglo']
False
['El', 'Campeon del siglo']
[1891, 'El', 'Campeon del siglo', 'El', 'Campeon del siglo']
False
Índice: 0, Valor: 1
Índice: 1, Valor: 2
Índice: 2, Valor: 3
Índice: 3, Valor: 4
```

▼ Solución

Haz click para ver el resultado

```

# TODO write some variables holding lists, both empty and populated
empty_list = []
car_list = ["Opel", "Renault", "Citroen", "Ford"]

# Add few elements to the empty list
empty_list.append(1)
empty_list.append(2)
print(empty_list)

# Check if the empty list is now empty (returning boolean)
check_one = len(empty_list) == 0
check_two = not empty_list
print(check_one)
print(check_two)

# Get a sublist of one of the non-empty lists
subcar_list = car_list[:2]
print(subcar_list)


# Concatenate the result with the empty list
new_list = subcar_list + empty_list
print(new_list)

# Check if one of the followin elements are in the concatenated list: 1, "hello", "world"
print(1 in new_list)
print("hello" in new_list)
print("world" in new_list)

# Create a sublist with the concatenated list elements that match these elements: 0, 1, 2, 3, "hello", "world"
check_list = [0, 1, 2, 3, "hello", "world"]
targeted_list = [0, 1, 3, 6, 7, "hello", "patata"]
print(list(set(targeted_list) & set(check_list)))

check_list = [0, 1, 2, 3, "hello", "world"]
targeted_list = [0, 1, 3, 6, 7, "hello", "patata"]
new_list = []
for element in targeted_list:
    if element in check_list:
        new_list.append(element)
print(new_list)

```

 [1, 2]
 False
 False
 ['Opel', 'Renault']
 ['Opel', 'Renault', 1, 2]
 True
 False
 False
 [0, 1, 3, 'hello']
 [0, 1, 3, 'hello']

Machine learning


[Numpy](#) es una librería en Python para crear y manipular **matrices**, la estructura principal dentro de los algoritmos de Machine Learning. Las [Matrices](#) son objetos que se usan para almacenar valores en columnas y filas.

Como veremos, las Listas son Matrices de un solo nivel, podemos crear multiniveles de forma sencilla. Numpy permite hacer operaciones con matrices de una forma muy similar a las listas en Python. Vamos a ver algunos ejemplos:

```

scalar = 1
vector = [1, 2, 3]
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix)

```

 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

```
import numpy as np

one_dimensional_array = np.array([1.2, 2.4, 3.5, 4.7, 6.1, 7.2, 8.3, 9.5])
print(one_dimensional_array)

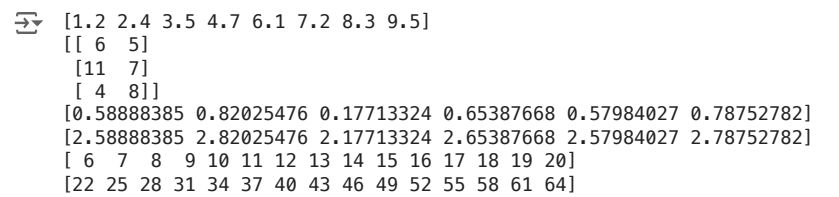
two_dimensional_array = np.array([[6, 5], [11, 7], [4, 8]])
print(two_dimensional_array)

random_floats_between_0_and_1 = np.random.random([6])
print(random_floats_between_0_and_1)

random_floats_between_2_and_3 = random_floats_between_0_and_1 + 2.0
print(random_floats_between_2_and_3)

# label = (3)(feature) + 4

feature = np.arange(6, 21)
print(feature)
label = (feature * 3) + 4
print(label)
```



```
[1.2 2.4 3.5 4.7 6.1 7.2 8.3 9.5]
[[ 6  5]
 [11  7]
 [ 4  8]]
[0.58888385 0.82025476 0.17713324 0.65387668 0.57984027 0.78752782]
[2.58888385 2.82025476 2.17713324 2.65387668 2.57984027 2.78752782]
[ 6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
[22 25 28 31 34 37 40 43 46 49 52 55 58 61 64]
```

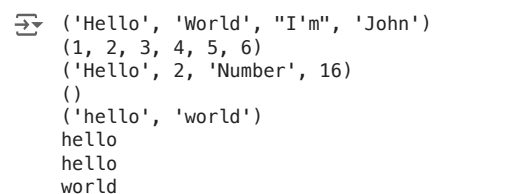
▼ Tuplas

Las tuplas son secuencias inmutables de objetos en Python. Son muy parecidas a las listas con la diferencia que las listas es un conjunto de elementos variables mientras que las tuplas son inmutables. Para crear una tupla, se definen unos valores separados por comas contenidos entre paréntesis.

```
# Tuple examples
tuple_example_strings = ("Hello", "World",
                        "I'm", "John")
print(tuple_example_strings)
tuple_example_number = (1,2,3,4,5,6)
print(tuple_example_number)
tuple_example_mix = ("Hello", 2, "Number", 16)
print(tuple_example_mix)
another_tuple = tuple()
print(another_tuple)

tuple_not_parenthesis = "hello", "world"
print(tuple_not_parenthesis)
value_1, value_2 = tuple_not_parenthesis
print(value_1)
```

```
hello, world = "hello", "world"
print(hello)
print(world)
```



```
('Hello', 'World', 'I'm', 'John')
(1, 2, 3, 4, 5, 6)
('Hello', 2, 'Number', 16)
()
('hello', 'world')
hello
hello
world
```

Las tuplas, al ser inmutables, no pueden **actualizar** ni **eliminar** elementos ya definidos, solo es posible acceder a sus valores.

```
# Index
element_1 = tuple_example_strings[0]
print(element_1)

# Slicing
mult_elements = tuple_example_number[:3]
print(mult_elements)
```

```

Hello
(1, 2, 3)

```

Las tuplas también pueden definirse sin paréntesis, forma muy común de ver en la devolución de una función.

```

def get_tuple():
    return "Hello", "World"

```

```

hello, world = get_tuple()

```

```

print(f'{hello}, {world}')
print (get_tuple())

```

```

Hello, World
('Hello', 'World')

```

```

def get_info():
    return ["Lucas", 184]

```

```

info = get_info()
print(f'name: {info[0]}, height: {info[1]}')

```

```

name: Lucas, height: 184

```

```

def sum_numbers(a, b):
    return a + b

```

```

result = sum_numbers(2, 3)
print(result)

```

```

5

```

Otra propiedad interesante de las tuplas es que al almacenarse en una variable pueden extraerse con una asignación múltiple

```

new_tuple = ("Hello", "World", 1, 10)
first, second, third, fourth = new_tuple
print(f'first: {first}, second: {second}, third: {third}, fourth: {fourth}')

```

```

first: Hello, second: World, third: 1, fourth: 10

```

Listas y Tuplas, artículo de refuerzo:

<https://realpython.com/python-lists-tuples/>

▼ Ejercicio

```

# Create an empty tuple with the built-in python declaration
empty_tuple = ()

```

```

# Create a tuple with different data types (more than one)
new_tuple = ("uno", "dos", 3, 4)

```

```

# Print the second value of the new tuple
print (new_tuple [1])

```

```

# Assing those values to multiple variables and print them
one, two, three, four = new_tuple
print (f'Hola {one}, como estas {two}')

```

```

# Add a value to a tuple

```

```

dos
Hola uno, como estas dos

```

Haz doble clic (o pulsa Intro) para editar

> Solución

Haz click para ver el resultado

[] ↪ 1 celda oculta

✓ Diccionario

Los diccionarios son una colección de valores sin orden que pueden intercambiarse y están indexados por una clave. Estas claves deben ser únicas, es decir, no puede haber repetición y mientras que el valor de las claves puede mutar, éstas son inmutables.

Los diccionarios se definen en python con un conjunto de clave/valor separado por dos puntos (:), estos elementos están separados por comas y agrupados por llaves.

```
# Dictionary definition
empty_dict = {}
print(empty_dict)
string_dict = {"first": "Hello",
               "second": "World"}
print(f'{string_dict["first"]} {string_dict["second"]}')
mixed_dict = {"first": "values", "second": 34}
print(mixed_dict)
```

```
{}
```

```
Hello World
```

```
{'first': 'values', 'second': 34}
```

Acceder a valores

Para acceder a valores en un diccionario usamos una sintaxis similar que en las secuencias, elegimos la clave que queremos extraer entre corchetes, siempre teniendo en cuenta que el acceso de una clave inexistente da como resultado un *Error de Clave*

```
# Accesing Values
string_dict = {"first": "Hello", "second": "World"}
ex_str = string_dict["first"]
print(ex_str)
```

```
ex_str_2 = string_dict.get("second", "failed")
print(ex_str_2)
```

```
def replicate_get(dictionary, key, default_value):
    try:
        return dictionary[key]
    except:
        return default_value
```

```
replicate_get(string_dict, "asdfasdf", "failed")
```

```
{}
```

```
Hello
```

```
World
```

```
'failed'
```

```
# Error Accesing Values
string_dict = {"first": "Hello", "second": "World"}
print(string_dict)
ex_str = string_dict["third"]
```

```
{'first': 'Hello', 'second': 'World'}
```

```
KeyError                                Traceback (most recent call last)
<ipython-input-25-0a7088ca5225> in <cell line: 4>()
      2 string_dict = {"first": "Hello", "second": "World"}
      3 print(string_dict)
----> 4 ex_str = string_dict["third"]

KeyError: 'third'
```

Añadir un elemento

Para añadir un elemento solo hay que asignar un valor a una nueva clave entre corchetes.

```
# Adding element

dict_test = {"first": "Hello", "second": "World"}
print(dict_test)

dict_test["third"] = "!"
dict_test["fourth"] = 4
print(f'We have this {dict_test}')
```

```

↵ {'first': 'Hello', 'second': 'World'}
   We have this {'first': 'Hello', 'second': 'World', 'third': '!', 'fourth': 4}

```

Actualizar un elemento

Actualizar elementos funciona de manera similar que añadir, solo hay que elegir una clave ya establecida y asignarle un nuevo valor

Updating element

```
dict_test = {"first": "Hi", "second": "World"}
print(dict_test)
```

```
dict_test["first"] = "Hello"
print(f"We have this {dict_test}")
```

```

↵ {'first': 'Hi', 'second': 'World'}
   We have this {'first': 'Hello', 'second': 'World'}

```

Eliminar elementos

Hay tres acciones para eliminar elementos en los diccionarios:

- Borrar una entrada del diccionario.
- Borrar todas las entradas del diccionario.
- Borrar el diccionario entero.

Deleting elements

```
dict_test = {"first": "Hi", "second": "World"}
del dict_test["second"] # remove entry with key `second`
print(dict_test)
dict_test.clear() # remove all the entries in the dictionary
print(dict_test)
del dict_test # remove the dictionary
```

Funciones y métodos

Hay varios operadores elementales dentro de las funciones y métodos de un diccionario.

```
dict_test = {"first": "Hi", "second": "World"}
dict_2_test = {"first": "Hi", "second": "potato", "third": "tomato"}
```

```
# Get lenght
print(len(dict_test))
print(len(dict_2_test))
```

```
# String representation
print(str(dict_test))
```

```
# Copy a dictionary
dict_test_copy = dict_test.copy()
dict_test_no_copy = dict_test
# dict_test_copy = dict_test
print(dict_test_copy)
dict_test_copy["first"] = "Patata"
print(dict_test_copy)
print(dict_test)
dict_test_no_copy["first"] = "Patata2"
print(dict_test_no_copy)
print(dict_test)
```

```
# Get an element
first_element = dict_test.get("first", "not found")
fourth_element = dict_test.get("fourth", "not found")
print(f"First element: {first_element}")
print(f"Fourt element: {fourth_element}")
```

```
# Return all keys
print(dict_2_test.keys())
```

```

↵ 2
   3
   {'first': 'Hi', 'second': 'World'}
   {'first': 'Hi', 'second': 'World'}
   {'first': 'Patata', 'second': 'World'}
   {'first': 'Hi', 'second': 'World'}
   {'first': 'Patata2', 'second': 'World'}

```

```
{'first': 'Patata2', 'second': 'World'}
First element: Patata2
Fourt element: not found
dict_keys(['first', 'second', 'third'])
```

```
dict_test = {"first": "Hi", "second": "World"}
```

```
dict_test_copy = dict_test.copy()
#dict_test_copy = dict_test
```

```
dict_test_copy.clear()
print(dict_test_copy)
print(dict_test)
```

▼ Ejercicio

```
# TODO write some variables holding dictionaries, both empty and populated
colores = {}
numeros = {}
```

```
# Add few elements to the empty dictionary
colores ["primero"] = "rojo"
print (colores)
```

```
# Access one of the elements of the dictionary by its key
primero = colores ["primero"]
print (primero)
```

```
# Update an element of the list
colores ["primero"] = "azul"
print (colores)
```

```
# Remove all the entries of the dictionary
colores.clear()
print (colores)
```

```
# Check if the empty dictionary is in fact empty (by returning a boolean)
print(colores=={})
```

```
# Check if this key is in the populated dictionary: "world"
print ("World" in colores)
```

```
# Try to get this element of the dictionary, if fails return "not found": "hello"
```

```
↗ {'primero': 'rojo'}
rojo
{'primero': 'azul'}
{}
True
False
```

▼ Solución

Haz click para ver el resultado

```
# TODO write some variables holding dictionaries, both empty and populated
new_dictionary = {"first": "Hello", "second": "World", "third": "Hurray"}
empty_dictionary = {}

# Add few elements to the empty dictionary
empty_dictionary["user"] = "test"
empty_dictionary["password"] = "test"
print(empty_dictionary)

# Access one of the elements of the dictionary by its key
username = empty_dictionary["user"]
print(username)

# Update an element of the list
empty_dictionary["password"] = "!890fas78ds"

# Remove all the entries of the dictionary
empty_dictionary.clear()

# Check if the empty dictionary is in fact empty (by returning a boolean)
print(len(empty_dictionary) == 0)
print(not empty_dictionary)

# Check if this key is in the populated dictionary: "world"
print("world" in new_dictionary)

# Try to get this element of the populated dictionary, if fails return "not found": "hello"
element = new_dictionary.get("hello", "not found")
print(element)
```

Machine learning

Ahora vamos a hablar de [DataFrames](#), que son la estructura central de la API de pandas. Vamos a ver cómo Pandas usa sintaxis de clave valor parecidas a los diccionarios para crear sus estructuras.

Un DataFrame es similar a un **excel**. Como una hoja de cálculo:

- Un DataFrame almacena valores en celdas
- Un DataFrame tiene columnas y filas numeradas

```
import numpy as np
import pandas as pd

# Create and populate a 5x2 NumPy array.
my_data = np.array([[0, 3], [10, 7], [20, 9], [30, 14], [40, 15]])
print(my_data)

# Create a Python list that holds the names of the two columns.
my_column_names = ['temperature', 'activity']

# Create a DataFrame.
my_dataframe = pd.DataFrame(data=my_data, columns=my_column_names)

# Print the entire DataFrame
print(my_dataframe)

print("=====")

# Create a new column named adjusted.
my_dataframe["adjusted"] = my_dataframe["activity"] + 2

# Print the entire DataFrame
print(my_dataframe)

print("=====")

print("Rows #0, #1, and #2:")
print(my_dataframe.head(3), '\n')

print("Row #2:")
print(my_dataframe.iloc[[2]], '\n')

print("Rows #1, #2, and #3:")
print(my_dataframe[1:4], '\n')

print("Column 'temperature':")
print(my_dataframe['temperature'])
```

```
→ [[ 0  3]
    [10  7]
```

```

[20  9]
[30 14]
[40 15]]
  temperature  activity
0           0         3
1          10         7
2          20         9
3          30        14
4          40        15
=====
  temperature  activity  adjusted
0           0         3         5
1          10         7         9
2          20         9        11
3          30        14        16
4          40        15        17
=====
Rows #0, #1, and #2:
  temperature  activity  adjusted
0           0         3         5
1          10         7         9
2          20         9        11

Row #2:
  temperature  activity  adjusted
2          20         9        11

Rows #1, #2, and #3:
  temperature  activity  adjusted
1          10         7         9
2          20         9        11
3          30        14        16

Column 'temperature':
0           0
1          10
2          20
3          30
4          40
Name: temperature, dtype: int64

```

✓ Operadores

Los operadores son construcciones de Python que pueden manipular el valor de los operandos.

✓ Operadores Aritméticos

Los operadores aritméticos modifican el valor de una variable. Existe un orden de precedencia de como se realizan las operaciones.

Operator	Description
(expressions...), [expressions...], {key: value...}, {expressions...}	Binding or parenthesized expression, list display, dictionary display, set display
x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference
await x	Await expression
**	Exponentiation 6
+=x, -=x, ~x	Positive, negative, bitwise NOT
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder 5
+=, -	Addition and subtraction
<<, >>	Shifts
&	Bitwise AND
^	Bitwise XOR
in, not in, is, is not, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
not x	Boolean NOT
and	Boolean AND
or	Boolean OR
if - else	Conditional expression
lambda	Lambda expression
:=	Assignment expression

```
# Ejemplo de operador aritmético
a = 30
b = 50

# Addition
add = a + b
print(f"add: {add}")

# Subtraction
sub = a - b
print(f"sub: {sub}")

# Multiplication
mul = a * b
print(f"mul: {mul}")


# Division
div = a / b
print(f"div: {div}")

# Module
modd = b % a
print(f"modd: {modd}")

# Exponent
exp = a**b
print(f"exp: {exp}")
```




✓ Ejercicio



```
# TODO write a program that ask for a 24 hour date and conver it into 12 hour date.
# Ex. Input -> 17 Output -> 5
hour = input("Indicame la hora en formato 24hs ")
print (f"A esa hora le corresponde {int(hour) % 12}")
```

```
# Add parenthesis to the expression to match the following output ----> 7:
```

```
result = 140 + 20 * 30 + 47 / (12 * 20)
print(f"Expected result to be 7 is: {result}")
```



```
Indicame la hora en formato 24hs20
A esa hora le corresponde 8
Expected result to be 7 is: 740.1958333333333
```


✓ Solución

Haz click para ver el resultado

```
# TODO write a program that ask for a 24 hour date and conver it into 12 hour date.
# Ex. Input -> 17 Output -> 5
hour = input("Give me an hour: ")
print(f"Hour ----> {int(hour) % 12}")
```

```
# Add parenthesis to the expression to match the following output:
```

```
result = (140 + 20 * (30 + 47)) / (12 * 20)
print(f"Expected result to be 7 is: {result}")
```



```
Give me an hour: 17
Hour ----> 5
Expected result to be 7 is: 7.0
```

✓ Operadores de comparación

Comparan los valores de ambos lados de la operación y decide la relación entre ambos.

```
# Ejemplos de operadores de comparación
a = 30
b = 50
```