

Sesión 2

✓ Control de flujo

En el nivel más elemental, un programa no es más que una ejecución secuencial de instrucciones. Esta ejecución tiene un flujo predefinido que el desarrollador querrá controlar con diversas técnicas.

Hay veces que una instrucción se tiene que repetir tantas veces como elementos de una lista, otras instrucciones solo se deben ejecutar dependiendo del resultado de otra...

Para todos estos casos Python provee de una sintaxis que permite controlar el flujo de la aplicación.

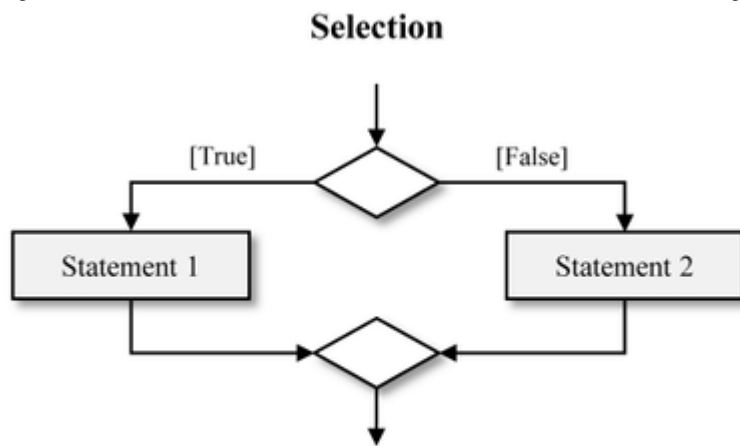
✓ Decisión condicional

Una decisión condicional permite ejecutar deliberadamente instrucciones dependiendo de una condición. Esta condición se reduce a un elemento Booleano con dos valores, verdadero o falso.

Python provee de tres tipos de sintaxis para la decisión condicional:

- **If:** La declaración más sencilla, consiste en una expresión booleana seguida de una o varias instrucciones
- **if...else:** Consiste en una expresión booleana con una o varias instrucciones seguida de una declaración **else** que se ejecuta cuando la expresión condicional es **False**
- **Expresiones if anidadas:** Los condicionales pueden anidarse para generar otras condiciones en cascada.

Hay un par de reglas en Python para ciertos tipos especiales, todos los valores que **no son cero** y no son **conjuntos vacíos (None)** se asumen como valores verdaderos (**true**). Los valores que son **cero** o **null** son asumidos como falsos (**false**).



```
# If example
counter = 4
if counter == 5: #conditional test
    print("We've reached 5")
```

```
# If..else example
if counter == 6:
    print("We've reached 6")
    print("Hello")
else:
    print("We've got something else")
```

```
counter += 1
# Nested if
if counter == 1:
    print("We've reached 1")
elif counter == 2:
    print("We've reached 2")
elif counter == 3:
    print("We've reached 3")
elif counter == 4:
    print("We've reached 4")
elif counter == 5:
    print("We've reached 5")
else:
    print("We've got more than that")
```

⇒ We've got something else
We've reached 5

✓ Ejercicio

```
# Write "all present" if all of these words are in a given phrase: Hello, world,
phrase = "Hello world i'm writting this story"
if ("Hello" in phrase and "world" in phrase and "story" in phrase):
    print ("all present")

# Write "one or more present" if all of these words are in a given phrase and wri
⇒ all present
```

✓ Solución

Haz click para ver el resultado

```
# Write "all present" if all of these words are in a given phrase: Hello, world,
phrase = "Hello world i'm writting this story"
if("Hello" in phrase and "world" in phrase and "story" in phrase):
    print("all present")

# Write "one or more present" if all of these words are in a given phrase and wri
if("panda" in phrase or "robert" in phrase or "python" in phrase):
    print("one or more present")
else:
    print("none present")

⇒ all present
   none present
```

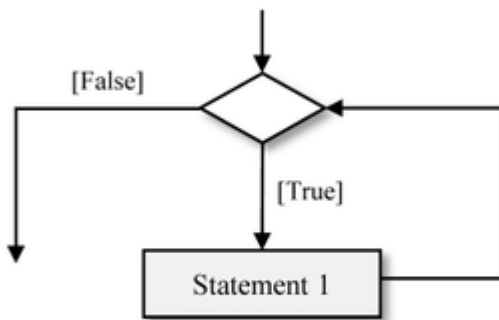
✓ Bucle "for"

Hay situaciones en las que es necesario que un bloque de instrucciones se ejecute una cantidad determinada de veces, bien sea para iterar entre múltiples elementos, crear nuevas secuencias...

Para eso existen declaraciones de bucles. En el caso del bucle for, se itera una secuencia hasta que no queden más elementos y se ejecuta múltiples veces una instrucción.

```
for element in sequence:
    statement(s)
```

Iteration



```
# For loop example
elements = [1,2,3,4,5,6,7,8]
```

```
for element in elements:
    print("Hola")
```

```
for letter in "Hello World":
    print(letter)
```

```
# For loop with range
for index, element in enumerate(elements):
    if element % 2 == 0:
        print(f"Element {element} is in index {index}")
```

```
⇒ Hola
    Hola
    Hola
    Hola
    Hola
    Hola
    Hola
    Hola
    Hola
    H
    e
    l
    l
    o

    W
    o
    r
    l
    d
    Element 2 is in index 1
    Element 4 is in index 3
    Element 6 is in index 5
    Element 8 is in index 7
```

```

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]])


# 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)

for index, row in my_dataframe.iterrows():
    if index <= 2:
        print(index, row.temperature, row.activity)

```



	temperature	activity
0	0	3
1	10	7
2	20	9
3	30	14
4	40	15

```

0 0 3
1 10 7
2 20 9

```

✓ Ejercicio

```

# Sum all the elements of the given list
a = [10,20,30,20,10,50,60,40,80,50,40]

# Get the largest number from the list

# Get the smallest number from the list

# Create a list without the duplicates of the list

```

✓ Solución

Haz click para ver el resultado

```
# Sum all the elements of the list
## first method
a = [10,20,30,20,10,50,60,40,80,50,40]
sum_list = 0
for x in a:
    sum_list += x
print(sum_list)
## second method
print(sum(a))

# Get the largest number from a list
## first method
max_number = 0
for x in a:
    if(x > max_number):
        max_number = x
print(max_number)

## second method
print(max(a))

# Get the smallest number from a list
## first method
min_number = max(a)
for x in a:
    if(x < min_number):
        min_number = x
print(min_number)

## second method
print(min(a))

# Create a list without the duplicates of the given list
## first method
a = [10,20,30,20,10,50,60,40,80,50,40]

uniq_items = []
for x in a:
    if x not in uniq_items:
        uniq_items.append(x)
print(uniq_items)

## second method
uniq_items_comp = list(set(a))
print(uniq_items_comp)
```

✓ Bucle "while"

Mediante el bucle **while** podemos repetir un conjunto de instrucciones mientras se cumpla una condición. La sintaxis es muy sencilla:

```
while expression:
    statement(s)
```

En el momento en que la condición es falsa el bucle se interrumpe dejando de ejecutar la instrucción.

El bucle **while** es más peligroso de usar que otras declaraciones ya que puede llevar a bucles infinitos si no se deja de cumplir una condición.

```
# While loop
counter = 1

while (counter < 9):
    print(counter)
    counter += 1
```

```
⇒ 1
   2
   3
   4
   5
   6
   7
   8
```

```
# While loop infinite DO NOT EXECUTE

counter = 1

while (counter != 2):
    print(counter)
    counter += 2
```

✓ Ejercicio

```
# Guess a number between 1 to 9 finish the program (get the input from the user u
import random
target_num, guess_num = (random.randint(1, 10), 0)
```

✓ Solución

Haz click para ver el resultado

```
# Guess a number between 1 to 9 finish the program
import random
target_num, guess_num = (random.randint(1, 10), 0)
while target_num != guess_num:
    guess_num = int(input('Guess a number between 1 and 10 until you get it right
print('Well guessed!')
```

✓ Funciones

Una función es un bloque organizado de código reutilizable. Las funciones otorgan modularidad a una aplicación y un nivel de reusabilidad. Hay varios consejos que debe seguir una función:

- Las funciones deben ser lo más nucleares posibles: Una función debe realizar una sola tarea con un objetivo claro
- Cualquier código que se repita en tu programa es candidato de convertirse en una función.
- Las funciones deben ser descriptivas en su funcionamiento: El nombre de la función debe dar una descripción aproximada de su funcionamiento.

Definición

Una función se define con la siguiente sintaxis en Python:

1. Los bloques de una función empiezan con la palabra reservada **def** seguida de paréntesis.
2. Cualquier parámetro o argumento debe declararse dentro de los paréntesis.
3. El bloque de la función se separa con los dos puntos ":" seguido de una indentación.
4. La palabra reservada **return** termina la ejecución de una función devolviendo opcionalmente un valor.

```
def function_example(parameter):
    """DocSttring"""
    print(parameter)
    return
```

Aquí podemos ver un ejemplo de función, tiene un Docstring, que es documentación acerca de la función y sus parámetros, el cuerpo donde se ejecuta el código y luego la llamada de **return** para salir de la función. A partir de aquí una función podrá ser ejecutada tantas veces como se quiera al invocar una llamada a la función.

```
function_example("Hola Mundo")
```

⇒ Hola Mundo

Se puede realizar tantas llamadas como sea necesario, cambiando el valor de los parámetros cuando se quiera.

```
function_example("Que tal estás")
```

⇒ Que tal estás

✓ Ejercicio

```
# Create a function that takes two numbers and returns the sum
def suma (a,b) :
    return a + b

print (suma(2,3))
```

⇒ 5

✓ Solución

Haz click para ver el resultado

```
# Create a function that takes two numbers and returns the sum
def sum_numbers(a, b):
    return a + b
```

Haz doble clic (o pulsa Intro) para editar

✓ Parámetros por valor o por referencia

Todos los parámetros **de tipos no primitivos** en Python se pasan por referencia. Esto significa que si cambias el valor dentro de una función también cambiará fuera de ella, y es un error muy común que se comete en programación.

Visualmente se puede diferenciar el paso de parámetros de valor y de referencia con este gif:

pass by reference*fillCup()**pass by value**fillCup()*

www.mathwarehouse.com

Como podemos observar, al pasar por referencia, si cambiamos el valor interno en la función cambiará en la variable externa, en cambio si la cambiamos al pasar la variable por valor no cambia en la variable externa.

Esto se puede ver también en código:

```
def empty_list(list):
    list.clear()
    print(f"Dentro de empty_list: {list}")
```

```
# Reference
list_reference = [1,2,3]
print(list_reference)
empty_list(list_reference)
print(list_reference)
```

```
⇒ [1, 2, 3]
   Dentro de empty_list: []
   []
```

```
# Value
list_value = [1,2,3]
print(list_value)
empty_list(list_value.copy())
print(list_value)
```

```
⇒ [1, 2, 3]
   Dentro de empty_list: []
   [1, 2, 3]
```

```
def save_data_db(list):
    print("Saving data to db")
    list.clear()
    print(f"Emptyin list just for releasing in the db: {list}")
```

```

persisten_data = [0, 2, 3, 4, 5]
persistent_data_2 = [0, 2, 3, 4, 5]
persistent_data_backup = persisten_data
print(persisten_data)
save_data_db(persistent_data_backup)
save_data_db(persistent_data_2[:])
print(persisten_data)
print(persistent_data_2)

```

```

⇒ [0, 2, 3, 4, 5]
Saving data to db
Emptyin list just for releasing in the db: []
Saving data to db
Emptyin list just for releasing in the db: []
[]
[0, 2, 3, 4, 5]

```

✓ Parámetros

En Python pueden utilizarse cuatro tipos de argumentos en una función:

1. Argumentos obligatorios
2. Argumentos por clave
3. Argumentos por defecto
4. Argumentos de longitud variable (simples o con clave/valor)

Cada uno de ellos tiene una propiedad diferente.

Argumentos obligatorios

Los argumentos obligatorios son aquellos que se pasan a una función en el orden correcto y con el número exacto de variables para argumentos.

```

def required_args(first, second, third):
    print(first)
    print(second)
    print(third)

```

```
required_args("first", "2", "third")
```

```

⇒ first
2
third

```

```

# Will fail
required_args()

```



```

-----
TypeError                                Traceback (most recent call last)
<ipython-input-28-cd453ab0a777> in <cell line: 2>()
      1 # Will fail
----> 2 required_args()

TypeError: required_args() missing 3 required positional arguments: 'first',
'second', and 'third'

```

Argumentos por clave

En python también puedes referenciar a un argumento por su nombre, pudiendo así cambiar el orden de llamada.

```

def keyword_args(first, second, third):
    print(first)
    print(second)
    print(third)

keyword_args(first="first", third="third", second="second")

```



```

first
second
third

```

Argumentos por defecto

Un argumento por defecto es un argumento que asume su valor por defecto, estos argumentos siempre van después de los argumentos por clave y pueden ser omitidos en la llamada a la función.

```

def default_args(first, second, third="third", fourth="4"):
    print(first)
    print(second)
    print(third)
    print(fourth)
    print("=====")

# You can call the third value
default_args("first", "second", "3")

# You can omit the value
default_args("first", "2")

# You can change the default value
default_args("first", "second", fourth="other")

```



```

first
second
3
4
=====

```

```

first
2
third
4
=====
first
second
third
other
=====

```

Podremos ver incluso objetos instanciados que se pasan como argumentos que a su vez tienen argumentos de iniciación:

```

model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=1, input_shape=[1])
])

```

✓ Return

La palabra reservada **return** sirve para terminar una función y devolver una expresión a la línea de código que la ha llamado. Una función sin **return** es lo mismo que si devuelve None.

Puede devolverse cualquier valor, y como mencionamos en el curso anterior, pueden **devolverse múltiple valores mediante tuplas**.

```

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

```

```

result = simple_sum(4, 2)
print(result)

```

```

def return_none(a, b):
    a + b
result = return_none(2,3)
print(result)

```

```

def increment_two_values(a, b):
    a += 1
    b += 1
    return a, b

```

```

inc_1, inc_2 = increment_two_values(3, 5)
tuple_test = increment_two_values(4, 8)
print(f"{inc_1} and {inc_2}")
print(tuple_test)

```

```

⇒ 6
   None
   4 and 6
   (5, 9)

```

✓ I/O

Print

Dentro de las funciones de entrada y de salida hemos estado viendo particularmente una durante las últimas clases, la función **print**. Si no os lo habíais preguntado antes, *print* es una función del sistema que imprime por pantalla una cadena de caracteres. Supongo que ya lo domniaréis pero se declara de la siguiente forma:

```

print("Hello World")
test = "Other"
print(test)

```

```

⇒ Hello World
   Other

```

Input

La función de entrada en python se escribe con la palabra reservada **input**. Seguro que os sonará de un ejercicio en el que pedíamos una entrada de caracteres, y como se supone, esta función recoge la entrada del usuario por línea de comandos y la devuelve en formato de cadena de caracteres.

```

name = input("Say your name:")

print(f"Hello, my name is {name}")

```

```

⇒ Say your name:Lucas
   Hello, my name is Lucas

```

Open

La función open permite abrir determinados archivos, su sintaxis es la siguiente:

```
> f = open('workfile', [flag])
```

Siendo el primer argumento la ruta del archivo a abrir y la segunda el modo en el que se abre, que puede ser:

- **r** -> Abre el fichero para solo lectura. El puntero se sitúa al principio del archivo. Es el modo por defecto.
- **rb** -> Abre el fichero en modo solo lectura con formato binario. El puntero se sitúa al principio del archivo.
- **r+** -> Abre el archivo en modo lectura y escritura. El puntero se sitúa al principio del archivo.
- **rb+** -> Abre el archivo en modo lectura y escritura con formato binario. El puntero se sitúa al principio del archivo.
- **w** -> Abre un archivo para solo escritura. Sobreescribe el fichero si existe. Si no existe, crea el archivo.
- **wb** -> Abre un archivo para solo lectura en formato binario. Sobreescribe el fichero si existe. Si no existe, crea el archivo.
- **a** -> Abre el archivo para agregar texto. El puntero se sitúa al final del archivo si existe. Si no existe crea uno nuevo para escritura.
- **a+** -> Abre el archivo para agregar texto en formato binario. El puntero se sitúa al final del archivo si existe. Si no existe crea uno nuevo para escritura.
- **ab** -> Abre el archivo para agregar texto y lectura. El puntero se sitúa al final del archivo si existe. Si no existe crea uno nuevo para escritura.
- **ab+** -> Abre el archivo para agregar texto y lectura en formato binario. El puntero se sitúa al final del archivo si existe. Si no existe crea uno nuevo para escritura.

File

La función **open** nos devuelve un objeto de tipo **file**. Con este objeto podremos realizar las siguientes operaciones:

- **file.closed** -> Nos devuelve si el archivo está cerrado.
- **file.mode** -> Nos indica el tipo de acceso.
- **file.name** -> Devuelve el nombre del archivo.
- **file.close()** -> Función que cierra y el archivo y serializa los cambios.

Abrir csv en pandas

Vamos a ver como procesar en Colab csv, utilizando una de las librerías antes mencionadas.

Para ello vamos a usar el dataset de [abalones](#) de la [uci](#).

Para ello, descargar el archivo **Abalones** de la carpeta principal.

Para importar los archivos, vamos a utilizar la librería `google.colab`, que permite controlar aspectos del cuaderno, para forzar la carga de archivos. Posteriormente con la librería `io` serializaremos estos datos.

En un código ejecutado en una máquina este paso no sería necesario, podríamos usar `pd.read_csv` en la carpeta donde se encontrase nuestro archivo.

```
import pandas as pd
from google.colab import files
import io

uploaded = files.upload()
abalone_train = pd.read_csv(io.BytesIO(uploaded['abalone.csv']), names=["Length",
                                "Viscera weight", "Shell weight", "Age"])

abalone_train.head()
```



Elegir archivos Sin archivos seleccionados Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving abalone.csv to abalone (5).csv

```
-----
KeyError                                Traceback (most recent call last)
<ipython-input-45-ef30925f1b47> in <cell line: 6>()
      4
      5 uploaded = files.upload()
----> 6 abalone_train = pd.read_csv(io.BytesIO(uploaded['abalone.csv']))
      7
      8 abalone_train.head()

KeyError: 'abalone.csv'
```

CSV

Uno de los formatos más comunes dentro del mundo de Data Science es organizar los datos en CSV delimitados, para ello hay que usar la función **open** y hacer un cast a **csvfile** y llamar al lector de csv, como en el siguiente ejemplo:

```
import csv

with open('data.csv') as csvfile:
    reader = csv.DictReader(csvfile, delimiter=';')
    for row in reader:
```

✓ Control de proyectos

✓ Gestión de módulos

Un módulo permite organizar de forma lógica el código en Python. A medida que va aumentando en complejidad el proyecto las líneas de código pueden crecer a un ritmo exponencial. Python no obliga al desarrollador a mantener el código separado, siempre puedes tener todo tu código recogido en un solo fichero, pero este se hace impracticable en un momento determinado.

Para solucionar este problema existen los módulos. Un módulo es un fichero Python donde se pueden definir funciones, clases y variables que puedes referenciar y añadir a otro código.

Este código puede importarse usando la palabra reservada **import** o **from** con el siguiente formato:

```
import module1, module2.....  
from module1 import [reference]  
  
# Import module  
import time  
  
# call module by calling its name  
seconds = time.time()  
  
print(seconds)
```

Por otro lado, con la notación **from ... import** podemos importar atributos específicos de un módulo al *namespace* actual.

```
from time import time  
  
seconds = time()  
  
print(seconds)
```

También podremos poner alias a nuestros **imports** para referenciar el paquete con el nombre que queramos

```
import tensorflow as tf  
import numpy as np
```

Cuando el interprete se encuentra la palabra reservada **import**, busca en la ruta declarada (puede ser relativa o absoluta) el fichero con el nombre declarado. Este fichero, al ser declarado se buscará en las siguientes secuencias hasta que se encuentre:

- En el directorio actual.
- Si no se encuentra en el directorio actual, Python busca en todos los directorios correspondientes a la variable de entorno PYTHONPATH.
- Si todo esto falla, Python comprueba la ruta por defecto. En UNIX por ejemplo es </usr/local/lib/python>.

La ruta de búsqueda de Python se encuentra en **sys.path**, esta ruta puede ampliarse en un proyecto para importar por defecto módulos.

✓ Gestión de Paquetes

Muchos módulos pueden empaquetarse en un conjunto de ficheros que aportan determinada utilidad. Estos módulos pueden gestionarse en nuestro proyecto a través de un **gestor de paquetes**.

[Pip](#) es el gestor de paquetes más popular de Python y permite instalar rápidamente módulos que necesitamos.

Los módulos publicados en pip se pueden encontrar en [PyPI](#), donde se puede encontrar información relativa a un paquete como el historial, referencia al código, licencias...

Para instalar un paquete, solo tenemos que ejecutar:

```
!pip install numpy
```

```
➞ Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
```

```
import numpy
```

```
print(numpy.__version__)
```

```
➞ 1.23.5
```

✓ Entornos Virtuales

Los entornos virtuales son una herramienta fundamental en el desarrollo de software en Python. Permiten crear instalaciones de Python aisladas, lo que es especialmente útil cuando trabajas en múltiples proyectos con diferentes requisitos de paquetes o dependencias. Son conceptos similares a los **entornos de anaconda**

Configuración inicial

Antes de comenzar, verifica si tienes el módulo venv instalado en tu sistema ejecutando `python -m venv --version`. Si no está instalado, puedes instalarlo utilizando el administrador de paquetes de Python: `pip install venv`.

Creación de un nuevo entorno virtual

Para crear un nuevo entorno virtual, ejecuta el siguiente comando en tu terminal:

```
python -m venv <nombre_entorno>
```

Esto creará un directorio con el nombre del entorno virtual y contendrá la instalación aislada de Python.

Activación y desactivación

Para activar y desactivar estos entornos, tendremos que acceder al proyecto donde hemos creado el entorno virtual y ejecutar:

```
source <nombre_entorno>/bin/activate
```

Y con esto entraremos en el entorno virtual. Para desactivarlo solo tendremos que ejecutar:

```
deactivate
```

Gestión de paquetes

Una vez activado el entorno virtual, podemos usar `pip install` para instalar dependencias. Una vez instaladas todas las dependencias, podemos hacer un **snapshot** de las dependencias con `pip freeze`, si queremos tener las dependencias listadas para compartir con otros colaboradores, podemos ejecutar lo siguiente:

```
pip freeze > requirements.txt
```

Generando la lista de dependencias del proyecto.

Fichero dependencias

Si deseas replicar el entorno en otro proyecto, ahora se puede usar el archivo `requirements.txt`. Para ello se puede compartir el proyecto, crear un entorno virtual e instalar las dependencias con:

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

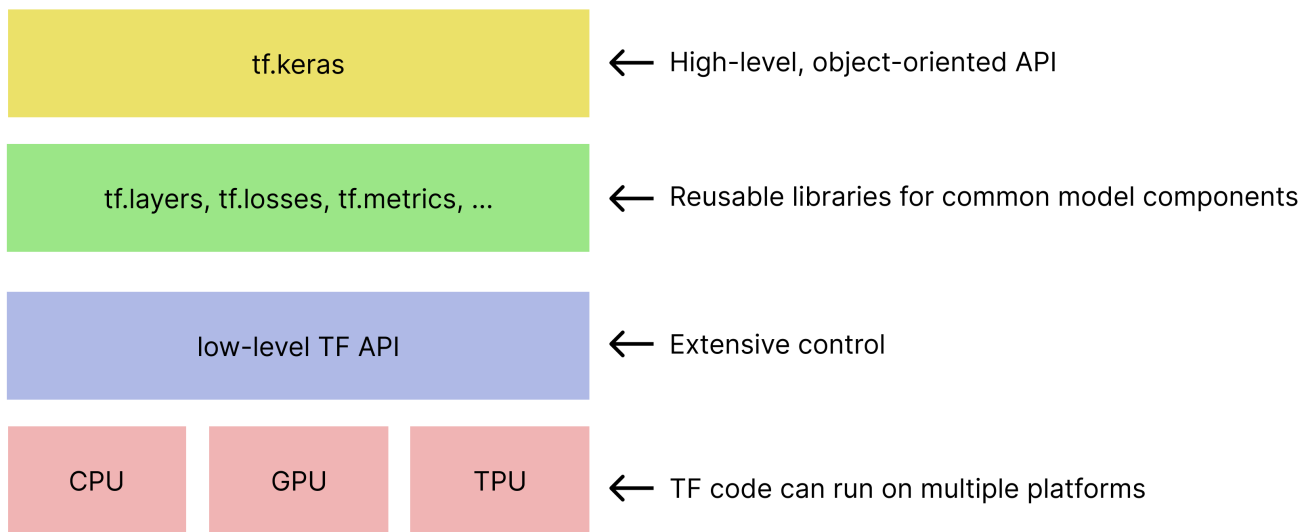
✓ Introducción a Tensorflow

Tensorflow es un framework computacional creado para construir modelos de *Machine Learning*. Tensorflow proporciona una variedad de herramientas que permiten construir modelos a distintos niveles de abstracción.

Así un programador puede usar las *APIs* de bajo nivel para construir modelos basados en una serie de operaciones matemáticas como usar las *APIs* de alto nivel para especificar arquitecturas predefinidas como regresiones lineales o redes neuronales.

Arquitectura

Esta foto, sacada de Google, nos muestra la arquitectura de Tensorflow:



Así pdoemos definir las siguientes capas:

- **Keras (tf.keras)** --> OOP API de alto nivel que desde Tensorflow 2.0 es la API central.
- **Estimator (tf.estimator)** --> OOP API de alto nivel, ha ido perdiendo tracción en los recientes años.
- **tf.layers/tf.losses/tf.metrics** --> Bibliotecas de modelos comunes.
- **Python Tensorflow** --> Wrap de Tensorflow en Python, parte de la API de bajo nivel.
- **C++ TensorFlow** --> Aplicación de TensorFlow escrita en C++ (POO), abstrae el hardware, parte de la API de bajo nivel.
- **CPU/GPU/TPU** --> Hardware usado por TensorFlow, parte de su potencia es que puede ejecutarse en muchas arquitecturas.

✓ Hello World en Tensorflow

En Tensorflow, se declara el helloworld importando la librería *tensorflow*. Las constantes analizadas para ejecutar en una sesión de tensorflow se declaran con `tf.constant` y se ejecuta

una interfaz sesión con *tf.Session()*

A partir de Tensorflow 2.0 los programas se ejecutan en "modo impaciente" por lo que no se necesitan sesiones para crear constantes

```
# Hello World In Tensorflow 1
# ATENCIÓN ESTO ESTÁ DEPRECADO NO USAR!!!!!!

import tensorflow as tf

with tf.compat.v1.Session() as sess: # NO HACE FALTA EN TENSORFLOW 2.0
    hello = tf.constant("Hello")
    world = tf.constant(" World!")
    hello_world = hello + world
    res = sess.run(hello_world) # NO HACE FALTA EJECUTAR ESTO
    print(res)
    print(hello_world)

# Hello World in Tensorflow 2
import tensorflow as tf

print(tf.__version__)

hello = tf.constant('Hello')
world = tf.constant('World!')
hello_world = hello + world

#print the message
print(hello_world)

⇒ 2.13.0
tf.Tensor(b'HelloWorld!', shape=(), dtype=string)

print(hello_world)
```

Esta variable conforma el protocolo del modelo de grafo computacional (computation graph model) de Tensorflow, que primero define que computos deben hacerse para luego ejecutarlos en un mecanismo externo.

✓ Conceptos Básicos

Vamos ahora a mencionar por encima los elementos clave con los que operar en **Tensorflow**

✓ Tensores

Escalar Vector Matriz Tensor

1

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$\begin{bmatrix} \begin{bmatrix} 1 & 2 \end{bmatrix} & \begin{bmatrix} 3 & 2 \end{bmatrix} \\ \begin{bmatrix} 1 & 7 \end{bmatrix} & \begin{bmatrix} 5 & 4 \end{bmatrix} \end{bmatrix}$$

Tensorflow opera en matrices multidimensionales o *tensores**. Se representan con el objeto `tf.Tensor`. Ya hemos introducido el concepto de calculos matriciales con **numpy**.

Los atributos más importantes de un **tensor** son:

- `Tensor.shape` : Indica el tamaño del tensor a lo largo de cada uno de sus ejes.
- `Tensor.dtype` : Proporciona el tipo de todos los elementos en el tensor.

```
import tensorflow as tf
```

```
first_tensor = tf.constant([[1., 2., 3.],
                           [4., 5., 6.]])
```

```
print(first_tensor)
print(first_tensor.shape)
print(first_tensor.dtype)
```

```
second_tensor = tf.constant([["a", "b", "c"],
                             ["e", "f", "g"],
                             ["c", "v", "o"]])
```

```
print(second_tensor)
print(second_tensor.shape)
print(second_tensor.dtype)
```

```
⇒ tf.Tensor(
  [[1. 2. 3.]
   [4. 5. 6.]], shape=(2, 3), dtype=float32)
(2, 3)
<dtype: 'float32'>
tf.Tensor(
  [[b'a' b'b' b'c']
   [b'e' b'f' b'g']
   [b'c' b'v' b'o']], shape=(3, 3), dtype=string)
(3, 3)
<dtype: 'string'>
```

✓ Variables

Los objetos `tf.Tensor` son inmutables. Para almacenar pesos de modelos u otros estados mutables, podemos hacer uso de `tf.Variable`

```
l0_weights = tf.Variable([0.0, 0.0, 0.0])
l0_weights.assign([2, 3, 4])
```

```
↳ <tf.Variable 'UnreadVariable' shape=(3,) dtype=float32, numpy=array([2., 3., 4.], dtype=float32)>
```

✓ Otros conceptos

El resto de conceptos los vamos a mencionar de pasada, ya que vamos a abarcar su funcionalidad con la API **Keras** por lo que no es necesario conocerlos en profundidad de momento.

- **Automatic differentiation:** Se utiliza para el cálculo de la **Descenso de Gradiente** para minimizar la función de pérdida.

```
x = tf.Variable(1.0)
```

```
def f(x):
    y = x**2 + 2*x - 5
    return y
```

```
with tf.GradientTape() as tape:
    y = f(x)
```

- **Grafos y tf.function:** Además de usar Tensorflow de forma interactiva, puedes usar `tf.function` para optimización de rendimiento y exportar el modelo una vez entrenado, ya que una vez ejecutado por primera vez devolverá un **grafo optimizado** representando los cálculos de Tensorflow.

```
@tf.function
def my_func(x):
    print('Tracing.\n')
    return tf.reduce_sum(x)
```

```
x = tf.constant([1, 2, 3])
my_func(x)
```

- **Modulos, capas y modelos:** `tf.Module` es una clase para procesar las `tf.Variables` y `tf.function` que hayamos declarados.

```

class MyModule(tf.Module):
    def __init__(self, value):
        self.weight = tf.Variable(value)

    @tf.function
    def multiply(self, x):
        return x * self.weight

mod = MyModule(3)
mod.multiply(tf.constant([1, 2, 3]))

save_path = './saved'
tf.saved_model.save(mod, save_path)

```

- **Bucles de entrenamiento:** Con esto podremos entrenar nuestros modelos

```

# Format training loop
for epoch in range(epochs):
    for x_batch, y_batch in dataset:
        with tf.GradientTape() as tape:
            batch_loss = mse_loss(quad_model(x_batch), y_batch)
            # Update parameters with respect to the gradient calculations
            grads = tape.gradient(batch_loss, quad_model.variables)
            for g,v in zip(grads, quad_model.variables):
                v.assign_sub(learning_rate*g)
        # Keep track of model loss per epoch
        loss = mse_loss(quad_model(x), y)
        losses.append(loss)
    if epoch % 10 == 0:
        print(f'Mean squared error for step {epoch}: {loss.numpy():0.3f}')

```

✓ Construcción de modelos en Keras

Como hemos comentado, desde Tensorflow 2.0 [keras](#) es la API por defecto para construir modelos. De momento en nuestros ejemplos hemos visto una sola forma de crear modelos, pero actualmente hay diferentes maneras, orientadas al nivel de experiencia y los requerimientos del proyecto. Estos tres modos son:

✓ API Sequential

La api `Sequential` es la mejor forma de construir modelos sencillos, si os fijáis, los ejemplos que hemos propuesto son con este método. Permite crear de forma sencillas modelos que tienen una entrada, una salida y múltiples capas. Es una opción muy buena para aprender a utilizar el Framework y asimilar conceptos.

La característica más destacable es que podemos pasar diferentes capas de forma rápida como un solo argumento en una **lista**

```
sequential_model = Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Sequential API
import tensorflow as tf
from tensorflow.keras.models import Sequential

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

sequential_model = Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])

sequential_model.compile(optimizer='adam',
                        loss='sparse_categorical_crossentropy',
                        metrics=['accuracy'])

sequential_model.fit(x_train, y_train, epochs=5)
sequential_model.evaluate(x_test, y_test)
```

```
Epoch 1/5
1875/1875 [=====] - 7s 3ms/step - loss: 0.2993 - acc: 0.0699
Epoch 2/5
1875/1875 [=====] - 7s 4ms/step - loss: 0.1421 - acc: 0.1327
Epoch 3/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.1041 - acc: 0.1875
Epoch 4/5
1875/1875 [=====] - 7s 4ms/step - loss: 0.0877 - acc: 0.2292
Epoch 5/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.0727 - acc: 0.2619
313/313 [=====] - 1s 2ms/step - loss: 0.0699 - acc: 0.2619
[0.06991760432720184, 0.979200005531311]
```

✓ API Functional

La API `Functional` es el método más popular de construir modelos en **Keras**. Puede hacer todo lo que hace la API `Sequential` y además permite múltiples entradas y salidas, ramificaciones y compartir capas. Es un método bastante sencillo y aún así permite una gran personalización y flexibilidad.

En este caso vamos a definir la entrada de forma separada y luego crear un objeto salida añadiendo todas las capas, para luego pasárselo al objeto `Model`

```
inputs = Input(shape=(28, 28))
x = Flatten()(inputs)
x = Dense(256, "relu")(x)
outputs = Dense(10, "softmax")(x)

model = Model(inputs=inputs, outputs=outputs, name="mnist_model")

# Functional API
import tensorflow as tf
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras import Input
from tensorflow.keras import Model

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

inputs = Input(shape=(28, 28))
x = Flatten()(inputs)
x = Dense(256, "relu")(x)
outputs = Dense(10, "softmax")(x)

functional_model = Model(inputs=inputs, outputs=outputs, name="mnist_model")

functional_model.compile(optimizer='adam',
                        loss='sparse_categorical_crossentropy',
                        metrics=['accuracy'])

functional_model.fit(x_train, y_train, epochs=5)
functional_model.evaluate(x_test, y_test)
```

➡ Epoch 1/5
 1875/1875 [=====] - 9s 5ms/step - loss: 0.2322 - acc: 0.1000
 Epoch 2/5
 1875/1875 [=====] - 7s 4ms/step - loss: 0.0945 - acc: 0.2000
 Epoch 3/5
 1875/1875 [=====] - 8s 5ms/step - loss: 0.0621 - acc: 0.3000
 Epoch 4/5

```
1875/1875 [=====] - 9s 5ms/step - loss: 0.0450 - acci  
Epoch 5/5  
1875/1875 [=====] - 7s 4ms/step - loss: 0.0329 - acci  
313/313 [=====] - 1s 2ms/step - loss: 0.0721 - accura  
[0.07211676239967346, 0.9776999950408936]
```

✓ Model Subclassing

Ya hemos adelantado este modo en la sección de **herencia** y es que a través de ese concepto vamos a definir nuestro modelo. Evidentemente este modo te da mucha más flexibilidad, ya que