Apuntes Fundamentos

Javier Plaza Rosique

Índice

1. Introducción.

En este documento, se podrán encontrar los conocimientos básicos sobre las siguientes herramientas:

- Git, y como server online del mismo GitHub.
- Linux.
- Python.
- Docker.
- SQL.

El objetivo final para mi es llegar a comprender los fundamentos de las herramientas anteriormente mencionadas. Aunque este documento lo publico por si hay alguna persona a la que le pueda llegar a ayudar.

Los conocimientos con los que ha sido redactado este documento son los adquiridos durante el módulo de Fundamentos del Master en Big Data que me encuentro en la actualidad cursando.

Cabe recalcar que ni mucho menos lo que esta en el presente documento es todo lo que se puede llegar a saber sobre las herramientas anteriormente mencionadas.

La manera de contactar conmigo (creador del documento) o de ver

- **Linkedin:** www.linkedin.com/in/javiplazarosique
- **GitHub:** https://github.com/JaviPlazaRosique
- **Correo electrónico:** j.plazarosique@gmail.com

2. Git.

Git es un sistema de control de versiones. Sirve para guardar el historial de cambios de los archivos de un proyecto, y también sirve para poder coordinar el trabajo entre varias personas sin que unos se pisen a otros.

2.1. Bases de Git.

Para entender el funcionamiento de Git, antes hay que tener varios conceptos claros. Los conceptos son los siguientes:

- Repositorio.
- Flujo de trabajo con Git: commit, update, push, pull.

Un **repositorio** de Git es una carpeta (como cualquier carpeta del ordenador), en donde se encuentra un subdirectorio (como un archivo) oculto llamado .git, en el cual se guardan los datos de las versiones de los cambios producidos dentro del repositorio. Al poder llegar a tener el subdirectorio descargado en el ordenador, no es necesario tener conexion a internet, pues todos los cambios se encuentran el subdirectorio, por lo que podremos realizar cualquier acción. El repositorio se puede encontrar en local (una capeta del ordenador) o en remoto. Los repositorios en remoto son copias de los repositorios alojados en servidores externos o en la nube, esto permite realizar el trabajo en equipo. Uno de los servicios de repositorios en remoto más extendido es GitHub, pero también se pueden encontrar otros como GitLab.

Commit es la acción que realizamos cada vez que realicemos cambios en el proyecto, y queramos guardarlos. En otras palabras un commit es una instantanea del proyecto a la hora de hacerlo. Dichos cambios se guardan en el repositorio local, es decir, en el ordenador. Las características que todo commit tiene son las siguientes:

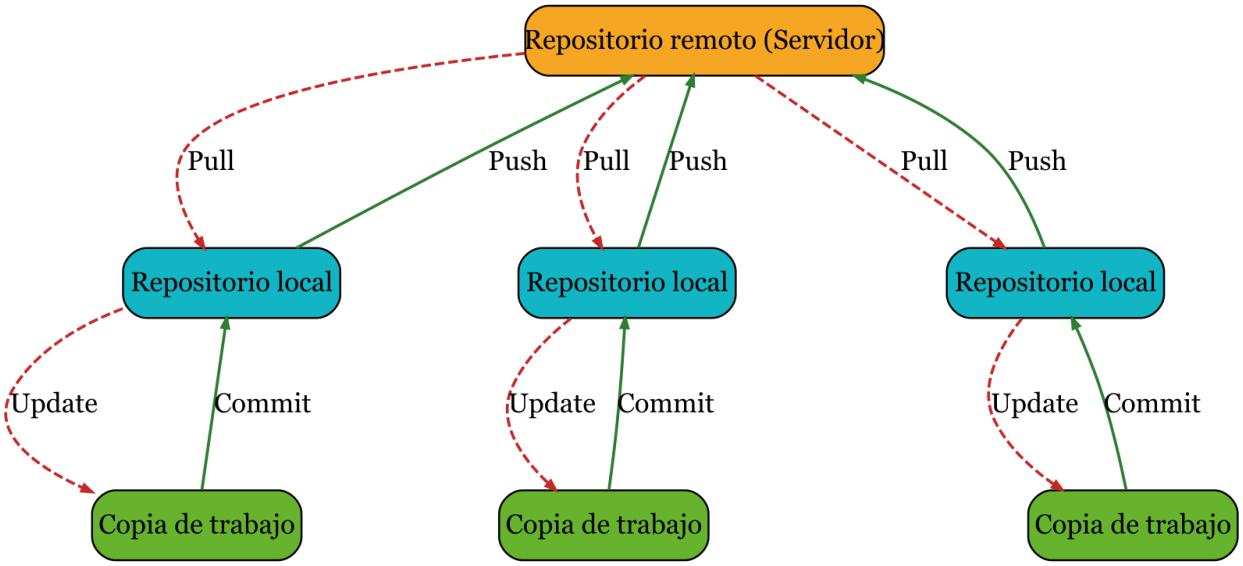
- Título.
- Descripción.
- Hash. Un hash es un código único, el cual es único por cada commit.
- Hora.

Update, en español actualizar, es llevar al espacio de trabajo la información existente en el repositorio local.

La acción **Push** envia los commits que se encuentran en el repositorio local (los nuevos, no los que ya están en remoto) al repositorio remoto. Con esta acción pueden surgir conflictos, es decir, cuando se intenta escribir por ejemplo en una línea ene la qeu ya había información. La solución del conflicto no la realiza ni Git, ni GitHub, dará error. Será la misma persona que realize el push u otra persona la que decide que hacer con el conflicto.

Pull es el update entre repositorios, es decir, que lleva al repositorio local la información recogida en el repositorio remoto. Al igual que en el push, también pueden surgir conflictos. En este caso los conflictos son más complicados que aparezcan, pues se supone que si alguien va a trabajar sobre un proyecto ya empezado, antes se bajará el proyecto al ordenador.

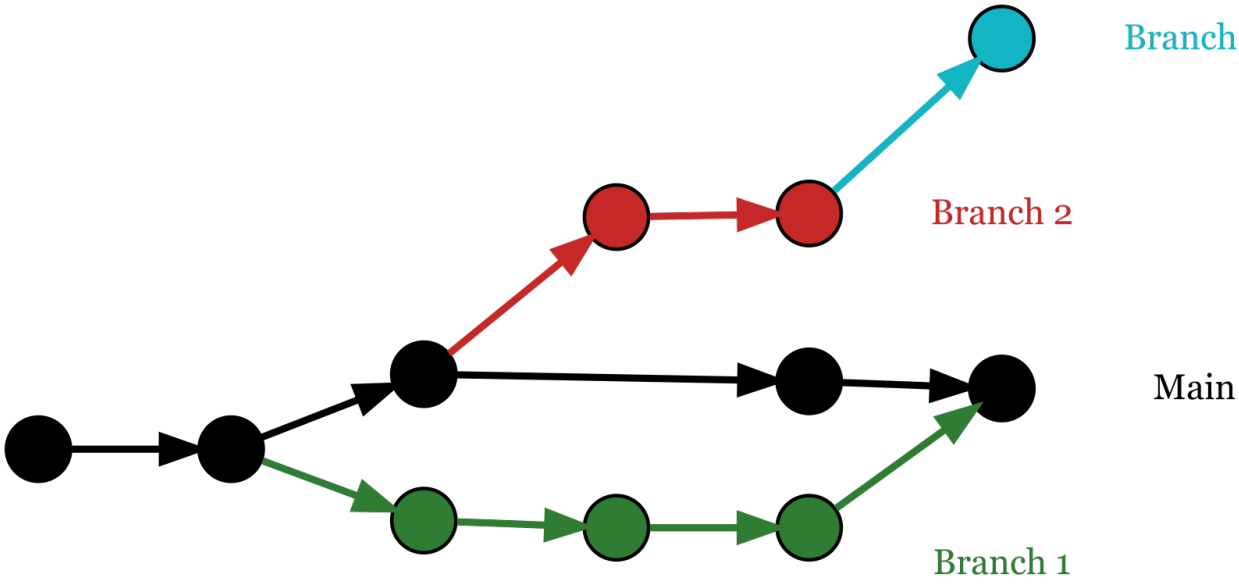
A continuación se encuentra una imagen, que refleja el flujo de trabajo con Git, con lo explicado anteriormente.



Además de lo explicado anteriormente, Git también permite realizar ramificaciones. Una rama es una línea de desarrollo independiente, que permite realizar cambios y probar nuevas cosas sin afectar a la rama principal. También permite trabajar en conjunto con un equipo, teniendo cada uno su rama. Las ramas pueden nacer de la rama principal, que por convención se llama "main" (anteriormente "master"), pero también pueden nacer de otras ramas. Cuando se termina el trabajo en una rama, se une (merge) a la rama de la que salió, pasando a la misma todos los commits realizados en la rama. Al realizar la unión entre ramas pueden ocurrir conflictos si se trabaja en las dos ramas implicadas.

Importante. Es recomendable nunca trabajar en la rama principal.

A continuación se puede observar una imagen donde se ve visualmente el funcionamiento de las ramas en Git.



Dentro de un repositorio de Git, se pueden crear una serie de archivos con funcionalidades especificas. Dichos archivos son el archivo llamado ".gitignore" y el archivo llamado ".gitkeep". Las funciones de cada archivo son las siguientes:

- **.gitkeep**. El archivo permite crear una carpera dentro del repositorio de git, sin la necesidad de tener que crear un archivo con contenido. Se emplea cuando se quiere crear la estructura del repositorio (carpetas), y aun no se quiere empezar a trabajar en el.
- **.gitignore**. El archivo sirve para excluir algunos archivos, por el motivo que sea. Al excluirlos, no los elimina, pero si que no los ratrea ni incluye en los commits. Los motivos para excluir algunos archivos pueden ser variados, pero los más comunes son por las vulnerabilidades que se pueden causar (contraseñas o similar) o por el tamaño de los archivos.

Las acciones que permite hacer Git como los commits, los push y los demás, se pueden hacer mediante interfaces simples, como puede ser el ejemplo de GitHub, el cual trabaja a partir de GitHub Desktop (luego se verá). Pero para poder trabajar sin una interfaz u en una interfaz que no "tengamos controlada", también se puede trabajar por comandos desde el bash.

De manera sencilla bash es un interprete de comandos del sistema, que nos permite realizar acciones como crear capetas, movernos por el ordenador (todo esto se verá en la parte de Linux), pero también permite tabajar con varias herramientas entre las cuales está Git. Según el sistema operativo del ordenador se puede usar el bash de manera diferente. Desde MacOS y Linux, viene por defecto en la terminal, y en Windows se puede emplear desde GitBash, el cual se instala junto a Git.

2.2. Comandos Git.

Los comandos más importantes para usar Git son los siguientes:

Creación del repositorio.

Si se quiere crear el repositorio en el directorio actual, es decir, si la carpeta en la que estamos situados queremos convertirla en un repositorio.

```
git init
```

En el caso de que queramos crear un repositorio de cero, se puede usar lo siguiente. Esto crea el repositorio (carpeta) con el nombre que se escriba en "". Para que se entienda, si estamos en una carpeta del ordenador, crea una carpeta dentro de ella con el nombre introducido, siendo esta el repositorio.

```
git init <directorio>
```

Conexión del repositorio local con el repositorio remoto.

En el caso de conectar un repositorio local con un repositorio remoto, se puede hacer en ambas vías. Es decir que se puede conectar un repositorio local con contenido a un repositorio remoto vacío (creado) o se puede conectar un repositorio remoto al ordenador creando así un repositorio local con el contenido existente en el repositorio remoto.

Para conectar un repo local con uno remoto, antes tenemos que crear un repositorio remoto. Dicho repo se quedará vacío, ya que va a tener los archivos del repo local al conectarlos. Para hacer esto se escribe el siguiente comando en el bash:

```
git remote add origin <URL del repositorio remoto>

# origin es el nombre por el que git identifica al repo remoto, se usa ese nombre
# por convención.
```

En el caso contrario se utilizará el siguiente comando, y esto se realizará unicamente la primera vez que "bajemos" el repo.

```
git clone <URL del repositorio remoto>
```

¿Cuál es la situación?

Para contestar a esta pregunta, existe un comando. Dicho comando nos muestra tres cosas:

- En que rama nos encontramos.
- Los archivos que han sido modificados.
- De los archivos que han sido modificados, cuales están preparados para hacer un commit.

El comando descrito es el siguiente:

```
git status
```

¿Cómo le cominicamos a Git que un archivo está preparado para hacer el commit?

Para comunicarle a Git que archivos están listos para ser commiteados, antes se le tiene que dar a guardar (si está guardado, git sabe que se ha modificado) y emplear el siguiente comando:

```
git add <Nombre del archivo>
```

Commit.

Ya sabiendo lo que es un commit, el comando que hará uno con los archivos que anteriormente se hayan añadido es el siguiente:

```
git commit -m "<Título del commit>"

# -m se pone para poner un mensaje, en este caso es el título del commit.
```

En el caso de que se haya realizado un commit, el cual es erroneo, se puede revertir empleando el siguiente comando:

```
git revert <Primera parte del hash>

# Con primera parte del hash, se pueden emplear los caracteres que hagan único
# el hash, deberían de bastar 6 o 7, pero se puede utilizar entero.

git revert HEAD

# HEAD significa el último commit realizado.
```

¿Cómo puedo saber el hash de un commit?

Para poder saber el hash de un commit, se puede acceder al historial de cambios en el proyecto. Esto se hace mediante el siguiente comando:

```
git log
```

Push.

Ya sabiendo lo que es un push, para hacer uno con los nuevos commits, se debe de utilizar el siguiente comando:

```
git push origin <Nombre de la rama a la que queremos hacer el push>
```

Pull.

Ya sabiendo lo que es un pull, el comando que se debe emplear para hacer uno es el siguiente:

```
git push origin <Nombre de la rama a la que queremos hacer el pull>
```

¿Cómo crear una rama?

Para crear una nueva rama, la cual se conecta a la rama en la que estamos situados, se utilizará el siguiente comando:

```
git branch <Nombre que queremos ponerle a la rama>
```

¿Cómo me puedo cambiar entre rama?

Para cambiar de una rama a otra, se debe de usar el siguiente comando:

```
git switch <Nombre de la rama a la que queremos cambiar>
```

¿Cómo puedo unir una rama con la rama de la que sale?

Para poder unir una rama con la rama de la que sale, primero debemos de encontrarnos en la rama de la que sale. Por ejemplo, si una ramma llamada "Rama 1" sale de main, para poder unir dicha rama con main, debemos de encontrarnos en main. Una vez estemos en la rama de la que sale, debemos de utilizar el siguiente comando:

```
git merge <Nombre de la rama que queremos unir>
```

¿Cómo puedo eliminar una rama?

Para poder eliminar una rama, se debe de emplear el siguiente comando:

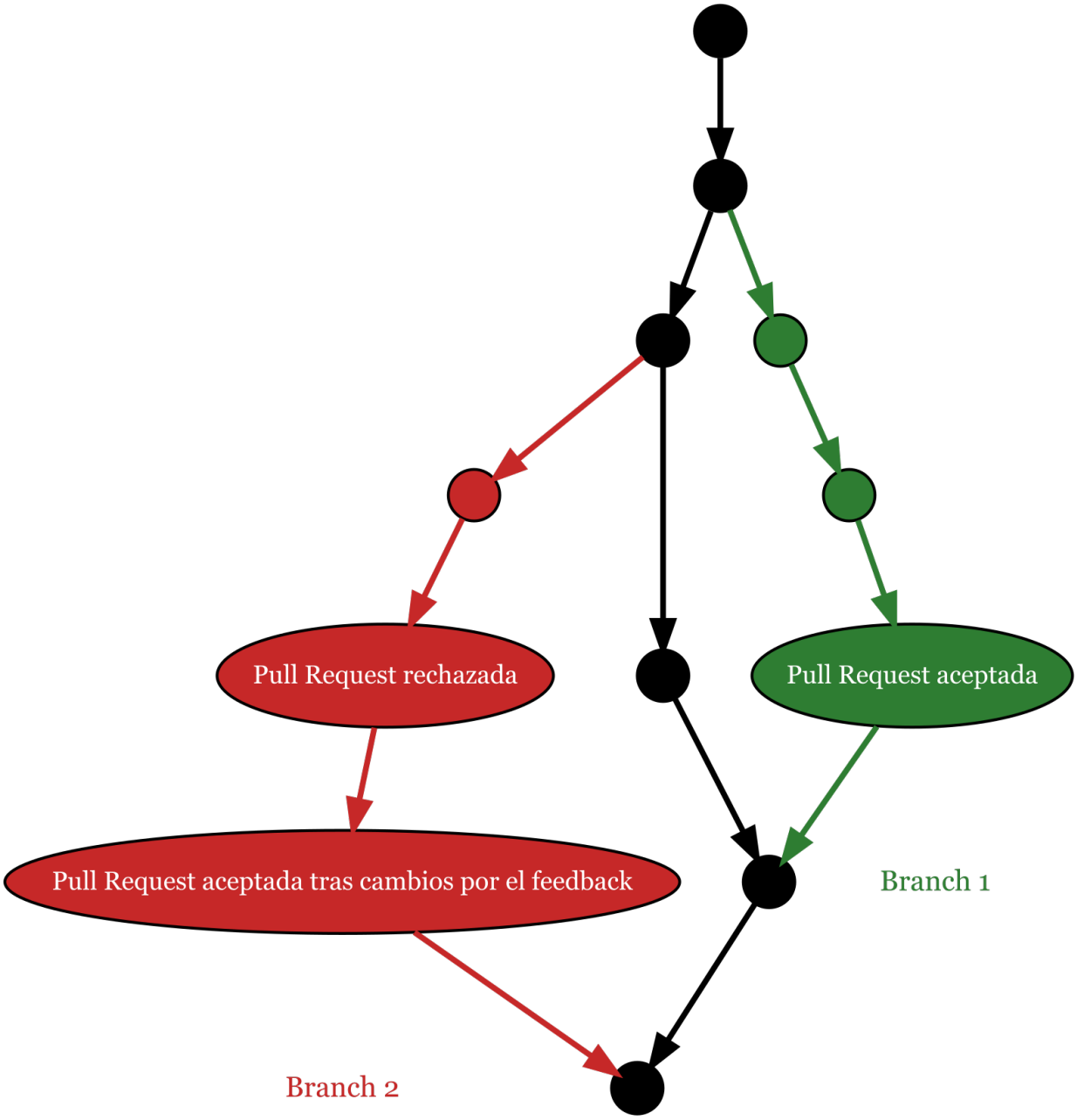
```
git branch -d <Nombre de la rama que queremos eliminar>

# -d se emplea cuando la rama ya está mergeada, si no lo está, no se podrá borrar

# Para forzar la eliminación de la rama, se debe de cambiar -d por -D
```

2.3. Servicio de repositorio remoto: GitHub.

GitHub es una plataforma en la nube que permite guardar y compartir proyectos que usan Git. Sirve para colaborar en equipo, revisar código y mantener un historial de versiones online de tus repositorios.

La parte más atractiva de GitHub, a la hora de compartir proyectos o de realizar tus propios proyectos son las Pull Request. Una Pull Request es una petición para que tus cambios se unan a la rama principal de un proyecto. Sirve para que otros revisen y aprueben tu código antes de fusionarlo.

Para entender el funcionamiento de las Pull Request, es más sencillo con la siguiente imagen.



Además de las Pull Request, GitHub ofrece otras funcionalidades. Una interesante puede ser realizar un Fork sobre un repositorio, que pertenezca a otro usuario. Realizar un Fork sobre un repo de otra persona es como hacer un copia y pega de su repositorio en los tuyos, y todos los cambios que se hagan, se hacen sobre tu copia sin afectar al repo original. A simple vista parece la función de clonar, pero es diferente. Al clonar el

repositorio los cambios se mandarían por una PR al dueño del repo.

3. Linux.

Linux es un sistema operativo libre y de código abierto. Es un sistema estable, rápido y completamente personalizable.

La importancia de aprender sobre linux radica en que es el sistema base en ciencia de datos e IA. Y además los servidores en la nube funcionan con linux.

4. Python.

Python es un lenguaje de programación considerado de alto nivel. Es muy popular y usado por los siguientes motivos:

- Es facil de aprender y leer al ser parecido al lenguaje humano (el inglés).
- Es multipropósito, sirve tanto para desarrollo web, como para análisis de datos, o incluso para videojuegos, entre otros.
- Al estar muy extendido existe una gran comunidad que lo emplea y una gran cantidad de librerías.

4.1. Mostrar por consola.

Para poder mostrar por consola el argumento que queramos, se deberá de emplear el siguiente comando:

```
print()

# Entre paréntesis se introducirá el argumento que queramos mostrar, este puede
# ser una variable predefinida, una cadena de texto, entre otras opciones.
```

Por ejemplo, mostraremos Hola, Mundo por consola.

```
print("Hola, Mundo")
```

```
## Hola, Mundo
```

4.2. Variables.

Una variable es un dato en forma de texto, de forma numérica u otro tipo, al que se le asigna un nombre.

Una variable se declarará del siguiente modo:

```
<Nombre de la variable> : <Tipo de dato> = <Dato>
```

En Python, no es necesario asignar el tipo de dato, pues se asigna solo. Sin poner el tipo de dato, se escribiría del siguiente modo:

```
<Nombre de la variable> = <Dato>
```

Para poder nombrar variables se deben tener varías reglas en cuenta:

- No pueden empezar por números, pero si pueden contenerlos.
- No pueden contener ni espacios ni guiones. Aunque si que pueden contener:
  - Letras, teniendo encuenta que Python distingue entre mayusculas y minusculas.
  - Guiones bajos (_)
  - Números
- No se pueden usar la palabras reservadas. La lista de las palabras reservadas se pueden obetener del siguiente modo:

```
import keyword

# Se importa este módulo (se verá más tarde) donde esta la lista de las
# palabras reservadas.

# Para mostrar la lista, y que se puedan ver todas las palábras, se usará un
# bucle, que se explicará más tarde.

for palabraReservada in keyword.kwlist:
  print(palabraReservada)
```

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

4.3. Tipos de datos.

En Python existen diferentes tipos de datos, los cuales están recogidos en la siguiente tabla.

Para saber que tipo de dato es un dato, se puede usar el siguiente comando:

Categoría	Tipo	Descripción
Numérico	int	Enteros (1, 2, 3)
	float	Reales con decimales (3.14)
	complex	Números complejos (3 + 2i)
Secuencia	str	Cadenas de texto
	list	Listas mutables
	tuple	Tuplas inmutables
Conjunto	set	Conjuntos mutables (sin duplicados)
	frozenset	Conjuntos inmutables
Mapeo	dict	Diccionarios (clave:valor)
Booleano	bool	Valores lógicos True/False
Nulo	NoneType	Ausencia de valor

```
type()

# Entreparéntesis, se meterá o bien directamente el dato o bien el nombre que
# se le haya asignado (variable).
```

Dejo varios ejemplos para que se aprecie su funcionamiento.

```
print(type(6))
```

```
## <class 'int'>
```

```
print(type('Hola'))
```

```
## <class 'str'>
```

```
print(type(True))
```

```
## <class 'bool'>
```

4.3.1. Tipos numéricos.

Como se ha observado en la tabla del inicio, dentro de los tipos de datos numéricos encontramos tres tipos diferentes. Los cuales son los siguientes:

- Enteros (int)
- Reales con decimales (float)
- Números complejos (complex)

Cuando utilizamos este tipo de dato, no se pone entre comillas.

Los numeros complejos (a + bi) se escribirán de la siguiente forma (a + bj), o incluso se pueden escribir con el siguiente comando:

```
complex(a, b)

# Siendo a la parte real y b la parte imaginaria.
```

A modo de prueba se le asignará un nombre a una varible con cada uno de los tipos de datos numéricos.

```
entero = 1
flotante = 1.1
complejo = 1+2j

# Para comprobar si se han escrito bien los datos, usaremos la funcion type()

print(type(entero))
```

```
## <class 'int'>
```

```
print(type(flotante))
```

```
## <class 'float'>
```

```
print(type(complejo))
```

```
## <class 'complex'>
```

Los tipos de datos numéricos se usan principalmente para realizar operaciones numéricas entre otras cosas.

4.3.2. Tipos secuenciales.

Como se ha observado en la tabla del inicio, dentro de los tipos de datos secuenciales encontramos tres tipos diferentes. Los cuales son los siguientes:

- Cadenas de texto (str)
- Listas mutables (list)
- Tuplas inmutables (tuple)

4.3.2.1. Cadena de texto o strig (str). Las cadenas de texto pueden contener letras, números, signos, etc. Estas se definen entre comillas, sirviendo tanto las comillas dobles como las simples.

Además se pueden introducir otras variables en una cadena de texto, para ello se puede usar la f, la cual se emplea de la siguiente manera:

```
f".....{<Aqui se pone el nombre de la variable>}................."
```

Por ejemplo, aquí tienes el resultado de emplear la f para juntar una cadena de texto con cualquier otra variable.

```
print(f"El número entero de antes es {entero}, mientras que el complejo es {complejo}")
```

```
## El número entero de antes es 1, mientras que el complejo es (1+2j)
```

A modo de prueba se le asignara a una variable un dato de tipo texto, para comprobar como sería y poder así también comprobar como funcionan algunos métodos de este tipo de dato.

```
texto = "hola"

texto2 = 'adiós'

# Para comprobar si se han escrito bien los datos, usaremos la funcion type()

print(type(texto))
```

```
## <class 'str'>
```

```
print(type(texto2))
```

```
## <class 'str'>
```

Para poder sacar una letra o un rango de caracteres de un dato de tipo texto, se usa el siguiente comando:

```
<Nombre de la variable>[<Aquí se pone la posición o rango que queramos extraer>]

# 0 es la primera posición,
# Con -1 tendríamos la última posición.
# El rango compredido entre la posicion X y la Y, se escribiría X:Y.
```

A continuación habrá una demostración de esto.

```
print(texto[0])
```

```
## h
```

```
print(texto[-1])
```

```
## a
```

```
print(texto[2:3])
```

```
## l
```

Además de extraer el caractér de la posición que queramos de una str, también podemos realizar otras opciones. A las funciones que realizan acciones sobre algún tipo de dato se llaman métodos. A continuación vendrán los métodos más empleados de los str, junto con una prueba de cada uno de ellos.

```
len(<Nombre de la variable>)

# Dice la longitud del texto.
```

```
print(len(texto))
```

```
## 4
```

```
<Nombre de la variable>.upper()

# Devuelve todo el texto en mayúsculas
```

```
print(texto.upper())
```

```
## HOLA
```

```
<Nombre de la variable>.lower()

# Devuelve todo el texto en minusculas
```

```
print(texto.lower())
```

```
## hola
```

```
<Nombre de la variable>.capitalize()

# Devuelve la primera letra del texto en mayúsculas
```

```
print(texto.capitalize())
```

```
## Hola
```

```
<Nombre de la variable>.title()

# Devuelve en mayúsculas la primera letra de cada palabra

# Para esta prueba se crea una variable con varias palabras

texto3 = "estoy programando con python"

print(texto3.title())
```

```
## Estoy Programando Con Python
```

```
<Nombre de la variable>.count("<argumento>")

# Devuelve cuantas veces se repite el argumento dentro del texto
```

```
print(texto3.count("o"))
```

```
## 5
```

```
<Nombre de la variable>.startswith("<argumento>")

# Devuelve True o False dependiendo de si el texto empieza con el argumento
```

```
print(texto3.startswith("estoy"))
```

```
## True
```

```
<Nombre de la variable>.endswith("<argumento>")

# Devuelve True o False dependiendo de si el texto termina con el argumento
```

```
print(texto3.endswith("estoy"))
```

```
## False
```

4.3.2.2. Listas mutables (list) Las listas son colecciones ordenadas y modificables (mutables) de elementos. Las listas no continen claves y dentro de sus elementos no tienen porque ser todos del mismo tipo de dato. Es decir, que dentro de una lista se pueden encontrar datos numéricos, de tipo texto, o incluso otra lista.

Las listas se definen de la siguiente manera:

```
<Nombre de la lista> = [<Elemento>, <Elemento>, <Elemento>, ...]
```

Para comprobar que las listas se definen como anteriormente se ha comentado, se creará una. Así también se podrá emplear para probar sus métodos.

```
lista = ["Hola", "Adiós", 1, 1+2j, ["día", "noche"]]

# Para comprobar si se han escrito bien los datos, usaremos la funcion type()

print(type(lista))
```

```
## <class 'list'>
```

Al igual que los str y todos los tipos de datos, existen métodos útiles para el empleo de los datos. En el caso de las listas, los métodos más empleados son los que están a continuación, junto con una prueba de cada uno con la lista creada anteriormente.

Para comenzar, en las cadenas de texto se puede acceder al carácter, que se encuentra en la posición que queramos. Pues en las listas podemos acceder al elemento que se encuentre en la posición que queramos. Además si un elmento es de tipo texto, tambien podemos acceder al caracter que este en la posición que queramos la mismo tiempo.

```
<Nombre de la lista>[<Posición>]

# Las posiciones funcionan igual que en las cadenas de texto

<Nombre de la lista>[<Posición en la lista>][<Posición en el texto>]
```

```
# Para hacer esto, el elemento de la lista debe ser de tipo str. Se puede saber
# de la siguiente manera.

type(<Nombre de la lista>[<Posición en la lista>])
```

```
print(lista[0])
```

```
## Hola
```

```
print(lista[-1])
```

```
## ['día', 'noche']
```

```
print(type(lista[0]))
```

```
## <class 'str'>
```

```
print(lista[1][0])
```

```
## A
```

A continuación estarán los demás métodos, sin estar todos, puesto que existen muchos. En el caso de querer saber más, hay que recurrir a la documentación.

```
<Nombre de la lista>.append(<Elemento>)

# Añade el elemento al final de la lista
```

```
lista.append(5)

print(lista)
```

```
## ['Hola', 'Adiós', 1, (1+2j), ['día', 'noche'], 5]
```

```
<Nombre de la lista>.insert(<Posición>, <Elemento>)

# Añade el elemento en la posición que queramos. Si la posición es central,
# Los elementos que vayan después, se desplazarán una posición.
```

```
lista.insert(3,"Posición 3")

print(lista)
```

```
## ['Hola', 'Adiós', 1, 'Posición 3', (1+2j), ['día', 'noche'], 5]
```

```
<Nombre de la lista>.remove(<Elemento>)

# Elimina el elemento de la lista con ese nombre
```

```
lista.remove("Hola")

print(lista)
```

```
## ['Adiós', 1, 'Posición 3', (1+2j), ['día', 'noche'], 5]
```

```
<Nombre de la lista>.pop(<Posición>)

# Elimina el elemento de la lista que se encuentre en esa posición

# También se puede emplear el siguiente:

del Nombre_de_la_lista[<Posición>]
```

```
lista.pop(-1)
```

```
## 5
```

```
print(lista)
```

```
## ['Adiós', 1, 'Posición 3', (1+2j), ['día', 'noche']]
```

```
del lista[0]

print(lista)
```

```
## [1, 'Posición 3', (1+2j), ['día', 'noche']]
```

```
len(<Nombre de la lista>)

# Cuenta el número de elementos dentro de una lista.
```

```
print(len(lista))
```

```
## 4
```

4.3.2.3. Tuplas inmutables (tuple). Las tuplas son como las listas, pero cuyo contenido no se puede modificar (es inmutable).

Las tuplas se definen de manera similar a las listas, lo único que cambia es que en la lista se emplean [], y en las tuplas (). Se definirían del siguiente modo:

```
<Nombre de la tupla> = (<Elemento>, <Elemento>, <Elemento>, ...)
```

Para comprobar que las tuplas se definen como se ha comentado, se generará una tupla. Y además se podrá emplear para probar sus métodos.

```
tupla = ["Hola", "Adiós", 1, 1+2j, ["día", "noche"]]

# Para comprobar si se han escrito bien los datos, usaremos la funcion type()

print(type(tupla))
```

```
## <class 'list'>
```

En el caso de las tuplas, al ser inmutables, no existen métodos que las modifiquen. Aunque si que existen algunos métodos para realizar operaciones con ellas, entre otras cosas.

La forma de acceder a cualquier elemento de la tupla es la misma forma que en las listas.

```
<Nombre de la tupla>.count(<Elemento>)

# Devuelve el número de elementos que coincide con el elemento introducido.
```

```
print(tupla.count(1))
```

```
## 1
```

```
len(<Nombre de la tupla>)

# Devuelve el número de elementos que hay en la lista
```

```
print(len(tupla))
```

```
## 5
```

Como se ha comentado anteriormente, hay métodos que permiten realizar operaciones matemáticas con los elementos de las tuplas. Para comprobar algunos, se generará una tupla con elementos de tipo numérico.

```
tuplaNumerica = (1, 2, 3, 4, 5)
```

```
max(<Nombre de la tupla>)

# Devuelve el número más alto que se encuentre dento de la tupla.
```

```
print(max(tuplaNumerica))
```

```
## 5
```

```
sum(<Nombre de la tupla>)

# Devuelve la suma de todos los elemntos del interior de la tupla.
```

```
print(sum(tuplaNumerica))
```

```
## 15
```

```
sorted(<Nombre de la tupla>)

# Devuelve la tupla de manera ordenada, esta opcion tambien existe en las listas,
# y se pueden ordenar siguiendo alguna norma. Su funcionamiento general es:

sorted(<Nombre variable>, key = <regla de ordenacion, ej. len>, reverse = True)

# Reverse = True, saca la lista invertida a como saldría sin ponerla.
```

```
print(sorted(tuplaNumerica))
```

```
## [1, 2, 3, 4, 5]
```

Para probar el método sorted, tambien con una tupla con unicamente texto, se creará una nueva tupla.

```
tuplaTexto = ("Hola", "Adiós", "uno", "esternocleidomastoideo")

print(sorted(tuplaTexto, key=len)) # Ordena por longuitud
```

```
## ['uno', 'Hola', 'Adiós', 'esternocleidomastoideo']
```

```
print(sorted(tuplaTexto, key=len, reverse=True)) # Ordenada de más largo a menos
```

```
## ['esternocleidomastoideo', 'Adiós', 'Hola', 'uno']
```

4.3.3. Tipos de conjuntos.

Como se ha observado en la tabla del inicio, dentro de los tipos de datos que son conjuntos encontramos dos tipos diferentes. Los cuales son los siguientes:

- Conjuntos mutables (set)
- Conjuntos inmutables (frozenset)

Tanto los conjuntos mutables como los inmutables, son similares al funcionamiento de las listas y las tuplas. La única diferencia es que en los conjunto no se pueden repetir elementos. A efectos prácticos, los conjuntos mutables funcionan como las listas y los conjuntos inmutables funcionan como las tublas.

La manera de definir los conjuntos mutables es la siguiente:

```
<Nombre del conjunto mutable> = [<Elemento>, <Elemento>, <Elemento>, ...]

<Nombre del conjunto mutable> = set(Nombre_del_cojunto_mutable)

# Tambiém se puede hacer sin generar una lista anteriormente

<Nombre del conjunto mutable> = set([<Elemento>, <Elemento>, <Elemento>, ...])
```

Para comprobar que es así, se genera un conjunto mutable. Teniendo en cuenta que los métodos de las listas son los mismos que el de los conjuntos mutables, no se probará ninguno, pues el funcionamiento es el mismo.

```
conjuntoMutable = [1,2,2,2,3,4,5,5,6]

conjuntoMutable = set(conjuntoMutable)

print(conjuntoMutable)
```

```
## {1, 2, 3, 4, 5, 6}
```

```
# Para comprobar si se han escrito bien los datos, usaremos la funcion type()

print(type(conjuntoMutable))
```

```
## <class 'set'>
```

La manera de definir los conjuntos mutables es la siguiente:

```
<Nombre del conjunto inmutable> = [<Elemento>, <Elemento>, <Elemento>, ...]

<Nombre del conjunto inmutable> = frozenset(Nombre_del_cojunto_inmutable)

# Tambiém se puede hacer sin generar una lista anteriormente

<Nombre del conjunto inmutable> = frozenset([<Elemento>, <Elemento>, <Elemento>, ...])
```

Para comprobar que es así, se genera un conjunto inmutable. Teniendo en cuenta que los métodos de las tuplas son los mismos que el de los conjuntos inmutables, no se probará ninguno, pues el funcionamiento es el mismo.

```
conjuntoInmutable = ['A','A','B','C','D','D']

conjuntoInmutable = frozenset(conjuntoInmutable)

print(conjuntoInmutable)
```

```
## frozenset({'C', 'B', 'A', 'D'})
```

```
# Para comprobar si se han escrito bien los datos, usaremos la funcion type()

print(type(conjuntoInmutable))
```

```
## <class 'frozenset'>
```

4.3.4. Diccionarios (dict).

Los diccionarios son colecciones de pares **clave:valor**. Los pares clave:valor son como una caja con etiquetas, cada etiqueta (clave) tiene algo guardado dentro (valor). El valor puede ser cualquier tipo de dato, incluyendo hasta a los mismos diccionario, mientras que las claves deben de ese de algún tipo de dato inmutable, sindo el más comun las str

Los diccionarios se definen del siguiente modo:

```
<Nombre del diccionario> = {<clave>:<valor>, <clave>:<valor>, ...}
```

A modo de prueba, para comprobar como se generan los diccionarios y además poder así probar los métodos de los diccionarios con ejemplos.

```
diccionario = {
  "nombre":"Antonio",
  "edad":55,
  "dirección":{
    "calle":"Calle de la música",
    "número":12
    },
  "colores_favoritos":["azul", "verde", "naranja"]
  }

# Para comprobar que lo hemos crado correctamente, probamos con type()

print(type(diccionario))
```

```
## <class 'dict'>
```

```
# Para comprobar los resultados de los métodos y como se vería el diccionario
# completo, se empleará el siguiente bucle (mas tarde se explicarán)

for clave, valor in diccionario.items():
  print(f"{clave}: {valor}")
```

```
## nombre: Antonio
## edad: 55
## dirección: {'calle': 'Calle de la música', 'número': 12}
## colores_favoritos: ['azul', 'verde', 'naranja']
```

Para poder acceder a cualquier valor que se encuentre dentro del diccionario, se debe emplear el siguiente comando:

```
<Nombre del diccionario>[<clave>]
```

```
print(diccionario["colores_favoritos"])
```

```
## ['azul', 'verde', 'naranja']
```

Al igual que todos los tipos de datos, los diccionarios tienen sus métodos. Algunos de los métodos más utilizados con los diccionarios son los que estarán a continución.

```
<Nombre del diccionario>.clear()

# Elimiina todos los elementos del diccionario
```

```
diccionario.clear()

# Se usa el condicional if para poder saber si de verdad viene vacío.

if len(diccionario) == 0:
  print("El diccionario está vacío.")
else:
  for clave, valor in diccionario.items():
    print(f"{clave}: {valor}")
```

```
## El diccionario está vacío.
```

Para poder comprobar los demás métodos, se generará nuevo contenido en el diccionario.

```
diccionario = {
  "nombre":"Antonio",
  "edad":55,
  "dirección":{
    "calle":"Calle de la música",
    "número":12
    },
  "colores_favoritos":["azul", "verde", "naranja"]
  }
```

```
<Nombre del diccionario>.item()

# Devuelve la clave:valor de todo el diccionario en forma de tupla.

# Este no se prueba, puesto que se usa en el bucle para mostrar los datos.
```

```
<Nombre del diccionario>.get(<clave>)

# Devuelve el valor que corresponde a la clave escrita.
```

```
print(diccionario.get("nombre"))
```

```
## Antonio
```

```
<Nombre del diccionario>.keys()

# Devuelve todas las claves del diccionario.
```

```
for clave in diccionario.keys():
  print(clave)
```

```
## nombre
## edad
## dirección
## colores_favoritos
```

```
<Nombre del diccionario>.values()

# Devuelve todos los valores del diccionario.
```

```
for valor in diccionario.values():
  print(valor)
```

```
## Antonio
## 55
## {'calle': 'Calle de la música', 'número': 12}
## ['azul', 'verde', 'naranja']
```

```
<Nombre del diccionario>.pop(<clave>)

# Elimina la calve y su valor asignado. También serviría:

del <Nombre del diccionario>[<clave>]
```

```
diccionario.pop("nombre")
```

```
## 'Antonio'
```

```
del diccionario["edad"]

for clave, valor in diccionario.items():
  print(f"{clave}: {valor}")
```

```
## dirección: {'calle': 'Calle de la música', 'número': 12}
## colores_favoritos: ['azul', 'verde', 'naranja']
```

```
<Nombre del diccionario>[<clave>] = <valor>

# Crea una nueva clave con su valor asignado.
```

```
diccionario[123] = "clave de forma numérica"

for clave, valor in diccionario.items():
  print(f"{clave}: {valor}")
```

```
## dirección: {'calle': 'Calle de la música', 'número': 12}
## colores_favoritos: ['azul', 'verde', 'naranja']
## 123: clave de forma numérica
```

4.3.5. Tipos booleanos (bool).

Los datos de tipo booleano, son de tipo lógico y expresan True o False, en función de si la afirmación es cierta (True) o falsa (False). Funciona principalmente con operadores, pero tambíen se puede definir una variable de este tipo.

La definición de una variable de este tipo se hará del siguiente modo:

```
<Nombre de la variable booleana> = True o False
```

Para comprobarlo, se crea una variable de tipo booleano.

```
booleano = True

# Para comprobar si se han escrito bien los datos, usaremos la funcion type().

print(type(booleano))
```

```
## <class 'bool'>
```

4.4. Operadores.

Un operador es un símbolo o palabra que indica una operación que se aplica a uno o más valores llamados operandos.

4.4.1. Operadores aritméticos.

Los operadores aritméticos sirven para realizar cáculos matemáticos elementales entre variables numéricas. Los operadores aritméticos son los siguientes:

- $+ \longrightarrow$ Suma.
- $- \longrightarrow$ Resta.
- $* \longrightarrow$ Multiplicación.
- $/ \longrightarrow$ División.
- $// \longrightarrow$ División entera (redondea a la baja).
- $\% \longrightarrow$ Módulo o resto de la división.
- $** \longrightarrow$ Potencia.

Se cearán dos variables numéricas para poder realizar las operaciones y comprobar como funcionan y sí realizan las operaciones anteriormente mencionadas.

```
a = 2
b = 9

print(a + b)
```

```
## 11
```

```
print(a - b)
```

```
## -7
```

```
print(a * b)
```

```
## 18
```

```
print(a / b)
```

```
## 0.2222222222222222
```

```
print(a // b)
```

```
## 0
```

```
print(a % b)
```

```
## 2
```

```
print(a ** b)
```

```
## 512
```

Para comprobar si los operadores aritméticos funcionan también con datos de tipo str, se van a crear dos variables de este tipo. Al mismo tiempo que se prueba con el método (try except), que se explicará más adelante. Los únicos operadores que podrían funcionar aparentemente con este tipo de datos, pueden llegar a ser la suma y la resta, por eso únicamente se probarán estos.

```
a = "Hola"
b = "Adios"

try:
  a + b
  print(a + b)
except Exception as e:
  print("Error", e)
```

```
## 'HolaAdios'
## HolaAdios
```

```
try:
  a - b
  print(a - b)
except Exception as e:
  print("Error", e)
```

```
## Error unsupported operand type(s) for -: 'str' and 'str'
```

```
try:
  a - "a"
  print(a - "a")
except Exception as e:
  print("Error", e)
```

```
## Error unsupported operand type(s) for -: 'str' and 'str'
```

4.4.2. Operadores de asignación.

Los operadores de asignación asignan valores a variables y, a veces, realizan operaciones al mismo tiempo. Los operadores de asignación son los siguientes:

- $=$ $\longrightarrow$ Asigna un valor.
- $<$ *Operador aritmtivo* $>=$ $\longrightarrow$ Realiza una operación sobre sí mismo.

Para comprobar como se puede realizar una operación sobre una variable predefinida, se crea una nueva variable y se prueba esto.

```
a = 999
a+=1

print(a)
```

```
## 1000
```

4.4.3. Operadores de comparación.

Los operaciones de coparación permiten comparar valores y devuelven un resultado booleano (True o False). Los operadores de comparación son los siguientes:

- $==$ $\longrightarrow$ Igual que.
- $!=$ $\longrightarrow$ Distinto de.
- $>$ $\longrightarrow$ Mayor que.
- $<$ $\longrightarrow$ Menor que.
- $>=$ $\longrightarrow$ Mayor o igual que.
- $<=$ $\longrightarrow$ Menor o igual que.

Para comprobar como funcionan, se crearán una serie de variables de tipo numérico para poder ver como sería el resultado.

```
a = 12
b = 5.25

print(a == b)
```

```
## False
```

```
print(a != b)
```

```
## True
```

```
print(a > b)
```

```
## True
```

```
print(a < b)
```

```
## False
```

```
print(a >= b)
```

```
## True
```

```
print(a <= b)
```

```
## False
```

El comportamiento de los operadores de comparación con variables de tipo str, es diferente y algo menos intuitivo. El lenguaje Python interpreta las comparaciones con las siguientes reglas:

- Al emplear == o !=, tiene en cuenta caractér por caractér, teniendo en cuenta las mayusculas y los espacios a la hora de decidir si es verdadera o falsa la afirmación.
- Al emplear los demás operadores (>, <, >=, <=), tiene en cuenta otros criterios, los cuales son los siguientes:
  - Tiene en cuenta primero el numero de caracteres, y en el caso de que el número de caracteres sea el mismo, tiene en cuenta el orden alfabético (dado por Unicode).

Para comporobar esto, se crearán una serie de variables y se emplearán una serie de operadores de comparación.

```
a = "abcd"
b = "abc"
c = "zyx"
e = "ABC"

print(a == b)
```

```
## False
```

```
print(b != e)
```

```
## True
```

```
print(a > b)
```

```
## True
```

```
print(b > c)
```

```
## False
```

4.4.4. Operadores lógicos.

Los operadores lógicos se usan para combinar condiciones. Devolviendo un valor booleando dependiendo de los argumentos empleados. Los operadores lógicos son los siguientes:

- and ⟶ Y ⟶ Verdadero si las condiciones combinadas son verdaderas.
- or ⟶ O ⟶ Verdader si una de las condiciones combinadas es verdadera.
- not ⟶ NO ⟶ Invierte el valor lógico.

Para comprobar el funcionamiento de los operadores lógicos, se emplearán una serie de variables que se van a crear a continuación. Como el funcionamiento de los operadores lógicos funcionan en conjunto con los operadores de comparación, solo se comprobarán con variables de tipo numérico.

```
a = 5
b = 10
c = 80

print(type(a) == int and a < b)
```

```
## True
```

```
print(a > b or b < c)
```

```
## True
```

```
print(a < b or b < c)
```

```
## True
```

```
print(not a < b)
```

```
## False
```

4.4.5. Operadores de identidad.

Los operadores de identidad comparan si dos objetos son el mismo, no solo si tienen el mismo valor. Devolviendo un valor booleano. Los operadores de identidad son los siguientes.

- is ⟶ Devuelve True si son el mismo objeto. La diferencia con el operador de comparación == es que no compara el valor, si tienen la misma referencia o no.
- is not ⟶ Devuelve True si no son el mismo objeto. La diferencia con el operador de comparación != es que no compara el valor, si tienen la misma referencia o no.

Para comprobar esto, puesto que es algo abstracto, se realizarán varios ejemplos.

```
a = [1, 2]
b = [1, 2]
c = a

print(a is b)
```

```
## False
```

```
print(a is c)
```

```
## True
```

```
print(a is not b)
```

```
## True
```

4.4.6. Operadores de pertenencia.

Los operadores de pertenenecia comprueban si un elemento está dentro de una secuencia, ya sea un texto (str), una lista u otra secuencia, devolviendo un valor booleano dependiendo de cuales sean los argumentos empleados. Los tipos de operadores de pertenencia son los siguientes:

- in ⟶ Es verdadero cuano el elemento esta dentro del otro.
- not in ⟶ Es verdadero cuando el elemento no esta dentro del otro.

Para comprobar su funcionamiento se crearán varios ejemplos de cada uno.

```
a = [1, 2]
b = "Hola"

print(1 in a)
```

```
## True
```

```
print("a" in b)
```

```
## True
```

```
print("h" not in b) # Para demostrar que python diferencia las mayúsculas.
```

```
## True
```

```
print(" " in b) # Para demostrar que python diferencia los espacios en blanco
```

```
## False
```

4.5. Condicionales.

Los condicionales son instrucciones que ejecutan un bloque de código solo si una condición es verdadera (True).

Si la condición es falsa (False), puede ejecutar otra parte diferente o no hacer nada, dependiendo de las instrucciones que le pongamos al código.

Existen dos tipos de condicionales, los cuales son los siguientes:

- El condicional **if-elif-else**.
- El condicional **match-case**.

4.5.1. IF-ELIF-ELSE.

El condicional if-elif-else, tiene el siguiente funcionamiento:

```
if <condición>:
  <código que se ejecutará cuando la condición sea cierta>
elif <condición>:
  <código que se ejecutará cuando la condicion sea verdadera y la primera sea falsa>
      .
      .
      . # Esto refleja que se pueden poner tantos elif como se quieran
      .
      .
elif <condición>:
  <código que se ejecutará cuando la condicion sea verdadera y las anteriores falsas>
else:
  <código que se ejecutará cuando todas las condiciones sean falsas>
```

El funcionamiento con palabras es:

1. Si pasa esto, haz esto. Si no pasa, pasa al siguiente paso.
2. Si pasa esto, haz esto. Si no pasa, pasa al siguiente paso.
3. Si no ha ocurrido nada de los que se ha dicho anteriormente, haz esto.

Para ver como funcionaría el condicional se emplearán un par de ejemplos.

```
persona = {
  'nombre':'Pepe',
  'edad':18
}

if persona['edad'] >= 18:
  print(f'{persona['nombre']} es mayor de edad')
else:
  print(f'{persona['nombre']} es menor de edad')
```

```
## Pepe es mayor de edad
```

```
persona = {
  'nombre':'Luisa',
  'nota':4.9
}

if persona['nota'] >= 9:
    print(f"{persona['nombre']} tiene un sobresaliente")
elif persona['nota'] >= 7:
    print(f"{persona['nombre']} tiene un notable")
elif persona['nota'] >= 5:
    print(f"{persona['nombre']} tiene un aprobado")
else:
    print(f"{persona['nombre']} esta suspenso, tiene que repetir la prueba")
```

```
## Luisa esta suspenso, tiene que repetir la prueba
```

4.5.2. MATCH-CASE.

El condicional match-case, es menos usado, pero también es interesante conocer su funcionamiento. El funcionamiento es el siguiente:

```
match <variable predefinida>:
  case <valor>:
    <código que se ejecutará cuando la variable sea == al valor>
  case <valor>:
    <código que se ejecutará cuando la variable sea == al valor>
        .
        .
        . # Esto refleja que se puede poner tantas veces como se quiera case
        .
        .
  case _:
    <código que se ejecutará si nunguno de los casos anteriores ocurren>
```

El funcionamiento en palabras de este condicional es parecido al if-elif-else, pero difiere un poco. Sería del siguiente modo:

1. Si el valor es igual a esto, haz esto, si no es igual, pasa al siguiente.
2. Si el valor es igual a esto, haz esto, si no es igual, pasa al siguiente.
3. Si el valor no es igual a nada de lo que he puesto, haz esto.

Para ver como funcionaría el condicional se empleará algún ejemplo.

```
color = "verde"

match color:
  case "rojo":
    print("Detente")
  case "verde":
    print("Sigue")
  case "amarillo":
    print("Precaución")
  case _:
    print("Color desconocido")
```

```
## Sigue
```

4.6. Bucles.

Los bucles son estructuras que permiten repetir un bloque de código varias veces. En lugar de escribir lo mismo una y otra vez, el programa repite las instrucciones mientras se cumpla una condición o para cada elemento de la secuencia (una lista, por ejemplo).

Existen dos maneras de definir un bucle, siendo las siguientes:

- while ⟶ "mientras que …"
- for ⟶ "por cada uno de los elementos …"

4.6.1. WHILE.

El bucle while repetirá un bloque de código siempre que la condición puesta sea verdadera, o dicho de otra manera, repetirá el bloque de código hasta que la condición sea falsa.

El bucle while se configurará del siguiente modo:

```
while <condición>:
  <código que se ejecutará mientras que se cumpla la condición>
  # Pueden ser varias líneas de código.
```

El funcionamiento del bucle while en palabras es el siguiente:

- Mientras que ocurra esto, no pares de hacer esto.

Para ver como funciona el bucle while, a continuación se encuentran varios ejemplos.

```
contador = 0

while contador <= 5:
  print(f"Se está imprimiendo en consola el contador nº {contador}")
  contador += 1
```

```
## Se está imprimiendo en consola el contador nº 0
## Se está imprimiendo en consola el contador nº 1
## Se está imprimiendo en consola el contador nº 2
## Se está imprimiendo en consola el contador nº 3
## Se está imprimiendo en consola el contador nº 4
## Se está imprimiendo en consola el contador nº 5
```

4.6.2. FOR.

El bucle for repetirá un bloque de código por cada elemento de una secuencia, como una lista, una cadena, una tupla o un rango de números.

El bucle for se configurará del siguiente modo:

```
for <variable> in <secuencia>:
  <código que se ejecutará mientras que hayan elementos en la secuencia>
  # Pueden ser varias líneas de código.
```

El funcionamiento del bucle for en palabras es el siguiente:

- Por cada variable en la secuencia, haz esto.

Para ver como funciona el bucle for, a continuación se encuentran varios ejemplos.

```
palabras = ['hola', 'adios', 'árbol', 'nube']

for palabra in palabras:
  print(f"La palabra por la que vamos en el bucle es: {palabra}")
```

```
## La palabra por la que vamos en el bucle es: hola
## La palabra por la que vamos en el bucle es: adios
## La palabra por la que vamos en el bucle es: árbol
## La palabra por la que vamos en el bucle es: nube
```

4.6.3. Consideraciones para ambos bucles.

Dentro de ambos bucles existen algunas funciones útiles, como pueden ser las siguientes:

- **break** ⟶ Sirve para detenr el bucle completamente cuando se cumple una condición, se debe de emplear con un if.
- **continue** ⟶ Sirve para saltar una interacción (una vuelta del bucle) y continuar con la siguiente, se debe de emplear con un if.
- **else** ⟶ Es parecido al "else" del condicional if-elif-else, pero en este caso, se ejecuta siempre que termina el bucle. Si se interrumpe con un break, no saldría.
- **zip()** ⟶ Une dos o más listas emparejando sus elementos. Se puede emplear en ambos tipos de bucles, pero esta diseñado para emplearse en el bucle for.
- **range()** ⟶ Genera una secuencia de números que se puede recorrer en un bucle. Se puede emplear en ambos tipos de bucles, pero esta diseñado para emplearse en el bucle for.
- **enumerate()** ⟶ Permite acceder al índice y al valor de una secuencia al mismo tiempo, muy útil cuando necesitas saber en qué posición estás dentro del bucle. Se puede emplear en ambos tipos de bucles, pero esta diseñado para emplearse en el bucle for.
- **pass** ⟶ UIndica que no se ejecutará ninguna acción en ese punto del código. Se usa como marcador temporal o cuando la estructura del bucle necesita una instrucción obligatoria.

A continuación se presenta algún ejemplo para cada una de las funcionalidades. Las funcionalidades que están pensadas para el bucle for, solo se probarán con este, pues no vamos a ir contracorriente al usar una función para el bucle while cuando no esta pensada para el mismo.

BREAK

```
contador = 0

while contador <= 10:
  print(contador)
  contador += 1
  if contador == 3:
    break
```

```
## 0
## 1
## 2
```

```
numeros = [1, 2, 3, 4, 5, 6]

for numero in numeros:
  print(numero)
  if numero == 3:
    break
```

```
## 1
## 2
## 3
```

CONTINUE

```
contador = 0

while contador <= 10:
  print(contador)
  contador += 1
  if contador == 3:
    continue
```

```
## 0
## 1
## 2
## 3
## 4
## 5
## 6
## 7
## 8
## 9
## 10
```

```
numeros = [1, 2, 3, 4, 5, 6]

for numero in numeros:
  print(numero)
  if numero == 3:
    continue
```

```
## 1
## 2
## 3
## 4
## 5
## 6
```

ELSE

```
contador = 0

while contador <= 10:
  print(contador)
  contador += 1
else:
  print("Se ha terminado el bucle")
```

```
## 0
## 1
## 2
## 3
## 4
## 5
## 6
## 7
```

```
## 8
## 9
## 10
## Se ha terminado el bucle
```

```
numeros = [1, 2, 3, 4, 5, 6]

for numero in numeros:
  print(numero)
else:
  print("Se ha terminado el bucle")
```

```
## 1
## 2
## 3
## 4
## 5
## 6
## Se ha terminado el bucle
```

ZIP()

El funcionamiento general de esta función es la siguiente:

```
for <variable 1>, <variable 2> in zip(<secuencia 1>, <secuencia 2>):
  <código que se ejecutará mientras que hayan elementos en las secuencias>

# Se pueden poner tantas variables, y por tanto, secuencias como se quieran.
```

Por ejemplo:

```
nombres = ["Ana", "Luis"]
edades = [20, 25]

for nombre, edad in zip(nombres, edades):
  print(f"{nombre} tiene {edad} años")
```

```
## Ana tiene 20 años
## Luis tiene 25 años
```

En el caso de que las secuencias no tengan el mismo número de elementos, el bucle funcionará, y repetirá el bloque de código el número de veces que sea igual al número de elmentos que tiene la secuencia con menos de los mismos. Tal y como se puede observar a continuación:

```
nombres = ["Ana", "Luis"]
edades = [20, 25, 30]

for nombre, edad in zip(nombres, edades):
  print(f"{nombre} tiene {edad} años")
```

```
## Ana tiene 20 años
## Luis tiene 25 años
```

RANGE()

El funcionamiento general de esta función es la siguiente:

```
for <variable> in range(<número>, <número>):
  <código que se ejecutará mientras que queden números en el rango>
```

Por ejemplo:

```
for i in range(2, 6): # En todos los bucles se pone i por convención, pero
  print(i)            # personalmente prefiero poner el nombre de la variable.
```

```
## 2
## 3
## 4
## 5
```

ENUMERATE()

El funcionamiento general de esta función es la siguiente:

```
for <enumeración>, <variable> in enumerate(<secuencia>, start=<número>):
  <código que se ejecutará mientras que hayan elementos en la secuencia>

# En enumeración se puede poner lo que se quiera.
```

Por ejemplo:

```
frutas = ["manzana", "pera", "uva"]

for enumeracion, fruta in enumerate(frutas, start=1):
    print(enumeracion, fruta)
```

```
## 1 manzana
## 2 pera
## 3 uva
```

4.7. Funciones.

Las funciones son bloques de código reutilizable que ejecuta una tarea concreta. Para poder activar una función, hay que poner el nombre asignado y tras ello, unos paréntesis. Las funciones se confeccionan del siguiente modo:

```
def <Nombre de la función>(<parámetros>):
  <codigo a ejecutar por parte de la función (pueden ser mas filas)>
  return <lo que queremos que devuelva la función>

# No es obligatorio establecer parámetros, se puede hacer sin ellos.

# No es necesario escribir return, pero si que es necesario definir que va a ocurrir.
```

Algunos ejemplos de funciones se pueden encontrar a continuación, teniendo diferentes ejemplos de como se pueden redactar (con return o sin return), o incluso sin parametros.

```
def sumar(a, b):
  return a + b

print(sumar(2, 5))
```

```
## 7
```

```
def suma(a, b):
  print(a + b)

suma(5, 8)
```

```
## 13
```

```
def Sumar(a, b):
  return print(a + b)

Sumar(1, 2)
```

```
## 3
```

```
def imprimeCosas():
  print("Estoy escribiendo por consola este mensaje a partir de una función")

imprimeCosas()
```

```
## Estoy escribiendo por consola este mensaje a partir de una función
```

A la hora de trabajar con las funciones en Python, se debe de tener en cuenta la diferencia entre argumentos posicionales y no posicionales. Y también se debe de tener en cuenta que ambos se pueden combinar.

- Los argumentos posicionales son valores que se pasan a una función basándose en su orden o posición en la llamada a la función.
- Los argumentos no posicionales son valores que se pasan a una función especificando explícitamente el nombre del parámetro al que pertenecen.

A modo de prueba, para entender como funcionan los argumentos posicionales y los no posicionales, se emplearán algunos ejemplos.

```
def posicionales(a, b, c, d):
    print(f"{a}, {b}, {c}, {d}")

posicionales(1, 2, 3, 4)
```

```
## 1, 2, 3, 4
```

```python
def NOposicionales(a, b, c, d):
    print(f"{a}, {b}, {c}, {d}")

NOposicionales(d=4, b=2, a=1, c=3)
```

```
## 1, 2, 3, 4
```

```python
# Al combinarlos, primero posicionales y luego por nombre, sin repetir parámetros.
def combinados(a, b, c, d):
    print(f"{a}, {b}, {c}, {d}")

combinados(2, 4, c=3, d=1)
```

```
## 2, 4, 3, 1
```

Además de esto, también se debe de tener en cuenta que se puede hacer que el número de argumentos que acepte una función sean variables. Esto se puede hacer, poniendo tanto un número variable de argumentos posicionales como de argumentos no posicionales. La manera de indicar que se quieren poner un número variable de argumentos en una función es de la siguiente forma:

```python
# Para argumentos posiconales variables
def <Nombre de la función>(*args): # Se pone args por convención, pero se puede
                                   # poner otro nombre. Lo que importa es el *

# Para argumentos no posiconales variables
def <Nombre de la función>(**kwargs): # Se pone kwargs por convención, pero se puede
                                      # poner otro nombre. Lo que importa es el **

# También se pueden combinar los dos, pero siempre se deben de poner primero los
# argumentos posicionales.
```

```python
def sumar_todo(*args):
    total = sum(args)
    print(f"La suma de todos los números es: {total}")

sumar_todo(1, 2, 3)
```

```
## La suma de todos los números es: 6
```

```python
def mostrar_info(**kwargs):
    for clave, valor in kwargs.items():
        print(f"{clave}: {valor}")

mostrar_info(nombre="Antonio", edad=85, ciudad="Madrid")
```

```
## nombre: Antonio
## edad: 85
## ciudad: Madrid
```

```
def mezcla(a, *args, **kwargs):
    print(f"Argumento fijo: {a}")
    print(f"Argumentos posicionales variables: {args}")
    print(f"Argumentos no posicionales variables: {kwargs}")

mezcla(1, 2, 3, 4, nombre="Ana", edad=30, ciudad="Murcia")
```

```
## Argumento fijo: 1
## Argumentos posicionales variables: (2, 3, 4)
## Argumentos no posicionales variables: {'nombre': 'Ana', 'edad': 30, 'ciudad': 'Murcia'}
```

4.7.1. Funciones lamda.

Las funciones lamda son funciones sencillas que se escriben en una única línea. Este tipo de funciones se emplean para funciones sencillas, puesto que tendríamos la restricción de escritura de una línea. Las funciones lambda se definen de la siguiente manera:

```
<Nombre de la función> = lambda <parámetros>: <expresión>
```

Para probar como funciona, se realizan varios ejemplos.

```
suma = lambda a, b: a + b

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

```
## 5
```

```
textoMayusculas = lambda texto: texto.upper()

print(textoMayusculas("Lucas"))
```

```
## LUCAS
```

4.7.2. Función input().

La función input() permite pedirle un dato a un usuario por consola. La forma de emplear esta función es la siguiente:

```
<Nombre de la variable introducida> = input("<mensaje que se muestra en consola>")
```

Como este tipo de documento no permite interactuar, no se puede probar.

El principal problema que tiene esta función es que todos los datos que se introducen por consola vienen con tipo str. Para poder cambiar de tipo str a otro, se puede emplear el siguiente método:

```
int(<Nombre de la variable introducida>) # Si el número es entero

float(<Nombre de la variable introducida>) # Si el número tiene decimales
```

Para poder probar estos cambios, como no se puede hacer con input, se hará con unas variables inventadas que sean de tipo str.

```
input1 = "1"

print(type(input1)) # Antes de cambiarlo.
```

```
## <class 'str'>
```

```
input1 = int(input1)

print(type(input1)) # Después de cambiarlo.
```

```
## <class 'int'>
```

```
input2 = "2.22"

print(type(input2)) # Antes de cambiarlo.
```

```
## <class 'str'>
```

```
input2 = float(input2)

print(type(input2)) # Después de cambiarlo.
```

```
## <class 'float'>
```

4.8. Programación orientada a objetos.

La programación oritendada a objetos (POO) es una forma de organizar el código imitando el mundo real: en lugar de tener funciones y variables sueltas, se crean clases que representan cosas (objetos) con sus propiedades (atributos) y comportamientos (métodos).

Por ejemplo, si tenemos una clase que son perros:

- Sus atributos podrían ser su raza, su color o incluso su edad.
- Sus métodos serían por ejemplo el ruido que hacen, o si son capaces de caminar o volar.

Otro ejemplo pordría ser si tenemos una clase que son coches:

- Sus atributos podrían ser su color, su marca o el tamaño de sus llantas.
- Sus métodos podrían ser arrancar, frenar, etc.

Una clase es un molde para crear objetos. Para poder crear una, se debe de emplear el siguiente código:

```
class <Nombre de la clase>:
  # El nombre de la clase debe de empezar en mayúsculas, por convención.

  def __init__(self, <Atributo 1>, <Atributo 2>, <Atributo 3> = <Valor>):
    # __init__ se ejecutará automáticamente cuando se creen nuevos objetos, y
    # se usa para inicializar sus atributos (únicamente los que tienen un valor
    # asignado en la creación de la clase).
```

```
    # self representa al objeto que se esta creando dentro de la clase.

    self.<Atributo 1> = <Atributo 1>
    self.<Atributo 2> = <Atributo 2>
    self.<Atributo 3> = <Atributo 3>

    # A los atributos que se les ha asignado el mismo nombre que el del atributo,
    # tendrán parámetros que le asignemos nosotros a la hora de crear el objeto.

    # El atributo que tiene asignado un valor, se guardará en __init__, y este
    # será el mismo parámetro para todos los objetos de esta clase.

  def <Nombre de la función>(self):
    <Código que se ejecutará cuando el objeto utilice este método>
```

Un ejemplo de definición de clase puede ser el siguiente. Donde voy a crear la clase de coches, que tendrá como atributos el color, si es gasolina o diesel, y se va a suponer que todos los coches tienen el mismo precio (1000€). Además tendrá como metodo arrancar y tocar la bocina.

```
class Coches:
  def __init__(self, color, carburante, precio = 1000):
    self.color = color
    self.carburante = carburante
    self.precio = precio

  def arrancar(self):
    return "El coche ha arrancado (brummmm!!!)"

  def bocina(self):
    return "PI PIIIII"
```

Una vez creada la clase, se pueden crear objetos que permanezcan a esta clase de objetos, para ello se debe de emplear el siguiente código:

```
<Nombre del objeto> = <Nombre de la clase>(<Atributo 1>, <Atributo 2>)

# Los atributos que se ponen, son los que no tienen un valor asignado previamente.
```

Para comprobar esto, y para poder probar los métodos y luego más adelante las herencias, se van a crear varios objetos dentro de la clase, es decir, se van a crear varios coches.

```
coche1 = Coches("azul", "gasolina")

coche2 = Coches("negro", "diesel")

coche3 = Coches("blanco", "gasolina")
```

Para poder acceder alguno de los atributos de algún objeto ya creado, y para activar sus métodos hay que emplear los siguientes códigos.

```
# Para extraer un atributo de un objeto:

<Nombre del objeto>.<Nombre del atributo que queremos extraer>

# Si se prefiere también se puede crear un método que muestre toda la información.

# Para emplear los métodos de un objeto:

<Nombre del objeto>.<Método que se quiera emplear>()
```

Para probar esto, se mostrarán algunos de los atributos de los coches creados, y algunos emplearán sus métodos.

```
print(coche1.color)
```

```
## azul
```

```
print(coche2.carburante)
```

```
## diesel
```

```
print(coche1.precio)
```

```
## 1000
```

```
print(coche1.arrancar())
```

```
## El coche ha arrancado (brummmm!!!)
```

```
print(coche2.bocina())
```

```
## PI PIIIII
```

La herencia es un mecanismo que permite a una clase, llamada subclase o clase hija, heredar las propiedades y métodos de otra clase, llamada superclase o clase padre. El emplerar las herencias permite reutilizar código y crear una jerarquía de clases lógicas.

Para poder crear una herencia, es decir, una clase hija a partir de una clase padre, se debe de emplear el siguiente código:

```
# Si queremos que herede por defecto todos loa métodos y atributos:

class <Nombre de la clase hija>(<Nombre de la clase padre>):
  pass

# Si queremos se pueden reescribir sobre los métodos creados de la clase padre.
# Y también se pueden introducir nuevos métodos en cada clase hija.
# Sería del siguiente modo:
```

```
class <Nombre de la clase hija>(<Nombre de la clase padre>):

  def <Nombre de la función de la clase padre>(self):
    <Código que se ejecutará cuando el objeto utilice este método>

  def <Nombre de la función>(self):
    <Código que se ejecutará cuando el objeto utilice este método>
```

De la manera que hemos visto anteriormente, hemos visto como todos los atributos se heredan, pero no hemos visto como una clase hija puede tener sus atributos propios además de los heredados de la clase padre. Esto se puede lograr gracias a la función super(), la cual se emplea del siguiente modo:

```
class <Nombre de la clase hija>(<Nombre de la clase padre>):
  def __init__(self, <Atributo heredado>, <Atributo heredado>, <Atributo propio>):
    super().__init__(<Atributo heredado>, <Atributo heredado>)
    self.<Atributo propio> = <Atributo propio>
```

Para poder probar esto, se va a crear una clase hija (Marca), a partir de la clase padre (Coches). La clase hija heredará todos los métodos y atributos, y además tendrá como atributo propio el modelo, y un método propio que lo describa.

```
class Marca(Coches):
  def __init__(self, color, carburante, modelo, precio = 1000):

# Modelo va antes al no tener un valor asignado

    super().__init__(color, carburante, precio)
    self.modelo = modelo

  def queClase(self):
    print(f"Pertenezco a la clase {type(self).__name__}")
```

Una vez creada la clase hija, se crearán varios objetos dentro de esta clase para probar tanto como se crean como los métodos que tienen.

```
Hyundai = Marca("negro", "diesel", "i30 N")

Kia = Marca("azul", "gasolina", "ceed")

BMW = Marca("gris", "gasolina", "M3 Competition")
```

```
print(Hyundai.bocina())
```

```
## PI PIIIII
```

```
print(BMW.color)
```

```
## gris
```

```
print(f"Soy un Kia {Kia.modelo}, que se mueve con {Kia.carburante}")
```

```
## Soy un Kia ceed, que se mueve con gasolina
```

```
Kia.queClase()
```

```
## Pertenezco a la clase Marca
```

4.9. Control de errores y excepciones (try/except)

El control de errores y excepciones permite manejar situaciones inesperadas durante la ejecución de un programa sin que este se detenga abruptamente. En Python, se utiliza la estructura try-except para capturar y gestionar errores comunes, como entrada de datos inválida, operaciones matemáticas no permitidas o acceso a archivos inexistentes. Esta técnica mejora la robustez y usabilidad del código.

El método de try-except se usa principalmente, cuando no te fias al 100 % o bien de tu código o bien de que por ejemplo una API a la que le realizas una llamada, no este activa.

La forma de uso del try-except es el siguiente:

```
try:
  <Código que no sabemos si funciona>
except Exception as e:
  <Código que se ejecutará en el caso de que salte un error en el primero>

# Se pone Exception as e para saber que ha fallado al ejecutar el código. Para
# ver el error, solo habra que realizar un print(e) en la parte de except
```

Para que se entienda, en el código anterior, en palabras lo que hace el código es los siguiente:

- Ejecuta este código.
- Si falla haz esto (para que no se detenga la ejecución del programa)

Un ejemplo de un código que se ejecuta sin errores y otro que falla podrían ser los siguientes.

```
# Código que falla en su ejecución inicial.

try:
  operación = 10 / 0
  print(operación)
except Exception as e:
  print("Ha ocurrido un error!!!", e)
```

```
## Ha ocurrido un error!!! division by zero
```

```
# Código que no falla en su ejecución inicial.

try:
  operación = 10 / 5
  print(operación)
except Exception as e:
  print("Ha ocurrido un error!!!", e)
```

```
## 2.0
```

4.10. Módulos.

Los módulos en Python son archivos de código que contienen definiciones de funciones, clases y variables. Son archivos con extensión .py. Y pueden ser de creación propia o de creación ajena.

Existen diferentes maneras de poder implementar un módulo dentro del código. Las maneras son las siguientes:

```
# Forma 1.

import <Nombre del módulo>

# Forma 2.

import <Nombre del módulo> as <Alias que se le quiere asignar al módulo>
# Normalmente el alias se pone por convención como por ejemplo:
# pandas ----> pd
# numpy ----> np
# matplotlib ----> plt

# Forma 3.

from <nombre del módulo> import <Función que se quiere importar>

# Forma 4. (No recomendada)

from <nombre del módulo> import * # Importa todas las funciones
```

Con cada método de importación de los módulos hay una forma específica de emplear las funciones que se encuentran en su interior. La forma de llamar a las funciones según el método de impotación son las siguientes:

```
# Forma 1.

<Nombre del módulo>.<Nombre de la función>()

# Forma 2.

<Alias del módulo>.<Nombre de la función>()

# Forma 3.

<Nombre de la función importada>()

# Forma 4.

<Nombre de la función>()
```

Para mostrar como se importa un módulo, se empleará un módulo creado por mi (miModulo.py). Donde dentro únicamente se encuentran tres funciones matemáticas simples.

```
import miModulo

miModulo.sumar(2,5)
```

```
## 7
```

```
miModulo.restar(1,1)
```

```
## 0
```

```
miModulo.multiplicar(9,9)
```

```
## 81
```

Así es como se emplean los módulos propios. Para poder ver como se realiza con módulos de ajenos, en el siguiente apartado se empleará uno.

4.10.1. LLamada a una API con requests.

Una API (Interfaz de Programación de Aplicaciones) es un conjunto de reglas, protocolos y herramientas que permiten que diferentes aplicaciones o sistemas de software se comuniquen entre sí, definiendo cómo se pueden hacer solicitudes, qué datos se pueden intercambiar y en qué formato, facilitando la integración y el uso de funcionalidades externas sin necesidad de conocer los detalles internos de su implementación.

Para que se entienda, una API es como un camarero en un restaurante:

- Tú eres el cliente (programa de Python).
- La cocina es el sistema que tiene la información o realiza la acción (base de datos).
- El menú es lo que puedes pedir (las funciones que ofrece la API).
- El camarero (la API) toma tu pedido, lo lleva a la cocina, y trae de vuelta lo que pediste.

Para realizar una llamada a una API desde Python, se usa el módulo requests. Para emplearlo, lo primero que se debe realizar es importarlo.

```
try:
  import requests
  print("Se ha importado el módulo correctamente")
except Exception as e:
  print("Error!!!", e)
```

```
## Se ha importado el módulo correctamente
```

```
help(requests) # Muestra información sobre el módulo que queramos.
```

```
## Help on package requests:
##
## NAME
##     requests
##
## DESCRIPTION
##     Requests HTTP Library
##     ~~~~~~~~~~~~~~~~~~~~~
##
##     Requests is an HTTP library, written in Python, for human beings.
```

```
##     Basic GET usage:
##
##        >>> import requests
##        >>> r = requests.get('https://www.python.org')
##        >>> r.status_code
##        200
##        >>> b'Python is a programming language' in r.content
##        True
##
##     ... or POST:
##
##        >>> payload = dict(key1='value1', key2='value2')
##        >>> r = requests.post('https://httpbin.org/post', data=payload)
##        >>> print(r.text)
##        {
##          ...
##          "form": {
##            "key1": "value1",
##            "key2": "value2"
##          },
##          ...
##        }
##
##     The other HTTP methods are supported - see `requests.api`. Full documentation
##     is at <https://requests.readthedocs.io>.
##
##     :copyright: (c) 2017 by Kenneth Reitz.
##     :license: Apache 2.0, see LICENSE for more details.
##
## PACKAGE CONTENTS
##     __version__
##     _internal_utils
##     adapters
##     api
##     auth
##     certs
##     compat
##     cookies
##     exceptions
##     help
##     hooks
##     models
##     packages
##     sessions
##     status_codes
##     structures
##     utils
##
## FUNCTIONS
##     check_compatibility(
##         urllib3_version,
##         chardet_version,
##         charset_normalizer_version
##     )
```

```
##
## DATA
##     __author_email__ = 'me@kennethreitz.org'
##     __build__ = 143877
##     __cake__ = '    '
##     __copyright__ = 'Copyright Kenneth Reitz'
##     __description__ = 'Python HTTP for Humans.'
##     __license__ = 'Apache-2.0'
##     __title__ = 'requests'
##     __url__ = 'https://requests.readthedocs.io'
##     chardet_version = None
##     charset_normalizer_version = '3.4.4'
##     codes = <lookup 'status_codes'>
##
## VERSION
##     2.32.5
##
## AUTHOR
##     Kenneth Reitz
##
## FILE
##     c:\users\jplaz\appdata\local\python\pythoncore-3.14-64\lib\site-packages\requests\__init__.py
```

A la API que se le va a realizar la llamada para probar es la siguiente: https://randomuser.me/api/.

```
respuesta = requests.get("https://randomuser.me/api/")

print(respuesta)
```

```
## <Response [200]>
```

```
# Para transformar la respuesta a un JSON que se pueda leer se emplea la
# siguiente función:

datos = respuesta.json()

print(datos)
```

```
## {'results': [{'gender': 'male', 'name': {'title': 'Mr', 'first': 'Adán', 'last': 'Balderas'}, 'locati
```

```
# Como vienen muchas personas, se muestra unicamente el nombre y el apellido de
# una de la siguiente forma:

print(datos['results'][0]['name'])
```

```
## {'title': 'Mr', 'first': 'Adán', 'last': 'Balderas'}
```

5. SQL

SQL (Structured Query Language) es un lenguaje de programación que se utiliza para comunicarse con bases de datos relacionales. Sirve para guardar, buscar, actualizar o eliminar información de forma sencilla y rápida. Es como un traductor entre las personas y la base de datos.

Una base de dato relacional es un tipo de base de datos que organiza la información en tablas, como si fueran hojas de cáculo. Cada tabla contine filas y columnas. Donde las filas son registros y las columnas los campos o atributos. Lo que hace relacinal a una base de datos, es que las tablas se pueden conectar entre sí.

Existen muchos tipos de bases de datos SQL, como pueden ser:

- PostgreSQL.
- MariaDB.
- Microsoft SQL Server.
- MySQL.
- ect…

Pero la mayoría de ellas son muy parecidas y la lógica es la misma. Por lo que aprendiendo SQL en cualquiera de ellas se va a poder trabajar en cualquiera.

Para poder prácticar con SQL, y comprobar que los códigos funcionan, se va a trabajar con una base de datos SQLite. Se trabajará con esta, por la sencillez de conexión al no estar conectada a un servidor. Esta base de datos se ejecuta temporalmente en la memoria RAM del dispositivo.