# 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:

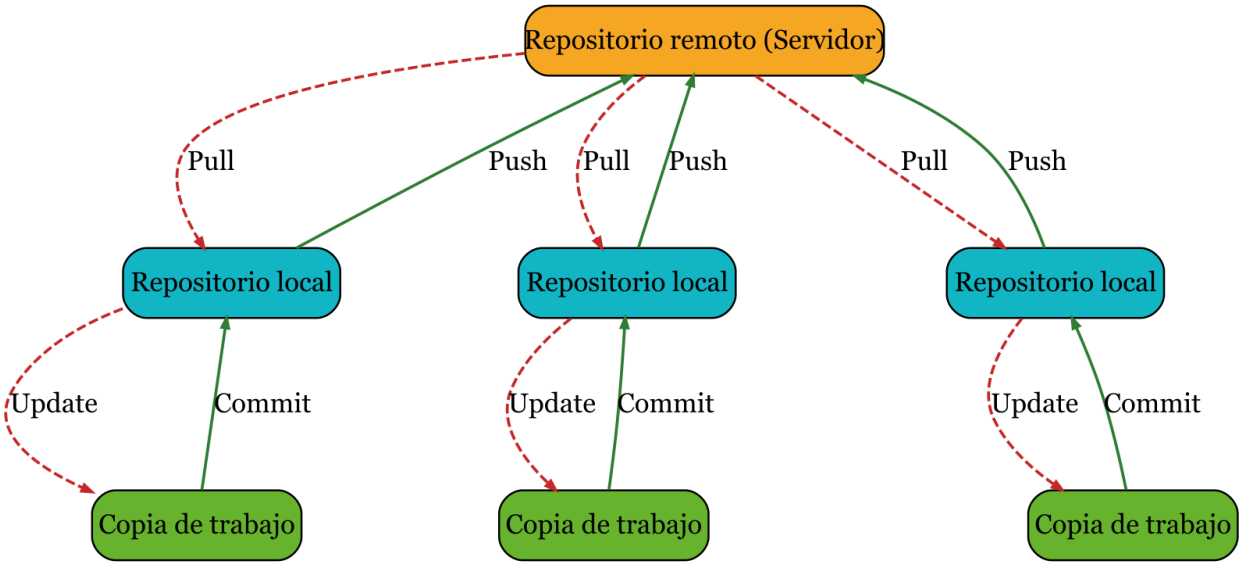
- Título.
- Descripción.
- Hash. Un hash es un código único, el cual es único por cada commit.
- Hora.

**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.

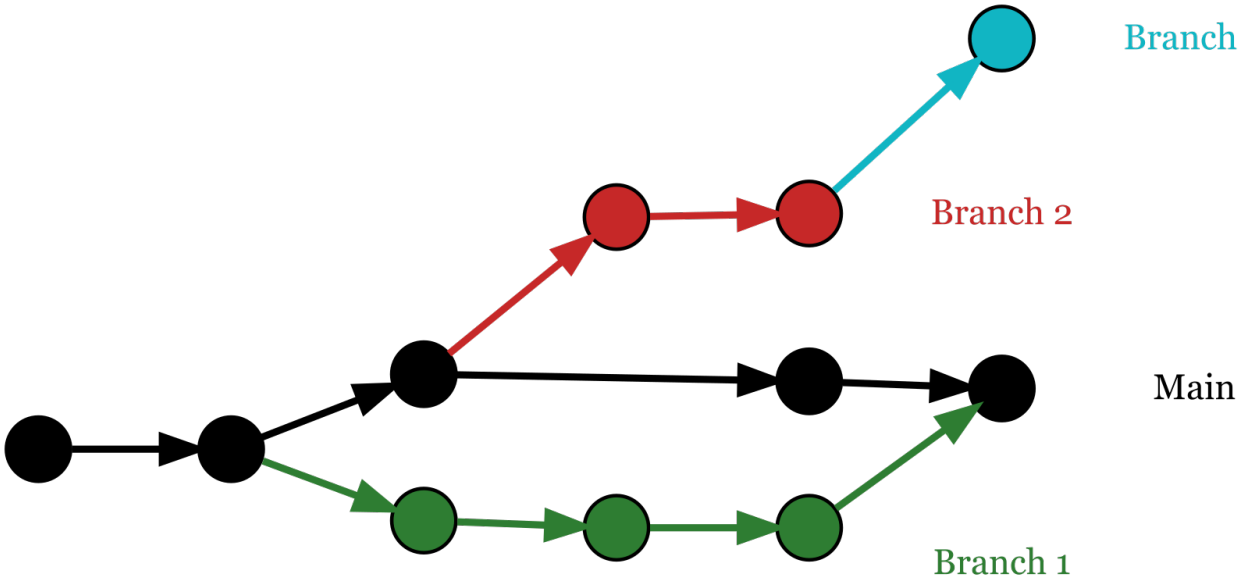
A continuación se encuentra una imagen, que refleja el flujo de trabajo con Git, con lo explicado anteriormente.




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:

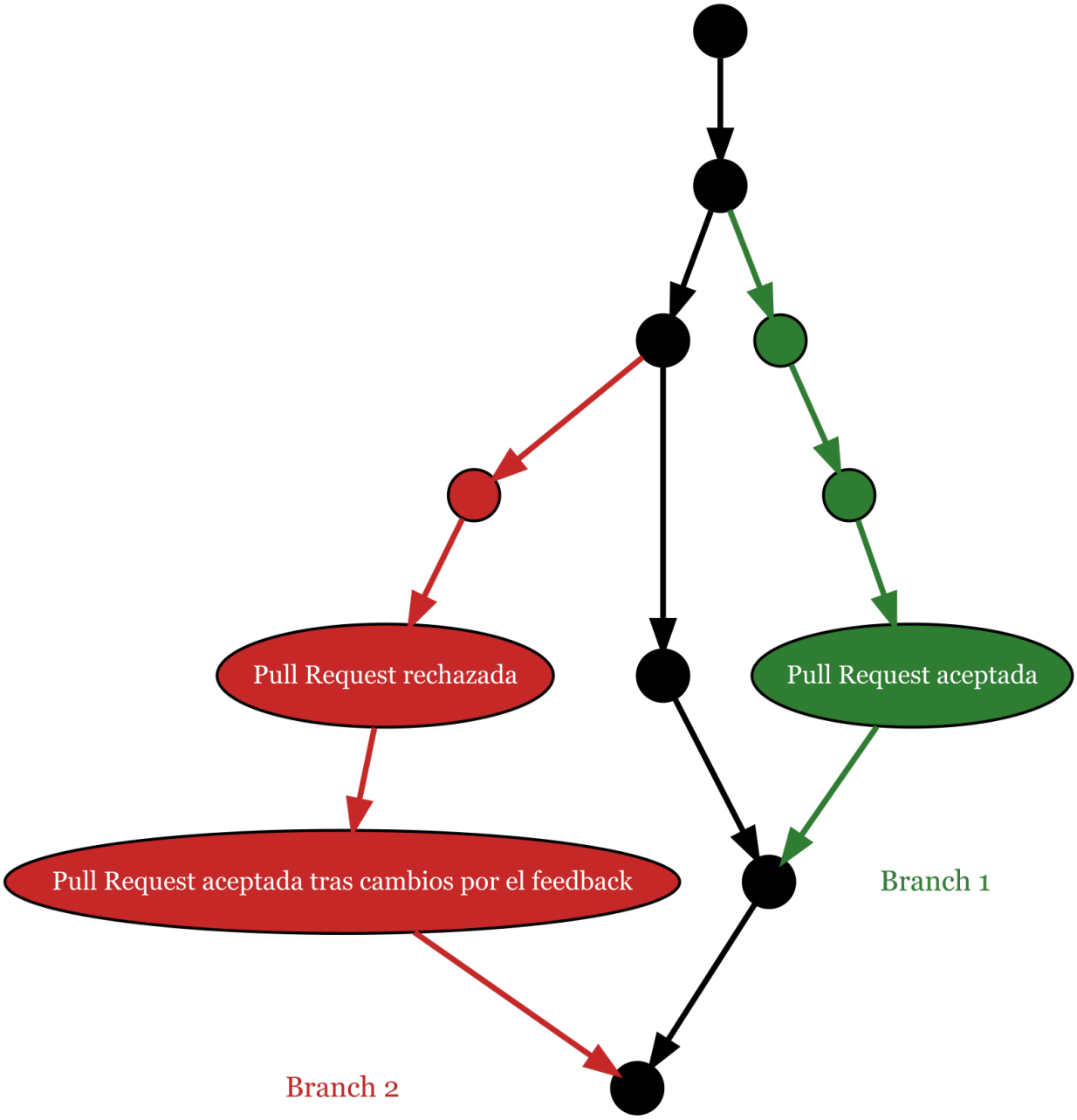
```
git branch -d <Nombre de la rama que queremos eliminar>

# -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 sse 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
```