

# Contents

<b>1</b>	<b>Python</b>	<b>3</b>
1.1	Fundamentos	3
1.1.1	Scope y variables del entorno	3
1.2	Tipos de datos, estructuras, objetos incorporados, funciones	5
1.2.1	variables y su almacenamiento	5
1.2.2	Funciones	5
1.2.3	Tupla	5
1.2.4	Lista	6
1.2.5	Diccionario	7
1.2.6	open()	8
1.3	Errores y debuggeo de los mismos	8
1.3.1	Tipos de testeo	8
1.3.2	Errores	9
1.3.3	Handlers	10
1.3.4	assertions	10
1.4	POO	10
1.4.1	Métodos importantes	11
1.5	Sobre algoritmos	11
<b>2</b>	<b>Pandas</b>	<b>12</b>
2.1	Funciones de entradas y salidas	12
2.1.1	pandas.read_csv()	13
2.2	Funciones generales	13
2.2.1	Filtrado de información	13
2.2.2	Eliminación de filas y columnas	14
2.2.3	Añadidura de columnas	15
2.2.4	Manipulación de valores	16
2.2.5	Métodos de ordenamiento	17
2.2.6	Estadísticas con agregación	18
2.2.7	Uso de funciones definidas en las tablas	18
2.2.8	Multiindexación	19
2.3	Objeto DataFrame	19
2.3.1	Constructor	20
2.3.2	Atributos	20
2.3.3	Métodos	21
2.4	Objeto Series	36
2.4.1	Métodos y atributos	36
2.4.2	Creación de series	39
2.5	Objeto index	41
2.5.1	Métodos	41
2.5.2	Atributos	45
<b>3</b>	<b>Scrapping</b>	<b>47</b>
3.1	Introducción al Scrapping	47
3.1.1	XPath	47

<b>4</b>	<b>Git and github</b>	<b>49</b>
4.1	Algunas extensiones de VSCode para Git	49
4.1.1	Git History	49
4.1.2	Git Blame	49
4.1.3	Git Lens	50
4.2	Fundamentos de Git: cómo funciona por dentro	50
4.2.1	Crear un nuevo repositorio	50
4.2.2	Objetos en Git	50
4.2.3	JSON	51
4.2.4	Hash	52
4.2.5	Exploración de objetos de git mediante cat	52
4.2.6	Creación de objetos mediante comandos de git	52
4.2.7	Tree	53
4.2.8	Permisos de objetos de git	53
4.2.9	Creación de objetos Tree	53
4.2.10	Tres pilares importantes	54
4.2.11	Git checkout index	54
4.3	Operaciones básicas de Git	54
4.3.1	¿Qué es commit?	54
4.3.2	Creación de las primeras versiones	55
4.3.3	Comandos básicos de git	55
4.3.4	Historial de un archivo	56
4.4	Las ramas	57
4.4.1	HEAD	57
4.5	Repositorios remotos	59
4.5.1	git diff	59
4.6	Fusión o combinación de ramas	59
4.6.1	conflictos de fusión	60
4.7	Comandos para los repositorios remotos	60
4.7.1	Git push	61
4.7.2	Git fetch y pull	61
4.7.3	Ramas rastreadas	62
4.7.4	Proceso pull	63
4.7.5	Fetch head	63
4.7.6	Git pull con modificacion de repositorio remoto	64
4.7.7	Subida de cambios al repositorio remoto	64
4.7.8	De rama local a rama nueva remota	65
4.7.9	Actualización de estados de ramas	65
4.8	Pull Requests	66
4.8.1	Paso a paso del proceso de solicitudes de integración	66
<b>5</b>	<b>Datos</b>	<b>68</b>
5.1	Modelo de datos	68
5.1.1	Normalización de los datos	68
5.1.2	Expresiones de análisis de datos	68
5.1.3	Parámetros 'what if'	72
5.2	SQL	72
5.2.1	Conceptos	72
5.2.2	Pasos para crear una database y usarla	74
5.2.3	Tipos de datos	74
5.2.4	Variables de puntos flotantes y puntos fijos	74
5.2.5	Otros tipos de datos	75
5.2.6	Creación de tablas	75
5.2.7	Restricciones de SQL	75
5.2.8	Buenas prácticas	78
5.2.9	Manipulación de datos	78

# Chapter 1

# Python

## 1.1 Fundamentos

### 1.1.1 Scope y variables del entorno

El término "scope" en el contexto de la programación se puede traducir al español como "ámbito" o "alcance". El alcance o ámbito de una variable en Python se refiere a la región del programa donde esa variable es válida y puede ser accedida.

En Python, existen diferentes niveles de alcance, como el alcance global y el alcance local. El alcance global se refiere a las variables que están definidas fuera de cualquier función o clase y son accesibles desde cualquier parte del programa. El alcance local se refiere a las variables que están definidas dentro de una función o clase y solo son accesibles dentro de esa función o clase. El ejemplo más claro puede ser visto a continuación

```
enemies = 1
```

```
def increase_enemies():
    enemies = 2
    print("enemies is", enemies)
```

```
increase_enemies()
print("enemies is", enemies)
```

En este caso se llama a una función que cambia la variable "enemies" de 1 a 2. Sin embargo, la salida de este programa es

```
enemies is 2
enemies is 1
```

Podemos ver que fuera de la función, que fue donde se declaró la variable `enemies` esta no cambió de valor; solamente fue cambiada dentro de la función. Si por ejemplo tratáramos de escribir un programa que declare una variable solamente dentro de la función y la intentamos imprimir fuera de ella,

```
def drink():
    var = 1
    print(var)
```

```
print(var)
```

Saldrá un error `NameError: name 'var' is not defined`. Esto es porque en este caso y en el anterior, las variables `enemies` y `var` son variables locales y están dentro del ámbito o alcance de la función que las declara o modifica.

Por su parte, cualquier variable declarada fuera de todas las funciones y clases, son denominadas como variables locales y el ámbito o alcance de estas será global. Y es accesible desde cualquier parte del programa. Este concepto de ámbito o alcance no solamente aplica para las variables sino que también aplica para funciones, entre otro tipo de elementos.

#### 1.1.1.1 Espacio de nombres

El concepto de espacio de nombres hace referencia a la manera que se tiene de organizar los diferentes nombres; sean de clases, funciones, variables, etc. Cada espacio de nombre puede ser visto como un contenedor con todos los nombres definidos. Existen diferentes tipos de namespaces:

1. Namespace Global: Es el namespace de nivel superior y contiene los nombres definidos en el alcance global, es decir, fuera de cualquier función o clase. Los nombres definidos en el namespace global son accesibles desde cualquier parte del programa.
2. Namespace Local: Es el namespace creado cuando se define una función o clase. Contiene los nombres definidos dentro de esa función o clase y solo son accesibles desde su interior.
3. Namespace de Módulo: Cada archivo de Python se considera un módulo y tiene su propio namespace. Los nombres definidos en un módulo son accesibles desde otros módulos si se realiza una importación.
4. Namespace Incorporado (Built-in): Contiene los nombres predefinidos que son proporcionados por Python de manera predeterminada. Estos nombres incluyen funciones y tipos incorporados como `print()`, `len()`, `str()`, etc.

Una característica importante de Python es que no tiene un ámbito de bloque, a diferencia de otros lenguajes de programación. Esto quiere decir, por ejemplo,

```
if var1:
    <code>
    <code>
    <code>
    <code>
```

Si el lenguaje tiene ámbito de bloque, entonces cualquier definición dentro de las líneas subordinadas al `if` anterior solamente existirán dentro del `if`. En Python esto no pasa, cualquier variable que este definida aquí estará dentro del ámbito en que se encuentre el `if`.

#### 1.1.1.2 Cómo modificar una variable global

Una cosa importante dentro del tema de los entornos es que siempre que tenemos dos ámbitos diferentes; por ejemplo el entorno global y un entorno local, una variable global comparada con una variable local son dos variables completamente diferentes, aunque tengan el mismo nombre. Es muy mala idea nombrar dos variables con el mismo nombre, aunque estén en dos ámbitos diferentes. En el caso dado de que se requiera modificar el valor de una variable global dentro de una variable local, es imprescindible indicar explícitamente que se trata de una variable global:

```
var = 2
def fun():
    global var
    var = 1
    print(var)
fun()
print(var)
```

En este caso, no se crea una nueva variable en el entorno local de la función, sino que se modifica la variable global declarada al principio.

En general no es muy recurrente el uso de este método de cambio de variables globales, porque se presta para confusiones y facilita de cierta manera los errores. Sin embargo, el uso de las variables globales es importante por ejemplo cuando se necesita declarar una variable constante dentro del programa. Siempre se usa como convención usar letras mayúsculas y barras bajas para declarar constantes dentro del programa (ejemplo `MY_EMAIL = "sebas@unal.edu"`).

## 1.2 Tipos de datos, estructuras, objetos incorporados, funciones

### 1.2.1 variables y su almacenamiento

al entrar en una funcion, normalmente el entorno se reinicia, con lo cual la mayoría de las variables que se declaren o se modifiquen solamente lo hará en ese entorno. Sin embargo, algunas funciones de algunos tipos de datos (mayoritariamente modificar) como `list.append()`

### 1.2.2 Funciones

Una manera de especificar mejor los argumentos de las funciones, es mediante la siguiente forma:

```
my_fun(a=1, b=2, c=3)
```

### 1.2.3 Tupla

De manera similar que una secuencia de caracteres, una tupla es una secuencia de datos de distintos tipos. Colección de distintos datos. Este no se puede modificar una vez se declara. Esto es, no son mutables y por tanto se almacenan en un solo bloque de memoria.

```
tupla = (2,"hola",False)
```

Si se suman las tuplas a y b, el resultado es una concatenación

```
tupla = (1,2,3)
tupla2 = (4,5,True)
tupla3 = tupla + tupla2
print(tupla3)
La salida es
(1,2,3,4,5,True)
```

Multiplicar una tupla:

```
a = (1,2)
print(a*5)
```

```
output: (1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
```

Recorrer una tupla:

```
for i in tuple:
    print(i)
```

Verificar si es miembro

```
3 in (1,2,3)
```

La tuplas son útiles para hacer cambios de variables en una línea misma:

```
x = 2
y = 5
(x,y) = (y,x)
print(x)
output: 5
```

También para definir una función en la que retornan varios valores

```
def funcion(x,y):
    q = x // y
    r = x % y
    return (q,r)
```

```
(cociente,residuo) = funcion(47,11)
print(residuo)
```

output : 3

Para retornar el enésimo elemento de una tupla se pone `tuple[i]`. Recordar que los índices van desde cero hasta `length(tuple)-1`.

**Nota:** `tuple[1:5]` retornará los valores de la tupla desde el índice 2 hasta el 4.

- `len(tuple)`: Retorna el tamaño de la tupla
- `max(tuple)`: Retorna el valor máximo de la tupla. Si en la tupla hay cadenas de caracteres, tuplas o listas, retorna un error.
- `tuple(List)`: Retorna una tupla conformada con los elementos de la lista List.

## 1.2.4 Lista

Otro tipo de arreglo de datos es la lista. La principal diferencia entre tupla y lista es que la lista sí es mutable y también ocupa dos bloques de memoria; esto hace que trabajar con tuplas sea más rápido pero la ventaja de la lista es que es modificable.

```
list1 = ['physics', 'chemistry', 1997, 2000]
```

El tipo de acceso de los elementos de una lista es el mismo que para las tuplas. De la misma forma las operaciones; ver la sección anterior en las operaciones de listas.

### 1.2.4.1 Lista de funciones

**len(list)** Retorna el tamaño de la lista.

**max(list)** Retorna el valor del máximo, si la lista contiene combinaciones de números y caracteres o listas o tuplas, genera error.

**min(list)** Retorna el valor mínimo dentro de la lista.

**list(seq)** Retorna una lista compuesta por los elementos de seq.

**sorted(list)** Retorna una lista con los elementos de list ordenados de menor a mayor.

### 1.2.4.2 Lista de métodos

**list.append(obj)** Añade el objeto obj al final de la lista

**list.count(obj)** Retorna el número de veces que el objeto obj ocurre en la lista.

**list.extend(seq)** Añade el contenido de seq a la lista. Lo añade al final de la lista.

**list.index(obj)** Retorna el índice más pequeño en la lista en que el objeto obj aparece.

**list.insert(index, obj)** Inserta el objeto obj en la casilla index de la lista, moviendo el resto una posición.

**list.pop(obj = list[-1])** Remueve y retorna el objeto que se encuentre en la posición obj. Por defecto si no se ingresa argumento, remueve el último objeto de la lista.

**list.remove(obj)** Remueve el primer objeto obj que encuentre en la lista.

**list.reverse()** Invierte el orden de los componentes de la lista.

**list.sort(key=None)** Organiza los elementos de la lista, si hay una directiva de ordenamiento, se puede ingresar como el argumento **key**, por defecto, organiza de menor a mayor.

**list.clear()** Elimina todos los componentes dentro de la lista dejándola como una lista vacía.

**list.copy()** Retorna una copia de la lista.

**Nota** Si se declara una lista como sigue

```
A = [1,2,3]
B = A
```

Tanto A como B están apuntando al mismo objeto, o dirección de memoria. Para realizar una copia de A en otro espacio de memoria se utiliza

```
B = A[:]
```

## 1.2.5 Diccionario

Es una lista especial en la que se puede dar un identificador especial al índice de la misma

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print(dict["Name"])
```

El primer objeto del diccionario tiene un identificador llamado "Name" y un valor asociado a él que en este caso es "Zara". La salida del código anterior será

```
Zara
```

Se puede modificar el valor de una entrada en el diccionario y se puede también añadir una nueva entrada

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School" # Add new entry
```

Para borrar elementos de un diccionario

```
del(dict["Name"])
```

Una propiedad importante de los diccionarios es que los elementos pueden ser cualquier objeto de python. Las 'llaves' o identificadores deben ser objetos inmutables como cadenas de caracteres o tuplas.

### 1.2.5.1 Funciones

**len(dic)**

**str(dic)** Retorna una cadena de caracteres imprimible de los elementos que contiene el diccionario **dic**

### 1.2.5.2 Métodos

**dic.clear()**

**dic.copy()**

**dic.fromkeys(iterable,value)** Retorna un nuevo diccionario cuyas 'llaves' estarán determinadas por los elementos de **iterable** y con un valor asociado (único) de **value**

**dic.get(key, default=None)** Retorna el valor correspondiente a la llave **key**. El segundo argumento es el retorno cuando no hay una llave que corresponda.

**dic.items()** Este método retorna una lista de tuplas, cada tupla corresponde a la llave y al valor correspondiente.

**dic.keys()** Retorna una vista de todas las llaves del diccionario. Se puede conseguir la lista nativa a través de `list()`.

**dic.setdefault(key, default = None)** Es similar a `get()` con la diferencia de que si la llave `key` no está en el diccionario, entonces la añade y su valor correspondiente será `default`

**dic.update(dict2)** Actualiza el diccionario añadiendo todos los pares (llave-valor) de `dict2`.

**dic.values()** Retorna una vista de los valores del diccionario. Se puede conseguir la lista nativa a través de `list()`.

## 1.2.6 open()

El retorno de esta función es un objeto `file`.

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

- `file` es un objeto `path-like`, es el nombre del archivo a abrir (incluyendo la ruta si es necesario) o crear.
- `mode` es un string opcional que especifica el modo en el que el archivo se abre, por defecto `'r'` para leer `'w'` para escribir, truncando el archivo primero `'x'` para creación de archivo nuevo, falla si ya hay un archivo con ese nombre, `'a'` para escribir en el archivo, anexando al final del archivo si este existe, `'b'` modo binario, `'t'` es modo de texto que está por defecto, `'+'` para abrir y actualizar (leer y escribir)

Los archivos abiertos en el modo binario retornan el contenido como objetos `byte` sin ninguna decodificación; en el modo texto, el contenido se lee como `string`.

### 1.2.6.1 Concatenación especial de cadenas de caracteres.

una forma interesante de hacer concatenación de caracteres es mediante la siguiente forma. Sea `score=0` una variable del tipo `int`. Si hacemos `print(f"your score is {score}")` se hará la conversión de entero a carácter automáticamente sin la necesidad de hacer `print("your score is" + str(score))`

## 1.3 Errores y debuggeo de los mismos

### 1.3.1 Tipos de testeo

#### 1.3.1.1 Test unitario

Si el programa es modular, es posible hacer tests que aseguren que cada función hace lo que se supone que debe hacer según las especificaciones.

#### Test de regresión

Cada vez que se soluciona un error, se realiza testeo nuevamente del código, con el objetivo de asegurarse que al realizar la corrección no se agregaron nuevos errores.

#### Test de integración

Realizar testeo del programa como un todo. Se ponen juntas cada una de las partes individuales



### 1.3.1.2 back box testing

Se tiene el código y se realizan las pruebas con diferentes casos con el fin de encontrar todas las rutas posibles que hay en el código.

Se determina el docstring de una función, ejemplo:

```
def sqrt(x,eps)
    """Asume x y eps como flotantes, mayores que cero o igual para x, y retorna un res tal que x
```

La idea es entonces realizar testeos de diferentes casos dadas las especificaciones del docstring.

En el caso del ejemplo anterior, se puede hacer un conjunto de pruebas con valores como raíces cuadradas perfectas, números irracionales, menores que 1, o por ejemplo con valores extremos como muy pequeño y muy grandes de ambos x y eps.

### 1.3.1.3 glass box testing

En este caso lo que se hace es utilizar directamente el código para guiar los casos de prueba. En este caso se pueden llegar a presentar muchas posibilidades de caminos disponibles, teniendo en cuenta la posible presencia de bucles y repeticiones en el código. Por ejemplo, para ramas en los que hay diferentes casos, es importante lograr hacer la prueba para todos y cada uno de los posibles caminos o casos. Para bucles `for`, se deben preparar pruebas en las que no se entra a dicho bucle, también pruebas en las que se entra una vez, dos veces, tres y así sucesivamente. Para bucles `while` es de manera similar, pero asegurándose de tener casos de prueba que puedan cubrir todas las formas posibles de romper el bucle.

Hacer el debugging tiene una variedad grande de posibilidades. Utilizar `print`, por ejemplo, dentro de funciones o bucles.

## 1.3.2 Errores

### 1.3.2.1 IndexError

```
test[1,2,3]
test[4]
```

### 1.3.2.2 TypeError

```
int(test)
```

### 1.3.2.3 NameError

cuando no se encuentra un nombre ya sea local o global.  
a una variable inexistente.

### 1.3.2.4 SyntaxError

Errores de sintaxis. Cuando python no puede interpretar o analizar gramaticalmente el código.

### 1.3.2.5 AttributeError

las referencias a atributos falla.

### 1.3.2.6 ValueError

el tipo de operador está correcto, pero el valor del mismo es imposible.

### 1.3.2.7 IOError

El sistema IO reporta una malfunción (por ejemplo un archivo no encontrado). Los errores son llamados excepciones.

ahora por alguna razón esto no deja seguir y continuar

### 1.3.3 Handlers

Los llamados manejadores, son lo que se encargan de llevar a cabo la rutina o ejecución necesaria cuando determinada cosa ocurre, sean interrupciones o excepciones.

```
try:
    xxxxxxxx
    xxxxxx
except (exception_type1):
    xxxxxxxx
    xxxxxx
except (exception_type2):
    xxxxxxxx
    xxxxxx
.
.
.

else:
```

lo anterior se ejecuta cuando el cuerpo del try asociado se ejecuta sin ninguna excepción.

```
finally:
```

Siempre se ejecuta después del try, else, y except, incluso cuando existen break, continue o return.

```
raise <ExceptionName> (<Arguments>)
raise <ValueError> ("Uis, algo esta mal")
```

### 1.3.4 assertions

programación "defensiva".

```
assert <lo_que_se_espera>, <mensaje>
```

Da un error de ejecución del tipo AssertionError. dando la explicación pertinente.

Ahora hay errores lógicos que son más complicados de tratar.

Cosas que no se deben hacer:

1. Escribir el código entero para hacer pruebas sobre él
2. Hacer debug en el programa entero
3. Olvidar en qué lugar estaba el bug
4. Olvidar cuáles fueron los cambios que se hicieron

Por el contrario es más recomendable escribir una función, probarla, hacer depuración, y así con cada función nueva que se escriba. Hacer Testeo de integración. También hacer copias de seguridad del código, cambiarlo, advertir mediante comentarios los cambios, y al realizar pruebas, hacer comparaciones.

## 1.4 POO

object es el tipo más básico en python.

```
class <name>(<parent_class>):

class coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

El self es un parámetro para referirse a la instancia de la clase, la que esté ejecutándose. El constructor siempre será def \_\_init\_\_():

### 1.4.1 Métodos importantes

Estos métodos sustituyen operadores o funciones importantes.

- `__str__()`: Su retorno es lo que se muestra cuando se ejecuta la función `print()`
- `__add__()`: Su retorno es el valor de `'a+b'`.
- `__sub__()`: Su retorno es el valor de `'a-b'`.
- `__mul__()`: Su retorno es el valor de `'a*b'`.
- `__eq__()`: Su retorno es el valor de `'a==b'`.
- `__lt__()`: Su retorno es el valor de `'a<b'`.
- `__len__()`: Su retorno es el valor de `'len(a)'`.

**nota** las **variables de clase** son variables cuyo valor se comparte entre todas las instancias de la clase

## 1.5 Sobre algoritmos

¿Cómo se puede establecer o sabe qué tan eficiente es mi algoritmo?

Los razonamientos que surgen a partir de este punto dan como resultado un análisis interesante sobre las siguientes cuestiones:

- Cómo podemos razonar sobre un algoritmo con el objetivo de predecir la cantidad de tiempo que este necesitará para resolver un problema de un tamaño en particular.
- Cómo podemos relacionar las opciones en el diseño de algoritmos con la eficiencia en tiempo del resultado.

## Chapter 2

# Pandas

En el contexto de la biblioteca ‘pandas’ en Python, un `**dataframe**` es una estructura de datos bidimensional, similar a una tabla en una base de datos, una hoja de cálculo o una tabla en lenguajes estadísticos como R. Es esencialmente una colección ordenada de columnas, donde cada columna puede tener un tipo de dato diferente (número, cadena, booleano, etc.).

El término “data tabular” se refiere precisamente a este tipo de datos en forma de tabla, donde:

- Las *filas* representan observaciones o entradas individuales.
- Las *columnas* representan características o variables de esas observaciones.

Matemáticamente, podríamos considerar un dataframe como una matriz  $M$  de dimensiones  $m \times n$ , donde  $m$  es el número de filas y  $n$  es el número de columnas. Cada elemento  $M_{ij}$  representa el valor en la  $i$ -ésima fila y  $j$ -ésima columna.

Un dataframe en ‘pandas’ proporciona una serie de funcionalidades útiles para manipular y analizar datos tabulares, tales como:

1. Operaciones de filtrado y selección basadas en condiciones.
2. Agrupaciones y operaciones de agregación.
3. Fusiones y uniones con otros dataframes.
4. Operaciones de pivote y reestructuración.
5. Funcionalidades para manejo de datos faltantes.
6. Integración con otras bibliotecas de Python como ‘numpy’, ‘scipy’ y ‘matplotlib’.

1.

Para ilustrar esto con un pequeño ejemplo, considera el siguiente dataframe:

Nombre	Edad	Profesión
Juan	30	Ingeniero
Ana	25	Doctora
Carlos	28	Arquitecto

En este dataframe: Las filas representan a diferentes individuos. Las columnas representan características de estos individuos: su nombre, edad y profesión.

La facilidad con la que ‘pandas’ permite manipular, filtrar y analizar este tipo de datos ha hecho que sea una herramienta esencial para cualquier persona que trabaje con análisis de datos en Python.

## 2.1 Funciones de entradas y salidas

Estas son las funciones mediante las cuales se pueden obtener diferentes objetos de pandas o guardar como salidas.

### 2.1.1 `pandas.read_csv()`

Esta función es una de las más utilizadas porque importa directamente un archivo `csv` y lo convierte en un `DataFrame` (en la siguiente sección se hablará de este objeto)

#### 2.1.1.1 Parámetros

##### `filepath_or_buffer`

- **Descripción:** Es la ruta del archivo `csv` a importar. Se puede usar una URL
- **Tipo:** `str`. También acepta objetos tipo `PathLike` u objetos `path`.

##### `sep`

- **Descripción:** El tipo de separador de los datos, normalmente son comas `,`, pero puede ser configurado según el archivo que se va a importar.
- **Tipo:** `str`.
- **Por defecto:** `,`

##### `header`

- **Descripción:** Es la especificación de la fila que contiene las etiquetas de las columnas. Por defecto siempre será la primera.
- **Tipo:** `int`, `infer` o `None`. Si se especifica `None`, las etiquetas de las columnas serán índices desde 0 hasta `n`. Si se especifica `infer`, se asume que `header = 0`
- **Por defecto:** `infer`.

##### `index_col`

- **Descripción:** Especifica la columna que se usará como etiquetas de las filas.
- **Tipo:** `hashable`. Comúnmente suelen ser del tipo numérico (`int` o `float`) o cadenas de caracteres `str`. Básicamente es la etiqueta de la columna que se usará como etiqueta
- **Por defecto:** Es opcional.

##### `usecols`

- **Descripción:** Se puede seleccionar un conjunto de columnas para importar y descartar las demás. Este parámetro especifica este conjunto
- **Tipo:** Lista de `hashables`.
- **Por defecto:** Es opcional.

## 2.2 Funciones generales

### 2.2.1 Filtrado de información

Existen varios métodos para filtrar los datos de un `dataframe`. Los más básicos son los que solamente tienen una condición. También están las múltiples condiciones.

### 2.2.1.1 Una sola condición:

Si un dataframe con nombre `df` tiene una columna cuya etiqueta o nombre es `sex`; una forma de filtrar sería la siguiente:

```
df[df.sex == "male"]
```

De esta manera se obtiene o se retorna un dataframe con la información filtrada.

Por su parte, se puede utilizar el método `loc` para realizar el filtrado:

```
df.loc[df.sex == "male"]
```

Y si por ejemplo, se quiere solamente se requiere sacar una columna, se usa

```
df.loc[df.sex == "male", "fare"]
```

### 2.2.1.2 Varias condiciones:

Una forma de filtrar por el tipo de variable en el dataframe, en este caso se seleccionan una o más columnas

```
mask = df.dtypes == "int64"  
df.loc[mask]
```

Se pueden definir diferentes máscaras, tanto para las filas como para las columnas.

```
mask1 = df["age"] >= 55  
mask2 = df.dtypes == "int64"  
df.loc[mask]
```

Adicionalmente se pueden usar funciones lógicas para mejorar la precisión del filtro:

```
df.loc[(df.sex == "male") & (df.age > 25)]
```

Existen algunas reglas adicionales para máscaras para filtrar de forma más práctica:

```
mask = summer["Year"].between(1960, 1984, inclusive = True)  
mask2 = summer["Year"].isin([1972, 1996])
```

Este último es útil para seleccionar explícitamente un conjunto de elementos a filtrar, seleccionándolas o extrayéndolas mediante la negación. El argumento siempre es una lista con los objetos que quiero filtrar.

## 2.2.2 Eliminación de filas y columnas

Existen varias formas de eliminar filas y columnas de los dataframes. La forma más básica es mediante el método `drop`. Este método tiene el parámetro `inplace`

```
df.drop(columns = "col")
```

Para eliminar más de una columna, se pasa una lista con las etiquetas correspondientes como argumento.

```
df.drop(columns = ["col1", "col2"])
```

Dos alternativas para eliminar una columna son las siguientes.

[Ir al Contenido](#)

```
df.drop(labels = "col_name", axis = "columns")
del df["col"]
```

Para eliminar las filas se utiliza la misma forma, reemplazando `columns` por `index`. Con esta forma se eliminan todas las instancias con el índice seleccionado, en caso que haya más de una fila con el mismo índice.

```
df.drop(index = index_name)
```

Otra forma es renombrando el dataframe con una versión filtrada de la misma

```
df = df.loc[df.col == value]
```

Aquí se seleccionan solamente las filas cuyo valor para la columna `col` es `value`, descartando todas las otras.

Si creamos diferentes máscaras, y luego filtramos la negación lógica de dichas máscaras, entonces estaremos eliminando las filas que cumplan con la máscara o máscaras.

```
mask1 = summer["Year"] == 2012
mask2 = summer["Country"] == "RUS"
summerf = summer.loc[~(mask1 & mask2)]
```

### 2.2.3 Añadidura de columnas

La forma más básica de añadir una nueva columna es asignando un valor constante a dicha columna

```
df["col_nueva"] = 0
```

Si `col_nueva` no existe, entonces se creará y se llenará con el valor 0 para todos los índices. Si la columna ya existe, entonces se reemplazarán todos los valores con la nueva asignación. Por su parte, si se crea una columna de la siguiente forma

```
df.new_col = 0
```

En realidad se creará un nuevo atributo para el dataframe, con valor 1, pero no se creará una columna.

Si se utiliza una operación tradicional con una columna del dataframe, se creará una columna nueva, y esto se puede utilizar para añadir columnas nuevas con base en datos de otras columnas:

```
titanic["yob"] = 1912 - titanic["age"]
```

Para añadir una nueva columna en una posición específica, se utiliza el método `insert`:

```
col_new = df[col] + 3
df.insert(loc = 6, column = "new_col_name", value = col_new)
```

De esta forma se asegura que la columna nueva sea localizada en la sexta posición. Las columnas que estén a la derecha serán desplazadas una posición.

Por otro lado, también se pueden añadir entradas nuevas o filas nuevas. Asignando una lista o iterable concordante con el data frame, a un nuevo índice.

## 2.2.4 Manipulación de valores

Se puede cambiar un solo valor mediante el atributo `loc` y también `iloc`

```
titanic.loc["index","col"] = 40
titanic.iloc[1,1] = 40
```

Para cambiar varios valores puede hacerse de la siguiente forma

```
titanic.loc[1:3, "age"] = 42
```

Se puede cambiar un rango específico de valores con una lista de valores compatible:

```
titanic.iloc[1:4, 3] = [43,44,45]
```

Recordar que con el comando `loc` si el índice es numérico, se selecciona incluyendo desde el primer número hasta el último, mientras que con `iloc` se selecciona el índice de forma tradicional como si se tratara de una lista.

Para reemplazar o cambiar valores basado en condiciones, se usa la siguiente forma

```
titanic.loc[titanic.age < 1, "age"]
```

Por su parte, si se requiere cambiar más de una columna para una entrada (fila) en particular se puede usar el siguiente código

```
titanic.loc[0,"survived":"age"] = [1, 2, "female", 24.0]
```

El método `replace` es útil cuando queremos reemplazar todas las entradas que cumplan una condición

```
titanic.replace(0, "zero")
```

Esto reemplaza todos los ceros que haya en la tabla por el texto "zero".

Es crucial entender que al declarar una variable como referencia a una columna de una tabla, en realidad estamos creando un apuntador a dicha columna. Esto significa que cualquier cambio o reemplazo de valores en esta variable implicará una modificación directa en la columna correspondiente de la tabla:

```
edad = titanic.age
edad.loc[1] = 40 # aquí
```

En el reemplazo de variables dentro de una tabla la manera correcta de realizarlo es como se mencionó en la sección anterior, realizarlo de otra forma se considera una mala práctica:

```
titanic.loc[1,"age"] = 40 # esta es forma correcta
titanic.age[1] = 40 # esta no se debe realizar
```

Por otro lado, si se asigna a una variable una o más columnas poniendo una lista como argumento, dicha variable si se modifica, no modificará la tabla original.

```
sexo = titanic[["sex"]]
sexo.loc[0] = "ajefemalee"
sexo.iloc[0] = "ajefemalee" # también aplica para iloc
titanic # no tendrá el valor modificado
```



Si se utiliza el campo para el índice [:] obtendremos el mismo resultado; sin embargo, al modificar una variable declarada de esta forma, Pandas no generará ninguna advertencia.

```
sexo = titanic.loc[:,["sex"]]
```

Ahora, si se intenta reemplazar valores mediante la notación de doble paréntesis, como se muestra a continuación

```
titanic[titanic["age"] < 1]["age"] = 1
```

La tabla no será cambiada, y se mostrará una advertencia. Cuando se ejecuta `titanic[titanic["age"] < 1]["age"]`, se está realizando dos operaciones de indexación consecutivas. Primero, `titanic[titanic["age"] < 1]` crea una vista o una copia temporal filtrada del DataFrame `titanic`, seleccionando solo las filas donde `age` es menor que 1. Luego, `["age"]` selecciona la columna `age` de esta vista o copia temporal. Al intentar asignar un nuevo valor a esta selección, se está modificando la copia temporal y no la tabla original. Pero, si se realiza una selección de la columna primero, y luego se realiza el filtro,

```
titanic["age"][titanic["age"] < 1] = 1
```

La tabla sí será modificada. Esto es debido a la forma en que se realiza la indexación y asignación en este caso específico. En Pandas, cuando se hace una indexación directa sobre una columna del DataFrame, como en `titanic["age"]`, se obtiene una vista directa de esa columna en el DataFrame original, no una copia. Esta vista es una referencia a los datos en la columna `age` dentro del DataFrame `titanic`. Luego, cuando se aplica un filtro adicional con `[titanic["age"] < 1]`, se está seleccionando ciertas posiciones de esa vista, y cualquier cambio que se haga aquí se reflejará en el DataFrame original. Al asignar `= 1` a esta selección, se está modificando directamente los datos en la columna `age` del DataFrame `titanic`. Es importante notar que, aunque esta forma de indexación y asignación puede funcionar, no es la recomendada, ya que puede llevar a confusión y a errores sutiles, especialmente en contextos más complejos.

En conclusión, es importante tener claridad sobre cuándo se está realizando una vista de un conjunto de datos tomados de una tabla (apuntador a la columna, o conjunto de datos) y cuándo se está realizando una copia (sea temporal o no) de los datos; la copia es un objeto completamente independiente de la tabla original. Si se selecciona una columna mediante la notación de atributo (`titanic.age`) o mediante la notación de paréntesis cuadrados, o cualquier otra notación, es fácil y muy útil utilizar el atributo `.is_view` y el método `.is_copy()`.

#### 2.2.4.1 Consejos de buena práctica:

En general, según el propósito que se tenga, es conveniente y recomendado utilizar cierto tipo de métodos y atributos a la hora de manipular datos de una tabla. Si se requiere trabajar y manipular la tabla entera, entonces el consejo es evitar el indexado encadenado. La indexación encadenada en Pandas se refiere al proceso de realizar varias operaciones de indexación sucesivas, una tras otra, típicamente utilizando corchetes []. Esto suele verse en la forma de `dataframe[...][...]`, donde se realizan dos (o más) operaciones de indexación de forma secuencial. Por su parte, si se desea trabajar con porciones de la tabla, es recomendado realizar una copia de dicha porción mediante el método `copy()` y trabajar sobre esta.

### 2.2.5 Métodos de ordenamiento

La manera clásica de ordenar una tabla basado en una columna es la siguiente

```
titanic.sort_values(by = "age")
```

Recordar que la línea anterior retorna la tabla ordenada, no modifica la actual, para hacer esto se debe usar el parámetro `inplace`. Recordar también el parámetro `ascending`.

Se puede hacer ordenamiento no solamente por una columna sino por varias, pasando la lista con las etiquetas de las columnas como argumento del parámetro `by`. La columna de la primera etiqueta de la lista que se pasa como argumento es la que tiene la primera prioridad. También se puede pasar una lista booleana especificando cuáles se ordenarán de forma ascendente y cuáles descendente.

```
titanic.sort_values(by = ["col1", "col2", "col3"], ascending = [True False False],
```

Siempre se puede reordenar la tabla con base en el índice de las filas:

```
titanic.sort_index(ascending = True, inplace = True)
```

Cuando se realiza un ordenamiento basado en una columna, los índices también se ordenan o "desordenan". En algunas ocasiones se requiere que el índice prevalezca ordenado aun cuando la tabla completa cambió con el ordenamiento. Esto se logra con dos formas distintas:

```
titanic.sort_values(by = age).reset_index(drop = True)
titanic.sort_values(by = "age", ignore_index = True)
```

### 2.2.6 Estadísticas con agregación

El método `agg` es utilizado para realizar operaciones de agregación sobre una tabla, permitiendo aplicar una o más funciones sobre sus ejes. Se puede especificar una función, nombre de función, lista de funciones o un diccionario que mapea etiquetas de eje a funciones. Por defecto, estas operaciones se realizan sobre las filas ('axis=0'), pero también pueden aplicarse a las columnas ('axis=1'). El resultado de 'agg' puede ser un escalar, una Serie o un DataFrame, dependiendo de la operación. Una forma efectiva de usarlo es seleccionar columnas de un tipo específico, como las numéricas, y luego aplicar un diccionario para realizar distintas operaciones en cada columna, tal como `df.select_dtypes("number").agg("col1": "mean", "col2": ["min", "std"])`. Esto permite realizar análisis detallados y personalizados sobre los datos de manera eficiente.

### 2.2.7 Uso de funciones definidas en las tablas

Una funcionalidad interesante es la posibilidad de ejecutar funciones sobre todos los elementos de una tabla. Si definimos una función

```
def range(series):
    return series.max() - series.min()
```

Al implementar el método `apply` se puede ejecutar la función para los elementos de la columna o los elementos de las filas, según se establezca:

```
sales.apply(range, axis = 0)
```

El valor de `axis = 0` es para las filas y `axis=1` para las columnas. Retornará una tabla con el resultado de la función.

Se puede pasar la función `lambda` directamente en caso de requerir poco código:

```
sales.apply(lambda x: x.max() - x.min(), axis = 0)
```

Si se selecciona una sola columna o porción de ella, el intérprete ya sabe que se ejecutará la función a dichos elementos:

```
sales.loc[1:10, "col1"].apply(lambda x: x[0])
```

El método `map` es similar, con la diferencia de que solamente aplica para series de datos, es decir una columna por ejemplo. Por último, Se puede aplicar una función a toda la tabla (o parte de ella) mediante el siguiente método.

```
df.applymap(func)
df.loc[1:99, "col1"].applymap(func)
```

### 2.2.8 Multiindexación

En una tabla es posible que como índice, no solamente haya una columna, sino más de una:

```
df.set_index(["col1", "col2 "])
```

Podemos ordenar de forma personalizada los diferentes índices:

```
df.sort__index(ascending = [True False])
```

Como podemos ver, hay un índices exterior y uno interior, este orden puede ser cambiado mediante el método:

```
df.swaplevel()
```

Se puede restablecer el índice de la siguiente forma:

```
df.reset_index(inplace = True)
```

El indexador loc puede ser usado de la siguiente manera para poder extraer o filtrar información necesaria de una tabla: si se desea filtrar ambos índices y luego ver una columna, los índices deben ser puestos en una tupla:

```
df.loc[(idx1,idx2), col]
```

Se puede no colocar ninguna columna y así veremos todas las columnas. Para seleccionar más de un valor para los índices, estos se colocan dentro de una lista. Para extraer todas las columnas, es necesario indicar los dos puntos:

```
df.loc([(idx1a,idx1b),idx2), col]  
df.loc([(idx1a,idx1b),idx2),:]
```

Para seleccionar todos los valores de un índice, la notación : produce un error, por tanto se debe usar slice(None):

```
df.loc([(idx1a,idx1b),slice(None)),:]
```

#### 2.2.8.1 Operaciones con cadenas de texto

Para todos los datos dentro de una tabla que sean cadenas de texto, si se requiere realizar cualquier operación, se debe implementar el método .str:

```
df.str.lower()
```

Para el método split, se tiene la opción de separar de manera automática el resultado en columnas separadas:

```
summer[col].str.split(", " n = 2, expand = True)
```

El parámetro n indica cuántos de los caracteres separadores se requiere tomar, y el parámetro expand indica si se requiere guardar el retorno en columnas separadas.

También es útil utilizar métodos de string como contains para realizar filtros personalizados y especializados:

```
summer[summer["Event"].str.contains("100M")]
```

## 2.3 Objeto DataFrame

Es una de las clases más importantes de Pandas; una estructura de datos tabular de dos dimensiones. Es una representación en Pandas de una tabla de datos.

### 2.3.1 Constructor

El constructor es el siguiente

```
class pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=None)
```

#### 2.3.1.1 Parámetros

##### data

- **Descripción:** Es el conjunto de datos que conformará el DataFrame.
- **Tipo:** `numpy.ndarray`, elementos iterables, `dict`, o `DataFrame`
- **Por defecto:** `None`

##### index

- **Descripción:** Es el conjunto de etiquetas de las filas de la estructura
- **Tipo:** `Index` o arreglo de índices (una lista de `str`).
- **Por defecto:** `None`

##### columns

- **Descripción:** Es el conjunto de etiquetas de las columnas de la estructura
- **Tipo:** `Index` o arreglo de índices (lista de `str`).
- **Por defecto:** `None`.

### 2.3.2 Atributos

La siguiente es la lista de los atributos de la clase `DataFrame`

#### 2.3.2.1 `T:`

La transposición del `DataFrame`.

#### 2.3.2.2 `at:`

Accede a un valor único para un par etiqueta de fila/columna.

#### 2.3.2.3 `attrs:`

Diccionario de atributos globales de este conjunto de datos.

#### 2.3.2.4 `axes:`

Devuelve una lista que representa los ejes del `DataFrame` (lista de `Index`).

#### 2.3.2.5 `columns[]:`

Las etiquetas de columna del `DataFrame`. Pueden indexarse como una lista.

#### 2.3.2.6 `dtypes:`

Devuelve los tipos de datos en el `DataFrame`.

#### 2.3.2.7 `empty:`

Indicador de si la Serie/`DataFrame` está vacío.

#### **2.3.2.8 flags:**

Obtiene las propiedades asociadas con este objeto pandas.

#### **2.3.2.9 iat:**

Accede a un valor único para un par fila/columna por posición entera.

#### **2.3.2.10 iloc:**

Indexación basada puramente en la ubicación por posición entera.

#### **2.3.2.11 index[:]:**

El índice (etiquetas de fila) del DataFrame. Puede indexarse como en una lista.

#### **2.3.2.12 loc:**

Accede a un grupo de filas y columnas por etiqueta(s) o un array booleano.

#### **2.3.2.13 ndim:**

Devuelve un entero que representa el número de ejes / dimensiones del array.

#### **2.3.2.14 shape:**

Devuelve una tupla que representa la dimensionalidad del DataFrame.

#### **2.3.2.15 size:**

Devuelve un entero que representa el número de elementos en este objeto.

#### **2.3.2.16 style:**

Devuelve un objeto Styler.

#### **2.3.2.17 values:**

Devuelve una representación en Numpy del DataFrame.

1. `pd.options.display.min_rows`
2. `pd.options.display.max_rows`

Estas variables se modifican a conveniencia para mostrar una cantidad mínima y máxima requerida de los datos de la tabla.

### **2.3.3 Métodos**

#### **2.3.3.1 abs()**

Devuelve una Serie/DataFrame con el valor numérico absoluto de cada elemento.

##### **parámetros**

- Ninguno

**Retorno:** `pandas.Series` o `Pandas.DataaFrame`

#### **2.3.3.2 add(other, axis, level, fill.value)**

Obtiene la suma de los elementos (elemento a elemento) del dataframe con otro dataframe (u objeto similar).

## parámetros

- other
  - **Descripción:** Objeto con el cual se realiza la suma.
  - **Tipo:** Serie, DataFrame, dict o cualquier escalar o secuencia.
  - **Por defecto:** Ninguno
- axis
  - **Descripción:** Selecciona si comparar por el índice o por la columna.
  - **Tipo:** 0 o 'index', 1 o 'column'
  - **Por defecto:** 'columns'
- fill\_value
  - **Descripción:** Rellenar valores que potencialmente serán no determinados (NaN). Esto es especialmente útil cuando se usan dos tablas con diferentes índices y/o columnas.
  - **Tipo:** float o None
  - **Por defecto:** None

**Retorno:** DataFrame con los resultados de los valores

### 2.3.3.3 add\_prefix(prefix, axis)

Añade un prefijo a las columnas. Si la columna es 'a' y el prefijo dado es 'col\_', la columna queda etiquetada como col\_a. No modifica el objeto (retorna uno nuevo)

## parámetros

- prefix
  - **Descripción:** La cadena de texto para prefijar.
  - **Tipo:** String
  - **Por defecto:** Ninguno
- axis
  - **Descripción:** Selecciona el eje al cual añadir el prefijo.
  - **Tipo:** 0 o 'index', 1 o 'column'
  - **Por defecto:** None

**Retorno:** DataFrame

### 2.3.3.4 add\_suffix(suffix, axis)

Añade un sufijo a las columnas. Análogo al método anterior.

## parámetros

- suffix
  - **Descripción:** La cadena de texto a añadir.
  - **Tipo:** String
  - **Por defecto:** Ninguno
- axis
  - **Descripción:** Selecciona el eje al cual añadir el sufijo.
  - **Tipo:** 0 o 'index', 1 o 'column'
  - **Por defecto:** None

**Retorno:** DataFrame

#### 2.3.3.5 agg(func, axis)

Agrega utilizando una o más operaciones sobre el eje especificado.

##### parámetros

- func
  - **Descripción:** La función o funciones para aplicar.
  - **Tipo:** Función o lista de funciones (como ejemplo 'sum')
  - **Por defecto:** Ninguno
- axis
  - **Descripción:** Eje sobre la cual realizar la función, si esta última está definida para listas.
  - **Tipo:** 0 o 'index', 1 o 'column'
  - **Por defecto:** 0

**Retorno:** DataFrame, Series o puede ser escalar, dependiendo de la función usada.

#### 2.3.3.6 align(other, join, axis, level, copy, fill.value)

Alinea dos objetos en sus ejes con el método de unión especificado. El método de unión está especificado para cada índice de eje.

##### Parámetros

- other
  - **Descripción:** DataFrame o Series con el que se quiere alinear.
  - **Tipo:** DataFrame o Series
  - **Por defecto:** Ninguno
- join
  - **Descripción:** Tipo de alineación a realizar ('outer', 'inner', 'left', 'right').
  - **Tipo:** Cadena de texto
  - **Por defecto:** 'outer'
- axis
  - **Descripción:** Eje permitido del otro objeto.
  - **Tipo:** 0, 1, o None
  - **Por defecto:** None
- level
  - **Descripción:** Difunde a través de un nivel, coincidiendo con los valores del índice en el nivel MultiIndex pasado.
  - **Tipo:** Entero o nombre de nivel
  - **Por defecto:** None
- copy
  - **Descripción:** Siempre devuelve nuevos objetos. Si copy=False y no se requiere reindexación, se devuelven los objetos originales.
  - **Tipo:** Booleano

- **Por defecto:** True
- **fill\_value**
  - **Descripción:** Valor a utilizar para valores faltantes. Por defecto es NaN, pero puede ser cualquier valor "compatible".
  - **Tipo:** Escalar
  - **Por defecto:** np.nan

**Retorno:** Tupla de (Series/DataFrame, tipo del otro objeto)

- **Descripción:** Objetos alineados.
- **Tipo:** Tupla

### 2.3.3.7 all(axis, bool\_only, skipna, \*\*kwargs)

Devuelve si todos los elementos son True, potencialmente a lo largo de un eje. Devuelve True a menos que haya al menos un elemento dentro de una serie o a lo largo de un eje de DataFrame que sea False o equivalente (p. ej., cero o vacío).

#### Parámetros

- **axis**
  - **Descripción:** Indica qué eje o ejes deben reducirse. Para Series, este parámetro no se utiliza y se establece en 0 por defecto.
  - **Tipo:** 0 o 'index', 1 o 'columns', None
  - **Por defecto:** 0
- **bool\_only**
  - **Descripción:** Incluye solo columnas booleanas. No implementado para Series.
  - **Tipo:** Booleano
  - **Por defecto:** False
- **skipna**
  - **Descripción:** Excluye valores NA/nulos. Si toda la fila/columna es NA y skipna es True, entonces el resultado será True, como en el caso de una fila/columna vacía. Si skipna es False, entonces los NA se tratan como True, porque no son iguales a cero.
  - **Tipo:** Booleano
  - **Por defecto:** True
- **\*\*kwargs**
  - **Descripción:** Palabras clave adicionales no tienen efecto pero podrían ser aceptadas para compatibilidad con NumPy.
  - **Tipo:** Cualquiera
  - **Por defecto:** None

**Retorno:** Series o DataFrame

- **Descripción:** Si se especifica un nivel, se devuelve un DataFrame; de lo contrario, se devuelve una Series.
- **Tipo:** Series o DataFrame



### 2.3.3.8 DataFrame.any(\*, axis=0, bool\_only=False, skipna=True, \*\*kwargs)

Devuelve si algún elemento es True, potencialmente a lo largo de un eje. Devuelve False a menos que haya al menos un elemento dentro de una serie o a lo largo de un eje de DataFrame que sea True o equivalente (p.ej., no cero o no vacío).

#### Parámetros

- **axis**
  - **Descripción:** Indica qué eje o ejes deben reducirse. Para Series, este parámetro no se utiliza y se establece en 0 por defecto.
  - **Tipo:** 0 o 'index', 1 o 'columns', None
  - **Por defecto:** 0
- **bool\_only**
  - **Descripción:** Incluye solo columnas booleanas. No implementado para Series.
  - **Tipo:** Booleano
  - **Por defecto:** False
- **skipna**
  - **Descripción:** Excluye valores NA/nulos. Si toda la fila/columna es NA y skipna es True, entonces el resultado será False, como en el caso de una fila/columna vacía. Si skipna es False, entonces los NA se tratan como True, porque no son iguales a cero.
  - **Tipo:** Booleano
  - **Por defecto:** True
- **\*\*kwargs**
  - **Descripción:** Palabras clave adicionales no tienen efecto pero podrían ser aceptadas para compatibilidad con NumPy.
  - **Tipo:** Cualquiera
  - **Por defecto:** None

**Retorno:** Series o DataFrame

- **Descripción:** Si se especifica un nivel, se devuelve un DataFrame; de lo contrario, se devuelve una Series.
- **Tipo:** Series o DataFrame

### 2.3.3.9 head(n)

Esta función retorna las primeras  $n$  filas del objeto basado en su posición. Es útil para comprobar rápidamente si el objeto contiene el tipo correcto de datos.

Para valores negativos de  $n$ , esta función devuelve todas las filas excepto las últimas  $|n|$  filas, equivalente a `df[: n]`.

Si  $n$  es mayor que el número de filas, esta función retorna todas las filas.

#### Parámetros

- **n**
  - **Descripción:** Número de filas a seleccionar.
  - **Tipo:** Entero
  - **Por defecto:** 5

**Retorno:** Mismo tipo que el objeto que llama

- **Descripción:** Las primeras  $n$  filas del objeto que llama la función.
- **Tipo:** Mismo tipo que el objeto que llama

**Tipo de dato de salida** DataFrame

#### 2.3.3.10 tail(n=5)

Esta función retorna las últimas  $n$  filas del objeto basado en su posición. Es útil para verificar rápidamente los datos, por ejemplo, después de ordenar o agregar filas.

Para valores negativos de  $n$ , esta función devuelve todas las filas excepto las primeras  $|n|$  filas, equivalente a `df[|n|:]`.

Si  $n$  es mayor que el número de filas, esta función retorna todas las filas.

#### Parámetros

- $n$ 
  - **Descripción:** Número de filas a seleccionar.
  - **Tipo:** Entero
  - **Por defecto:** 5

**Retorno:** Tipo del objeto que llama la función

- **Descripción:** Las últimas  $n$  filas del objeto que llama la función.
- **Tipo:** Tipo del objeto que llama la función

#### 2.3.3.11 info(verbose, buf, max\_cols, memory\_usage, show\_counts)

Este método imprime información sobre un DataFrame incluyendo el tipo de datos del índice y las columnas, los valores no nulos y el uso de memoria.

#### Parámetros

- `verbose`
  - **Descripción:** Si se imprime el resumen completo. Por defecto, se sigue la configuración en `pandas.options.display.max_info_columns`.
  - **Tipo:** Booleano, opcional
- `buf`
  - **Descripción:** Dónde enviar la salida. Por defecto, la salida se imprime en `sys.stdout`. Se puede pasar un búfer escribible si se necesita procesar más la salida.
  - **Tipo:** Búfer escribible, valor por defecto `sys.stdout`
- `max_cols`
  - **Descripción:** Cuándo cambiar de la salida detallada a la salida truncada. Si el DataFrame tiene más de `max_cols` columnas, se utiliza la salida truncada.
  - **Tipo:** Entero, opcional
- `memory_usage`
  - **Descripción:** Especifica si se debe mostrar el uso total de memoria de los elementos del DataFrame (incluido el índice).
  - **Tipo:** Booleano, cadena, opcional

- **show\_counts**
  - **Descripción:** Si se muestran los recuentos de no nulos. Por defecto, esto se muestra solo si el DataFrame es más pequeño que las opciones en `pandas.options.display.max_info_rows` y `pandas.options.display.max_info_columns`.
  - **Tipo:** Booleano, opcional

**Retorno:** Ninguno

- **Descripción:** Este método imprime un resumen de un DataFrame y no devuelve nada.
- **Tipo:** Ninguno

### 2.3.3.12 `describe(percentiles, include, exclude)`

Genera estadísticas descriptivas. Las estadísticas descriptivas incluyen aquellas que resumen la tendencia central, la dispersión y la forma de la distribución del conjunto de datos, excluyendo los valores NaN. Analiza tanto series numéricas como de objeto, así como conjuntos de columnas de DataFrame de tipos de datos mixtos. La salida variará dependiendo de lo que se proporcione. Consulte las notas a continuación para obtener más detalles.

#### Parámetros

- **percentiles**
  - **Descripción:** Los percentiles a incluir en la salida. Todos deben estar entre 0 y 1. El valor por defecto es `[.25, .5, .75]`, que devuelve los percentiles 25, 50 y 75.
  - **Tipo:** Lista de números
  - **Por defecto:** Ninguno (Opcional)
- **include**
  - **Descripción:** Lista blanca de tipos de datos para incluir en el resultado. Ignorado para Series.
  - **Tipo:** 'all', lista de tipos de datos o Ninguno (Por defecto)
  - **Opciones:** 'all', lista de dtypes, Ninguno
- **exclude**
  - **Descripción:** Lista negra de tipos de datos para omitir del resultado. Ignorado para Series.
  - **Tipo:** Lista de tipos de datos o Ninguno (Por defecto)
  - **Opciones:** Lista de dtypes, Ninguno

**Retorno:** Series o DataFrame

- **Descripción:** Estadísticas resumidas de la Series o DataFrame proporcionado.
- **Tipo:** Series o DataFrame

### 2.3.3.13 `len(df)`

Retorna el número de filas que hay en la tabla.

**Tipo de dato de salida** Int

**`round(df, 0)`** Retorna la tabla con los valores numéricos redondeados con las cifras especificadas.

**Tipo de dato de salida** Int

### 2.3.3.14 `mean(axis, skipna, numeric_only, **kwargs)`

Calcula la media de los valores a lo largo del eje solicitado.

#### Parámetros

- `axis`
  - **Descripción:** Eje sobre el que se aplica la función. Para Series, este parámetro no se utiliza y su valor predeterminado es 0. Para DataFrames, especificar `axis=None` aplicará la agregación en ambos ejes.
  - **Tipo:** {0 ('index'), 1 ('columns')}
  - **Por defecto:** 0
- `skipna`
  - **Descripción:** Excluir valores NA/nulos al calcular el resultado.
  - **Tipo:** Booleano
  - **Por defecto:** True
- `numeric_only`
  - **Descripción:** Incluir solo columnas de tipo float, int, booleano. No implementado para Series.
  - **Tipo:** Booleano
  - **Por defecto:** False
- `**kwargs`
  - **Descripción:** Argumentos de palabras clave adicionales para pasar a la función.
  - **Tipo:** Cualquiera

**Retorno:** Series o escalar

- **Descripción:** La media de los valores a lo largo del eje especificado.
- **Tipo:** Series o escalar

### 2.3.3.15 `sort_values(by, *, axis, ascending, inplace, kind, na_position, ignore_index, key)`

Ordena por los valores a lo largo de cualquiera de los ejes.

#### Parámetros

- `by`
  - **Descripción:** Nombre o lista de nombres (índices) por los que ordenar.
  - **Tipo:** Cadena de caracteres o lista de cadenas de caracteres, según el índice.
- `axis`
  - **Descripción:** Eje que será ordenado.
  - **Tipo:** {0 ('index'), 1 ('columns')}
  - **Por defecto:** 0
- `ascending`
  - **Descripción:** Ordenar de forma ascendente vs descendente. Especifique una lista para múltiples órdenes de clasificación.
  - **Tipo:** Booleano o lista de booleanos

- **Por defecto:** Verdadero
- **inplace**
  - **Descripción:** Si es Verdadero, aplica el ordenamiento al objeto.
  - **Tipo:** Booleano
  - **Por defecto:** Falso
- **kind**
  - **Descripción:** Elección del algoritmo de ordenación.
  - **Tipo:** {'quicksort', 'mergesort', 'heapsort', 'stable'}
  - **Por defecto:** 'quicksort'
- **na\_position**
  - **Descripción:** Ubica los NaN al principio si es 'first'; al final si es 'last'.
  - **Tipo:** {'first', 'last'}
  - **Por defecto:** 'last'
- **ignore\_index**
  - **Descripción:** Si es Verdadero, el eje resultante se etiquetará 0, 1, ..., n - 1.
  - **Tipo:** Booleano
  - **Por defecto:** Falso
- **key**
  - **Descripción:** Aplicar la función clave a los valores antes de ordenar.
  - **Tipo:** Función
  - **Opcional:** Verdadero

**Retorno:** DataFrame o None

- **Descripción:** DataFrame con valores ordenados o None si `inplace=True`.
- **Tipo:** DataFrame o None

### 2.3.3.16 Selección de columnas

Si se tiene un DataFrame, podemos usar el índice `[]` para seleccionar una o más columnas diferentes.

```
df["col1"]
```

Una forma alternativa es mediante la notación de punto:

```
df.col1
```

Si queremos obtener más de una columna, debemos ingresar las etiquetas de dichas columnas como una **lista**:

```
df[["col1", "col2"]]
```

**Tipo de dato de salida** Si se pone solamente la etiqueta de una columna como argumento, el tipo de retorno será **Series**. Si se pone una lista de una o más etiquetas, entonces la salida será otro **DataFrame**.

### 2.3.3.17 df.iloc[n,m]

Este método retorna la información de la fila enésima y columna enésima. Se pueden seleccionar una o más filas/columnas, y si se requiere un conjunto específico de filas/columnas, deben ingresarse dentro de una lista

```
df.iloc[25:30]
df.iloc[25,2]
df.iloc[25:30,2:5]
df.iloc[[100,345,778],[0,4]]
```

**Tipo de dato de salida** Series, DataFrame o el tipo de objeto en la celda, cuando se especifica una sola celda.

### 2.3.3.18 df.loc[Rlabel,Clabel]

Este método es muy similar al anterior, con la diferencia de que se asigna como argumento el texto de la etiqueta de la fila que se desea retornar. Si hay más de una fila con la misma etiqueta, entonces se retornan todas las filas.

**Tipo de dato de salida** Series, DataFrame o el tipo de objeto en la celda, cuando se especifica una sola celda.

Si se intenta capturar los datos de varias filas hasta una etiqueta especificada (por ejemplo, `df.loc[: "Clabel"]`) mostrará error si esta etiqueta no es única.

En general se puede acceder a las filas o columnas de un conjunto de datos pensando en la entrada de indexación como listas. Si por ejemplo se requiere un rango específico de filas más unas filas específicas, se puede usar lo siguiente

```
index = list(range(15,20)) + [35,45]
df.iloc[index]
```

O si se requieren, por ejemplo las tres primeras columnas, más dos columnas específicas más:

```
col_index = df.columns[:3].tolist() + ["col7","col9"]
df.loc[:, col_index]
```

### 2.3.3.19 copy(deep)

Realiza una copia de los índices y datos de este objeto.

**Descripción** Cuando `deep=True` (por defecto), se crea un nuevo objeto con una copia de los datos e índices del objeto que hace la llamada. Las modificaciones en los datos o índices de la copia no se reflejarán en el objeto original.

Cuando `deep=False`, se crea un nuevo objeto sin copiar los datos o el índice del objeto que hace la llamada (sólo se copian las referencias a los datos y el índice). Cualquier cambio en los datos del original se reflejará en la copia superficial (y viceversa).

#### Parámetros

- **deep**
  - **Descripción:** Realizar una copia profunda, incluyendo una copia de los datos y los índices. Con `deep=False` ni los índices ni los datos se copian.
  - **Tipo:** Booleano
  - **Por defecto:** Verdadero

**Retorno: Series o DataFrame**

- **Descripción:** El tipo de objeto coincide con el que hace la llamada.
- **Tipo:** Series o DataFrame

#### 2.3.3.20 `nlargest(n, columns, keep)`

Devuelve las primeras  $n$  filas ordenadas por las columnas en orden descendente.

**Descripción** Devuelve las primeras  $n$  filas con los valores más grandes en las columnas, en orden descendente. Las columnas que no se especifican también se devuelven, pero no se utilizan para ordenar. Este método es equivalente a `df.sort_values(columns, ascending=False).head(n)`, pero es más eficiente en términos de rendimiento.

**Parámetros**

- `n`
  - **Descripción:** Número de filas a devolver.
  - **Tipo:** Entero
- `columns`
  - **Descripción:** Etiqueta(s) de columna por las cuales ordenar.
  - **Tipo:** Etiqueta o lista de etiquetas
- `keep`
  - **Descripción:** Donde hay valores duplicados:
    - \* `first`: da prioridad a la primera ocurrencia(s).
    - \* `last`: da prioridad a la última ocurrencia(s).
    - \* `all`: no elimina ningún duplicado, incluso si eso significa seleccionar más de  $n$  elementos.
  - **Tipo:** {'first', 'last', 'all'}
  - **Por defecto:** 'first'

**Retorno: DataFrame**

- **Descripción:** Las primeras  $n$  filas ordenadas por las columnas dadas en orden descendente.
- **Tipo:** DataFrame

#### 2.3.3.21 `nsmallest(n, columns, keep)`

Devuelve las primeras  $n$  filas ordenadas por las columnas en orden ascendente.

**Descripción** Devuelve las primeras  $n$  filas con los valores más pequeños en las columnas, en orden ascendente. Las columnas que no se especifican también se devuelven, pero no se utilizan para ordenar.

Este método es equivalente a `df.sort_values(columns, ascending=True).head(n)`, pero es más eficiente en términos de rendimiento.

## Parámetros

- **n**
  - **Descripción:** Número de elementos a recuperar.
  - **Tipo:** Entero
- **columns**
  - **Descripción:** Nombre o nombres de columna por los cuales ordenar.
  - **Tipo:** Lista o cadena de caracteres
- **keep**
  - **Descripción:** Si hay valores duplicados:
    - \* **first:** toma la primera ocurrencia.
    - \* **last:** toma la última ocurrencia.
    - \* **all:** no elimina ningún duplicado, incluso si eso significa seleccionar más de  $n$  elementos.
  - **Tipo:** {'first', 'last', 'all'}
  - **Por defecto:** 'first'

## Retorno: DataFrame

- **Descripción:** Las primeras  $n$  filas ordenadas por las columnas dadas en orden ascendente.
- **Tipo:** DataFrame

### 2.3.3.22 idxmin(axis, skipna, numeric\_only)

Devuelve el índice de la primera ocurrencia del mínimo a lo largo del eje solicitado. Los valores NA/nulos son excluidos.

**Descripción** Esta función retorna el índice de la primera aparición del valor mínimo a lo largo del eje especificado, excluyendo los valores NA/nulos.

## Parámetros

- **axis**
  - **Descripción:** El eje a utilizar. 0 o 'index' para filas, 1 o 'columns' para columnas.
  - **Tipo:** {0 o 'index', 1 o 'columns'}
  - **Por defecto:** 0
- **skipna**
  - **Descripción:** Excluir valores NA/nulos. Si una fila/columna entera es NA, el resultado será NA.
  - **Tipo:** Booleano
  - **Por defecto:** Verdadero
- **numeric\_only**
  - **Descripción:** Incluir solo datos de tipo flotante, entero o booleano.
  - **Tipo:** Booleano
  - **Por defecto:** Falso

## Retorno: Series

- **Descripción:** Índices de los mínimos a lo largo del eje especificado.
- **Tipo:** Series



## Excepciones

- **ValueError**
  - **Descripción:** Se levanta si la fila o columna está vacía.

## `DataFrame.reset_index`

**Descripción:** Restablece el índice del DataFrame, y utiliza el índice predeterminado en su lugar. Si el DataFrame tiene un MultiIndex, este método puede eliminar uno o más niveles.

### Parámetros:

- **level**
  - **Descripción:** Solo elimina los niveles dados del índice. Elimina todos los niveles por defecto.
  - **Tipo:** int, str, tuple, o list
  - **Por defecto:** None
- **drop**
  - **Descripción:** Si se activa, quita completamente la columna de índices que estaba anteriormente, y la borra.
  - **Tipo:** bool
  - **Por defecto:** False
- **inplace**
  - **Descripción:** Si se debe modificar el DataFrame en lugar de crear uno nuevo.
  - **Tipo:** bool
  - **Por defecto:** False
- **col\_level**
  - **Descripción:** Si las columnas tienen múltiples niveles, determina en qué nivel se insertan las etiquetas. Por defecto, se insertan en el primer nivel.
  - **Tipo:** int o str
  - **Por defecto:** 0
- **col\_fill**
  - **Descripción:** Si las columnas tienen múltiples niveles, determina cómo se nombran los otros niveles.
  - **Tipo:** object
  - **Por defecto:** ''
- **allow\_duplicates**
  - **Descripción:** Permite la creación de etiquetas de columna duplicadas.
  - **Tipo:** bool, opcional
  - **Por defecto:** lib.no\_default
- **names**
  - **Descripción:** Utilizando la cadena dada, renombra la columna del DataFrame que contiene los datos del índice.
  - **Tipo:** int, str o lista 1-dimensional
  - **Por defecto:** None

**Tipo de retorno** DataFrame con el nuevo índice o None si inplace=True.

### 2.3.3.23 DataFrame.set\_index

**Descripción:** Establece el índice del DataFrame utilizando una o más columnas existentes o arrays del mismo tamaño. El índice puede reemplazar el índice existente o expandirse sobre él.

**Parámetros:**

- keys
  - **Descripción:** Este parámetro puede ser una clave de columna única, un array del mismo tamaño que el DataFrame que llama al método, o una lista que contiene una combinación arbitraria de claves de columna y arrays.
  - **Tipo:** Etiqueta o array-like o lista de etiquetas/arrays
  - **Por defecto:** No aplica
- drop
  - **Descripción:** Elimina las columnas que se utilizarán como el nuevo índice.
  - **Tipo:** bool
  - **Por defecto:** True
- append
  - **Descripción:** Si se deben agregar columnas al índice existente.
  - **Tipo:** bool
  - **Por defecto:** False
- inplace
  - **Descripción:** Si se debe modificar el DataFrame en lugar de crear uno nuevo.
  - **Tipo:** bool
  - **Por defecto:** False
- verify\_integrity
  - **Descripción:** Verifica el nuevo índice en busca de duplicados. De lo contrario, aplaza la verificación hasta que sea necesario.
  - **Tipo:** bool
  - **Por defecto:** False

**Tipo de retorno** DataFrame con las nuevas etiquetas de fila o None si inplace=True.

### 2.3.3.24 DataFrame.rename

**Descripción:** Renombra las etiquetas de columnas o índices. Los valores de la función o diccionario deben ser únicos (1-a-1). Las etiquetas no contenidas en un diccionario o Serie se mantendrán como están. Las etiquetas adicionales listadas no generarán un error.

## Parámetros:

- **mapper**
  - **Descripción:** Transformaciones dict-like o función para aplicar a los valores del eje especificado.
  - **Tipo:** dict-like o función
  - **Por defecto:** None
- **index**
  - **Descripción:** Alternativa para especificar el eje.
  - **Tipo:** dict-like o función
  - **Por defecto:** None
- **columns**
  - **Descripción:** Alternativa para especificar el eje.
  - **Tipo:** dict-like o función
  - **Por defecto:** None
- **axis**
  - **Descripción:** Eje para aplicar el renombramiento.
  - **Tipo:** {0 o 'index', 1 o 'columns'}
  - **Por defecto:** 0
- **copy**
  - **Descripción:** También copia los datos subyacentes.
  - **Tipo:** bool
  - **Por defecto:** True
- **inplace**
  - **Descripción:** Si se debe modificar el DataFrame en lugar de crear uno nuevo.
  - **Tipo:** bool
  - **Por defecto:** False
- **level**
  - **Descripción:** En caso de un MultiIndex, solo renombra las etiquetas en el nivel especificado.
  - **Tipo:** int o nombre de nivel
  - **Por defecto:** None
- **errors**
  - **Descripción:** Si 'raise', genera un KeyError cuando un mapper dict-like, index, o columns contiene etiquetas que no están presentes en el índice que se transforma. Si 'ignore', las claves existentes serán renombradas y las claves adicionales serán ignoradas.
  - **Tipo:** {'ignore', 'raise'}
  - **Por defecto:** 'ignore'

**Tipo de retorno** DataFrame con las etiquetas de eje renombradas o None si inplace=True.

## 2.4 Objeto *Series*

Cuando extraemos una columna entera con sus respectivas filas de un conjunto de datos, el resultado es del tipo `pandas.core.series.Series`. Este objeto se define dentro del contexto de Pandas como un arreglo unidimensional etiquetado.

En este objeto hay algunos métodos que son los mismos para los `DataFrame`

- `head()`
- `tail()`
- `dtype`
- `shape`
- `index`

### 2.4.1 Métodos y atributos

#### 2.4.1.1 `to_frame()`

Convierte la serie en un dataframe. Su argumento de entrada. Su único parámetro de entrada es la serie a convertir, y su retorno es un objeto de l tipo `DataFrame`.

Existen dos tipos principales de datos en una serie de datos: numéricos y no numéricos. En función del tipo de datos existen diferentes métodos de análisis de las series. Los siguientes son métodos para tipos numéricos.

#### 2.4.1.2 `Series.describe()`

Este método genera una descripción estadística de los datos.

##### Parámetros

- **percentiles:**
  - **Tipo:** list-like of numbers
  - **Descripción:** Los percentiles a incluir en la salida. Deben estar todos entre 0 y 1.
  - **Por defecto:** `[.25, .5, .75]`
- **include:**
  - **Tipo:** 'all', list-like of dtypes o None
  - **Descripción:** Tipo de datos a describir. Si es 'all', se describen todos los tipos.
  - **Por defecto:** None
- **exclude:**
  - **Tipo:** list-like of dtypes o None
  - **Descripción:** Tipo de datos a excluir de la descripción.
  - **Por defecto:** None
- **datetime.is\_numeric:**
  - **Tipo:** bool
  - **Descripción:** Si es True, se tratan las fechas como datos numéricos y se muestra un resumen numérico. Si es False, se muestra un resumen basado en fecha.
  - **Por defecto:** False

**Retorno** `Series` o `DataFrame` Con la descripción estadística de la Serie.

#### 2.4.1.3 `Series.count()`

Cuenta la cantidad de datos no nulos. El tipo de salida es entero.

#### 2.4.1.4 Series.sum()

Retorna la suma de los elementos en la serie de datos. Como característica particular de Pandas, puede manejar de manera apropiada con datos incompletos, o faltantes.

##### Parámetros

- **axis:**
  - **Tipo:** int
  - **Descripción:** El eje a lo largo del cual se aplicará la operación. Para Series, solo se permite el valor 0.
  - **Por defecto:** 0
- **skipna:**
  - **Tipo:** bool
  - **Descripción:** Especifica si se deben excluir los valores NaN al calcular la suma.
  - **Por defecto:** True
- **level:**
  - **Tipo:** int o str
  - **Descripción:** Si la Serie tiene un MultiIndex, especifica el nivel para calcular la suma.
  - **Por defecto:** None
- **min\_count:**
  - **Tipo:** int
  - **Descripción:** El número mínimo de observaciones válidas (no-NaN) requeridas para realizar la suma. Si no se alcanza este número, se devuelve NaN.
  - **Por defecto:** 0

**Retorno:** scalar Suma de los valores de la Serie.

#### 2.4.1.5 pandas.Series.mean()

Devuelve la media aritmética de los valores de la Serie.

##### Parámetros:

- **axis:**
  - **Tipo:** int
  - **Por defecto:** 0
- **skipna:**
  - **Tipo:** bool
  - **Por defecto:** True

#### 2.4.1.6 pandas.Series.median()

Devuelve la mediana de los valores de la Serie.

##### Parámetros:

- **axis, skipna, numeric\_only:** Igual que en Series.mean().

#### 2.4.1.7 `pandas.Series.std()`

Devuelve la desviación estándar de los valores de la Serie.

##### Parámetros:

- **axis, skipna, level:** Igual que en `Series.mean()`.
- **ddof**
  - **Tipo:** `int`
  - **Por defecto:** `1`
  - **Descripción:** Delta de los grados de libertad.

#### 2.4.1.8 `pandas.Series.min()`

Devuelve el valor mínimo de la Serie.

##### Parámetros:

- **axis, skipna, numeric\_only:** Igual que en `Series.mean()`.

#### 2.4.1.9 `pandas.Series.max()`

Devuelve el valor máximo de la Serie.

##### Parámetros:

- **axis, skipna, numeric\_only:** Igual que en `Series.mean()`.

#### 2.4.1.10 `pandas.Series.unique()`

Devuelve los valores únicos de la Serie.

**Parámetros:** Ninguno.

**Tipo de retorno:** `ExtensionArray`

#### 2.4.1.11 `pandas.Series.value_counts()`

Devuelve una Serie que representa la frecuencia de valores únicos.

##### Parámetros:

- **normalize:**
  - **Tipo:** `bool`
  - **Descripción:** Si es `True`, devuelve las proporciones en lugar de las cuentas.
  - **Por defecto:** `False`
- **sort:**
  - **Tipo:** `bool`
  - **Por defecto:** `True`
- **ascending:**
  - **Tipo:** `bool`
  - **Por defecto:** `False`
- **bins:**

- **Tipo:** int
- **Descripción:** Solo para Series numéricas. Divide los valores en intervalos.
- **Por defecto:** None
- **dropna:**
  - **Tipo:** bool
  - **Descripción:** Si es True, excluye los valores NaN.
  - **Por defecto:** True

#### **Tipo de retorno:** Series

Para las series con valores no numéricos, los atributos y métodos como `shape`, `size`, `count()` son muy similares a los de valores numéricos. Por su parte otros métodos como `describe()` muestra información como cuál es el valor más repetido, su respectiva frecuencia, etc.

Métodos como `min()` que retorna el valor más "pequeño" alfabéticamente hablando, tienen sus diferencias con los valores numéricos.

#### **2.4.1.12 Ejemplo**

Para un conjunto de datos de las medallas de oro de los juegos olímpicos históricos de verano, el siguiente código realiza una exploración de cuál es el nombre del atleta que más oros consiguió en la edición de los juegos olímpicos de 1972:

```
import pandas as pd

df = pd.read_csv("summer1972.csv", index_col="Year")

Athlete = df["Athlete"]

print(Athlete.loc[1972].value_counts().index[0])
```

### **2.4.2 Creación de series**

A partir de cualquier conjunto de datos importado o creado, si seleccionamos o extraemos una de las columnas, sea total o filtrada de alguna manera, vamos a obtener un objeto del tipo *Series*. La siguiente línea de código es un ejemplo de cómo se consigue una serie directamente importada del archivo:

```
pd.read_csv("file.csv", usecols = ["name_col"]).squeeze()
```

De este modo, la variable retornada será del tipo *Series* en lugar de *DataFrame*. Por su parte, existe el método para convertir una lista de python a una serie de pandas:

```
pd.Series([10,25,6,36,2])
```

Se pueden establecer las etiquetas a las filas de la serie creada de la siguiente forma:

```
pd.Series([10,25,6,36,2], index = ["Mon","Tue","Wed","Thu","Fri"])
```

Y se puede añadir la etiqueta de la columna también:

```
pd.Series([10,25,6,36,2], index = ["Mon","Tue","Wed","Thu","Fri"], name = "Sales")
```

#### **2.4.2.1 pd.Series**

Retorna un objeto del tipo *Series* a partir de los datos ingresados.

## Parámetros

- **data:**
  - **Descripción:** los datos que serán los elementos de la serie.
  - **Tipo:** elementos iterables, del tipo arreglo (`numpy.array`) o diccionarios.
  - **Por defecto:** None
- **index:**
  - **Descripción:** Es el conjunto de etiquetas para las filas de la serie.
  - **Tipo:** Tipo arreglo, puede ser una lista con los nombres requeridos.
  - **Por defecto:** None
- **data:**
  - **Descripción:** los datos que serán los elementos de la serie.
  - **Tipo:** elementos iterables, del tipo arreglo (`numpy.array`) o diccionarios.
  - **Por defecto:** None

## Tipo de retorno : Series

Por otro lado, se puede crear una variable del tipo `Series` a partir de arreglos de *numpy*, una lista de python, tupla, o diccionario:

```
import pandas as pd
import numpy as np
sales = np.array([10,25,6,36,2])
pd.Series(sales)
sales = [10,25,6,36,2]
```

Es importante mencionar la diferencia entre las referencias de filas realizadas con `loc` y las de `iloc`. Esta última se refiere al **índice**, iniciando desde cero hasta la longitud menos uno. La primera, `loc` se refiere a la **etiqueta**, la cual puede variar.

### 2.4.2.2 sort\_index

**Descripción:** Ordena la Serie por las etiquetas del índice. Devuelve una nueva Serie ordenada por etiqueta si el argumento `inplace` es `False`, de lo contrario, actualiza la Serie original y devuelve `None`.

## Parámetros:

- **axis**
  - **Descripción:** No utilizado. Parámetro necesario para compatibilidad con `DataFrame`.
  - **Tipo:** {0 o 'index'}
  - **Por defecto:** 0
- **level**
  - **Descripción:** Si no es `None`, ordena los valores en el nivel de índice especificado.
  - **Tipo:** int, opcional
  - **Por defecto:** None
- **ascending**
  - **Descripción:** Ordena en dirección ascendente o descendente.
  - **Tipo:** bool o lista de bools
  - **Por defecto:** True



- **inplace**
  - **Descripción:** Si es True, realiza la operación en el lugar.
  - **Tipo:** bool
  - **Por defecto:** False
- **kind**
  - **Descripción:** Elección del algoritmo de ordenación.
  - **Tipo:** {'quicksort', 'mergesort', 'heapsort', 'stable'}
  - **Por defecto:** 'quicksort'
- **na\_position**
  - **Descripción:** Posición de los NaNs en la Serie ordenada.
  - **Tipo:** {'first', 'last'}
  - **Por defecto:** 'last'
- **sort\_remaining**
  - **Descripción:** Si es True, también ordena por otros niveles después de ordenar por el nivel especificado.
  - **Tipo:** bool
  - **Por defecto:** True
- **ignore\_index**
  - **Descripción:** Si es True, el eje resultante se etiquetará de 0 a  $n - 1$ .
  - **Tipo:** bool
  - **Por defecto:** False
- **key**
  - **Descripción:** Si no es None, aplica la función clave a los valores del índice antes de ordenar.
  - **Tipo:** callable, opcional
  - **Por defecto:** None

**Tipo de retorno** Serie o None. La Serie original ordenada por las etiquetas o None si `inplace=True`.

## 2.5 Objeto index

Uno de los atributos de un dataframe es `df.columns`, al imprimir este tipo de objeto vemos que su salida es del tipo

```
Index(['col1', 'col2', 'col3'], dtype='object')
```

Estas son instancias de la clase `pandas.index` y se trata de una secuencia inmutable que se utiliza para la indexación y el alineamiento de los datos. Es el objeto en el que se guardan las etiquetas de los ejes en todos los objetos de Pandas.

### 2.5.1 Métodos

#### 2.5.1.1 `Index.append`

**Descripción:** Agrega una colección de opciones de índice juntas.

**Parámetros:**

- other
  - **Descripción:** Índice o lista/tupla de índices a agregar.
  - **Tipo:** Index o list/tuple de índices
  - **Por defecto:** No aplica

**Tipo de retorno** Índice (Index).

**2.5.1.2 Index.drop**

**Descripción:** Crea un nuevo índice con la lista de etiquetas pasadas eliminadas.

**Parámetros:**

- labels
  - **Descripción:** Etiquetas para eliminar del índice.
  - **Tipo:** array-like o escalar
  - **Por defecto:** No aplica
- errors
  - **Descripción:** Si es 'ignore', suprime el error y se eliminan las etiquetas existentes.
  - **Tipo:** {'ignore', 'raise'}
  - **Por defecto:** 'raise'

**Tipo de retorno** Índice (Index). Será del mismo tipo que el índice original, excepto para RangeIndex.

**Excepciones** KeyError si no se encuentran todas las etiquetas en el eje seleccionado.

**Index.isin**

**Descripción:** Devuelve un array booleano donde los valores del índice están en los valores pasados. Calcula un array booleano que indica si cada valor del índice se encuentra en el conjunto de valores pasado. La longitud del array booleano devuelto coincide con la longitud del índice.

**Parámetros:**

- values
  - **Descripción:** Valores buscados.
  - **Tipo:** conjunto o lista
  - **Por defecto:** No aplica
- level
  - **Descripción:** Nombre o posición del nivel del índice a utilizar (si el índice es un MultiIndex).
  - **Tipo:** cadena o entero, opcional
  - **Por defecto:** None

**Tipo de retorno** Array de NumPy de valores booleanos (np.ndarray[bool]).

**Notas** En el caso de MultiIndex, debe especificar los valores como un objeto similar a una lista que contenga tuplas que tengan la misma longitud que el número de niveles, o especificar el nivel. De lo contrario, se generará un ValueError.

### 2.5.1.3 `Index.unique`

**Descripción:** Devuelve los valores únicos en el índice. Los valores únicos se devuelven en el orden en que aparecen; esto no ordena los valores.

**Parámetros:**

- `level`
  - **Descripción:** Devuelve solo los valores del nivel especificado (para `MultiIndex`). Si es un entero, obtiene el nivel por posición entera; de lo contrario, por nombre del nivel.
  - **Tipo:** entero o hashable, opcional
  - **Por defecto:** `None`

**Tipo de retorno** Índice (`Index`).

**Ver también**

- `unique`: Array de NumPy de valores únicos en esa columna.
- `Series.unique`: Devuelve los valores únicos del objeto `Series`.

### 2.5.1.4 `Index.duplicated`

**Descripción:** Indica los valores duplicados del índice. Los valores duplicados se indican como valores `True` en el array resultante. Se pueden indicar todos los duplicados, todos excepto el primero o todos excepto el último.

**Parámetros:**

- `keep`
  - **Descripción:** El valor o valores en un conjunto de duplicados que se marcarán como faltantes.
  - **Tipo:** {'first', 'last', `False`}
  - **Por defecto:** 'first'

**Tipo de retorno** Array de NumPy de valores booleanos (`np.ndarray[bool]`).

**Ejemplos** Por defecto, para cada conjunto de valores duplicados, la primera aparición se establece en `False` y todas las demás en `True`. Al usar 'last', la última aparición de cada conjunto de valores duplicados se establece en `False` y todas las demás en `True`. Al establecer `keep` en `False`, todos los duplicados son `True`.

### 2.5.1.5 `Index.sort_values`

**Descripción:** Devuelve una copia ordenada del índice. Opcionalmente, devuelve los índices que ordenaron el índice en sí.

**Parámetros:**

- `return_indexer`
  - **Descripción:** ¿Deberían devolverse los índices que ordenarían el índice?
  - **Tipo:** `bool`
  - **Por defecto:** `False`
- `ascending`
  - **Descripción:** ¿Deberían ordenarse los valores del índice en orden ascendente?
  - **Tipo:** `bool`

- **Por defecto:** True
- `na_position`
  - **Descripción:** Argumento 'first' coloca NaNs al principio, 'last' coloca NaNs al final.
  - **Tipo:** {'first', 'last'}
  - **Por defecto:** 'last'
- `key`
  - **Descripción:** Si no es None, aplica la función clave a los valores del índice antes de ordenar.
  - **Tipo:** callable, opcional
  - **Por defecto:** None

**Tipo de retorno** Índice (`pandas.Index`) y opcionalmente un array de NumPy (`numpy.ndarray`).

#### 2.5.1.6 `Index.get_loc`

**Descripción:** Obtiene la ubicación entera, el segmento o la máscara booleana para la etiqueta solicitada.

**Parámetros:**

- `key`
  - **Descripción:** Etiqueta para la cual se desea obtener la ubicación.
  - **Tipo:** label
  - **Por defecto:** No aplica

**Tipo de retorno** Entero si el índice es único, segmento si el índice es monótono, de lo contrario, máscara.

#### 2.5.1.7 `Index.get_indexer`

**Descripción:** Calcula el indexador y la máscara para el nuevo índice dado el índice actual. El indexador debe usarse como entrada para `ndarray.take` para alinear los datos actuales con el nuevo índice.

**Parámetros:**

- `target`
  - **Descripción:** Índice objetivo.
  - **Tipo:** Index
  - **Por defecto:** No aplica
- `method`
  - **Descripción:** Método para encontrar coincidencias inexactas.
  - **Tipo:** {None, 'pad'/'ffill', 'backfill'/'bfill', 'nearest'}, opcional
  - **Por defecto:** Coincidencias exactas solamente
- `limit`
  - **Descripción:** Número máximo de etiquetas consecutivas en el objetivo para coincidencias inexactas.
  - **Tipo:** int, opcional
  - **Por defecto:** No aplica
- `tolerance`
  - **Descripción:** Distancia máxima entre las etiquetas originales y nuevas para coincidencias inexactas.
  - **Tipo:** opcional
  - **Por defecto:** No aplica

**Tipo de retorno** Array de NumPy de enteros (`np.ndarray[np.intp]`).

#### 2.5.1.8 `Index.intersection`

**Descripción:** Forma la intersección de dos objetos `Index`. Esto devuelve un nuevo `Index` con elementos comunes al índice y al otro objeto.

**Parámetros:**

- `other`
  - **Descripción:** Otro índice o objeto similar a un array con el que formar la intersección.
  - **Tipo:** `Index` o array-like
  - **Por defecto:** No aplica
- `sort`
  - **Descripción:** Si se debe ordenar el índice resultante.
  - **Tipo:** `True`, `False` o `None`
  - **Por defecto:** `False`

**Tipo de retorno** Índice (`Index`).

#### 2.5.1.9 `Index.tolist`

**Descripción:** Devuelve una lista de los valores. Cada uno de estos es un tipo escalar, que es un escalar de Python (para `str`, `int`, `float`) o un escalar de Pandas (para `Timestamp`/`Timedelta`/`Interval`/`Period`).

**Parámetros:** No tiene parámetros.

**Tipo de retorno** Lista de Python (`list`).

**Ver también**

- `numpy.ndarray.tolist`: Devuelve el array como una lista de Python anidada con `a.ndim` niveles de profundidad.

**Ejemplos** Para Series:

```
>>> s = pd.Series([1, 2, 3])
>>> s.tolist()
[1, 2, 3]
```

Para Index:

```
>>> idx = pd.Index([1, 2, 3])
>>> idx
Index([1, 2, 3], dtype='int64')
>>> idx.tolist()
[1, 2, 3]
```

## 2.5.2 Atributos

la lista de métodos es la siguiente:

### 2.5.2.1 **array:**

El `ExtensionArray` de los datos que respaldan esta Serie o Índice.

#### **2.5.2.2 dtype:**

Devuelve el objeto dtype de los datos subyacentes.

#### **2.5.2.3 has\_duplicates:**

Verifica si el Índice tiene valores duplicados.

#### **2.5.2.4 hasnans:**

Devuelve True si hay algún NaN.

#### **2.5.2.5 inferred\_type:**

Devuelve una cadena del tipo inferido a partir de los valores.

#### **2.5.2.6 is\_monotonic\_decreasing:**

Devuelve un booleano si los valores son iguales o decrecientes.

#### **2.5.2.7 is\_monotonic\_increasing:**

Devuelve un booleano si los valores son iguales o crecientes.

#### **2.5.2.8 is\_unique:**

Devuelve si el índice tiene valores únicos.

#### **2.5.2.9 name:**

Devuelve el nombre del Índice o MultiÍndice.

#### **2.5.2.10 nbytes:**

Devuelve el número de bytes en los datos subyacentes.

#### **2.5.2.11 ndim:**

Número de dimensiones de los datos subyacentes, por definición 1.

#### **2.5.2.12 nlevels:**

Número de niveles.

#### **2.5.2.13 shape:**

Devuelve una tupla de la forma de los datos subyacentes.

#### **2.5.2.14 size:**

Devuelve el número de elementos en los datos subyacentes.

#### **2.5.2.15 values:**

Devuelve una matriz que representa los datos en el Índice.

## Chapter 3

# Scrapping

### 3.1 Introducción al Scrapping

El web scraping es una técnica poderosa utilizada para extraer información de páginas web. Esta técnica emplea programas o scripts para simular la navegación de un usuario en la web, accediendo a páginas web y extrayendo los datos necesarios de ellas. La flexibilidad del web scraping permite su aplicación en una variedad de contextos, desde la recopilación de datos para análisis de mercado hasta la automatización de tareas de recolección de datos para investigación académica o desarrollo de productos.

Una característica notable del web scraping es su capacidad para interactuar tanto con páginas web estáticas como dinámicas. Las páginas estáticas son aquellas cuyo contenido no cambia con cada solicitud y generalmente se sirven directamente desde el servidor en formato HTML. Por otro lado, las páginas dinámicas son aquellas cuyos contenidos pueden cambiar en función de la interacción del usuario, los datos de entrada, o incluso el tiempo, y a menudo implican la carga de datos a través de APIs o tecnologías de frontend como JavaScript.

Además, el web scraping moderno puede manejar desafíos más complejos como el análisis y la interacción con APIs, así como la superación de CAPTCHAs, que son mecanismos diseñados para diferenciar entre usuarios humanos y bots automatizados. El manejo de CAPTCHAs es un área avanzada del web scraping, que a menudo implica técnicas sofisticadas y, en algunos casos, cuestiones éticas y legales.

En el corazón del web scraping se encuentran varias herramientas y tecnologías clave. Bibliotecas como BeautifulSoup (bs4) y Scrapy en Python son ampliamente utilizadas para parsear y navegar por el contenido de las páginas web. BeautifulSoup es particularmente conocida por su simplicidad y eficiencia en extraer datos de HTML y XML, mientras que Scrapy ofrece un marco más completo para la creación de spiders - programas que automatizan la navegación y extracción de datos de múltiples páginas.

La biblioteca requests en Python es otra herramienta esencial, que se utiliza para realizar solicitudes HTTP a servidores web. Esta biblioteca simplifica el proceso de envío de solicitudes y manejo de respuestas, lo cual es fundamental para acceder a los contenidos de las páginas web.

Un aspecto crucial del web scraping es el análisis y la selección de los datos específicos a extraer. Aquí es donde entra en juego XPath, un lenguaje que permite navegar a través de los elementos y atributos en los documentos HTML y XML. XPath proporciona una forma poderosa y flexible de identificar y extraer partes específicas de una página web, lo que facilita enormemente la tarea de localizar y recopilar los datos deseados.

En resumen, el web scraping es una técnica multifacética que se ha vuelto indispensable en el mundo del análisis de datos y la automatización. Con la ayuda de herramientas como BeautifulSoup, Scrapy, 'requests', y el uso de XPath para el análisis de HTML, los desarrolladores y analistas pueden extraer una gran cantidad de información de la web, abarcando desde páginas web estáticas sencillas hasta páginas dinámicas y APIs complejas. Sin embargo, es crucial tener en cuenta las implicaciones legales y éticas al realizar web scraping, asegurándose de respetar las políticas de uso de los sitios web y las leyes de protección de datos.

#### 3.1.1 XPath

XPath, que significa XML Path Language, es un lenguaje de consulta que se utiliza para seleccionar nodos de un documento XML. Este lenguaje ofrece una forma precisa y flexible de navegar y localizar partes específicas de un documento, lo cual es esencial en el contexto del procesamiento de XML y HTML, especialmente en tareas como el web scraping y la transformación de datos. XPath utiliza una sintaxis de ruta similar a las rutas

de archivo en sistemas operativos, lo que permite a los usuarios y desarrolladores especificar patrones para identificar nodos, atributos y valores dentro del documento XML.

La fuerza de XPath radica en su capacidad para realizar búsquedas tanto específicas como generales dentro de un documento. Por ejemplo, puede seleccionar todos los nodos que cumplen con cierto criterio, o puede enfocarse en un solo elemento con una identificación única. Además, XPath soporta funciones integradas para cadenas, números y lógica booleana, lo que añade una capa adicional de potencia y versatilidad a las consultas. Esta combinación de flexibilidad y precisión hace de XPath una herramienta indispensable en muchas aplicaciones que involucran XML, como la transformación de datos con XSLT, la configuración de ciertos frameworks de desarrollo de software y, más comúnmente, la extracción de datos específicos de páginas web en el web scraping. Con su enfoque en la estructura del documento y su capacidad para manejar consultas complejas, XPath se ha establecido como un estándar para la manipulación y consulta de documentos XML.

La siguiente es una lista de los comandos más utilizados en XPath:

- `/`: Selector de raíz, selecciona desde el nodo raíz.
- `//`: Selecciona nodos en todo el documento desde el nodo actual que coinciden con la selección.
- `.`: Selecciona el nodo actual.
- `..`: Selecciona el nodo padre del nodo actual.
- `@`: Selecciona atributos. Ejemplo: `@class`.
- `*`: Selecciona todos los nodos hijos del nodo actual.
- `node()`: Selecciona todos los tipos de nodos bajo el nodo actual.
- `[]`: Aplica un filtro, seleccionando elementos que cumplen el criterio dentro de los corchetes.
- `|`: Combina expresiones y devuelve la unión de sus resultados.
- `text()`: Selecciona todos los nodos de texto bajo el nodo actual.
- `@*[local-name()='nombre']`: Selecciona atributos con un nombre local específico.
- `[n]`: Selecciona el *n*-ésimo nodo en un conjunto.
- `[last()]`: Selecciona el último nodo en un conjunto.
- `[position()<n]`: Selecciona nodos en posiciones menores que *n*.
- `[contains(@atributo, 'texto')]`: Selecciona nodos con un atributo que contiene un texto dado.
- `[starts-with(@atributo, 'texto')]`: Selecciona nodos con un atributo que comienza con un texto dado.
- `[string-length(@atributo)>n]`: Selecciona nodos con la longitud de cadena de un atributo mayor que *n*.
- `[not(expresión)]`: Selecciona nodos que no cumplen con la expresión dada.



## Chapter 4

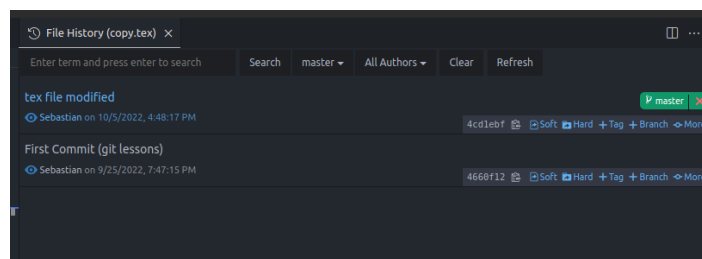
# Git and github

### 4.1 Algunas extensiones de VSCode para Git

#### 4.1.1 Git History

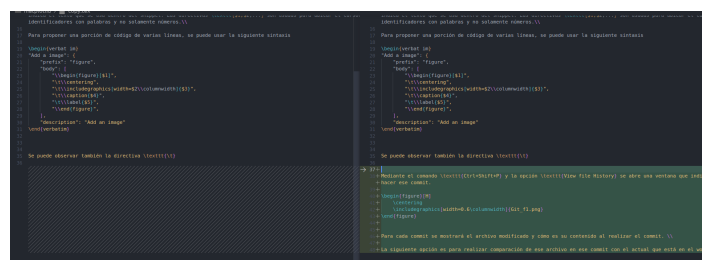
Es una extensión que sirve para visualizar los cambios históricos que se han hecho a los diferentes archivos.

Mediante el comando `Ctrl+Shift+P` y la opción `View file History` se abre una ventana que indica las versiones del proyecto (commits) y cómo han sido los archivos al momento de hacer la confirmación de esa versión.



Para cada versión se mostrará el archivo modificado y cómo es su contenido al realizar la confirmación.

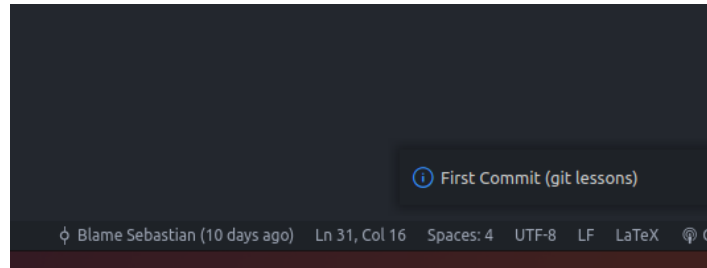
La siguiente opción es para realizar comparación de ese archivo en esa versión con la actual que está en el espacio de trabajo.



Las siguientes opciones permiten ver la comparación del archivo su versión anterior. Y la última opción permite ver la historia completa del archivo

#### 4.1.2 Git Blame

Esta extensión sirve para realizar revisión línea por línea de por quién, hace cuánto y en qué versión se realizó esa línea de código



### 4.1.3 Git Lens

Esta extensión es similar a las anteriores, en que ayuda a verificar las identidades de las personas que están modificando archivos, muestra línea por línea información del autor, y versión de la línea en cuestión.

## 4.2 Fundamentos de Git: cómo funciona por dentro

### 4.2.1 Crear un nuevo repositorio

La sentencia primaria y básica para inicializar un nuevo repositorio es

```
git init
```

Este comando se realiza dentro de la carpeta en la que se desea realizar el repositorio. Una vez creado el repositorio, se pueden añadir las carpetas correspondientes dentro. Al crearse el repositorio, se crea una carpeta oculta llamada `.git` de manera automática.

dentro de esta carpeta tenemos diferentes carpetas y archivos. Uno de ellos es el archivo `config`, en este archivo tenemos una serie de cadenas que nos indica las configuraciones que tiene el repositorio. El siguiente es un ejemplo de un archivo de configuración:

```
[core]
    repositoryformatversion = 0
    filemode = false
    bare = false
    logallrefupdates = true
    ignorecase = true
[remote "origin"]
    url = https://github.com/JhoAraSan/Process.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

Otro archivo es el de descripción `description`, el cual se puede editar para añadir una descripción al repositorio. Finalmente el archivo `HEAD`, el cual puede arrojar la siguiente cadena:

```
ref: refs/heads/master
```

Más adelante se dará la explicación correspondiente a este archivo.

Git tiene su propio sistema de archivos; en este sistema de archivos git guarda o almacena objetos, y estos objetos son guardados dentro de la carpeta correspondiente `objects`.

### 4.2.2 Objetos en Git

En git existen cuatro tipos de objetos:

```
Blob
Tree
Commit
Annotated Tag
```

Estos cuatro objetos son los suficientes para poder realizar todo el seguimiento de los archivos del repositorio.

#### 4.2.2.1 Blob

Este es el tipo de objeto en el que git guarda **archivos**. Todo tipo de archivos con la extensión que sea. Cualquier tipo de archivo serpa guardado como un blob.

#### 4.2.2.2 Tree

Este es el tipo de objeto en el que git almacena la información sobre los directorios. Dicho de otra fforma, un objeto Tree es una representación de una carpeta o un directorio.

#### 4.2.2.3 Commit

A través de este tipo de objeto, Git es capaz de almacenar diferentes versiones de uno o varios archivos a través del tiempo.

#### 4.2.2.4 Annotated Tag

Este objeto es esencialmente un texto que está apuntando a un commit versión específica.

Para poder gestionar o administrar objetos en git se usan los comandos de git de bajo nivel:

`git hash-object` Con esre comando podemos crear nuevos objetos con la estructura de git. `git cat-file` Con este comando se pueden leer los objetos git. `git mktree` con este comando se puede crear un nuevo objeto de tipo Tree

A manera de ejemplo, si colocamos un texto string cualquiera mediante el siguiente comando:

```
echo "Hello, Git" | git hash-object --stdin -w
```

Vamos a obtener como salida un hash, además de que se creará el objeto correspondiente en la carpeta `objects`; la carpeta será los dos primeros caracteres del hash, y dentro habrá un archivo con el resto de caracteres del hash. Solamente se crea el objeto, el repositorio seguirá estando vacío. Importante remarcar que el hash retornado es el hash del string que le metimos de entrada.

### 4.2.3 JSON

Las siglas significan "JavaScript Object Notation". Es un formato que permite el intercambio de datos entre diferentes servidores. Como ejemplo, podemos extraer datos mediante una API desde un servidor a una página web. La siguiente es un ejemplo de una estructura JSON:

```
{
  "id": "12345667",
  "name": "Mike",
  "age": 25,
  "city": "New York",
  "hobbies": ["Skate", "Running"]
}
```

Siempre será recomendado que las llaves en un archivo Json sean únicas. La estructura de datos que hay en git es muy similar a JSON; Git tambien almacena nombres "llave" y valores. Las "llaves" en git son los hashes de cada objeto. En git, el hash generado (el cual es equivalente a la key) es función o depende del valor.

#### 4.2.4 Hash

Al realizar el comando de la seccion anterior, vimos que `ek string "Hello, Git"` generó un hash `b7aec520dec0a7516c18eb4c`. Esto significa que se aplicó una funcion hash al string o al dato ingresado.

Una función hash es una función que toma una entrada de cualquier tamaño (longitud) y tiene una salida de un tamaño fijo. Es importante también notar que el hash es una función unidireccional, es decir, que si tenemos un hash generado no vamos a poder saber cuál fue la entrada que la generó. Para la misma función hash, la misma entrada siempre va a generar la misma salida.

Las funciones o algoritmos para generar hashes más importantes son las siguientes:

- MD5 (128 bit)
- SHA1 (160 bit)
- SHA256 (256 bit)
- SHA384 (384 bit)
- SHA512 (512 bit)

El algoritmo usado por git para generar sus hashes es SHA1.

Cada caracter de 4 bits de longitud está en formato hexadecimal. Por tanto, un hash de git tiene una longitud de 40 caracteres hexadecimales.

Dado lo anterior, surge la pregunta de cuantos archivos diferentes podemos guardar en el mismo repositorio.

La cantidad total de hashes diferentes es  $16^{40} \approx 1.46 \cdot 10^{48}$ . Por otro lado podemos hacernos la pregunta de cual es la posibilidad de que dos archivos diferentes produzcan el mismo hash?

La probabilidad de encontrar un hash específico es  $\frac{1}{16^{40}} \approx 6.84 \cdot 10^{-49}$ . Por tanto para saber la probabilidad de que dos archivos produzcan el mismo hash, tenemos

$$P = \frac{1}{16^{40}} \frac{1}{16^{40}} = \frac{1}{16^{80}} \approx 4.68 \cdot 10^{-97}$$

Como elemento adicional, tenemos que la probabilidad de que teniendo  $n$  archivos diferentes, dos de ellos generen el mismo hash, es el siguiente:

$$P = \frac{(2^{160} - n)!(2^{160})^{n-1} - (2^{160} - 1)!}{(2^{160} - n)! 2^{160(n-1)}}$$

Para que haya una probabilidad de 1 de que haya una colision de hash es necesario que en un repositorio hayan más archivos que número diferente de hashes.

#### 4.2.5 Exploración de objetos de git mediante cat

Recordemos que todo objeto de git tiene su correspondiente hash. Podemos usar el comando `git cat` para obtener información de cualquier objeto. Las opciones del comando son:

`git cat-file -p <hash>` Retorna el contenido del objeto.  
`git cat-file -t <hash>` Retorna el tamaño del objeto.  
`git cat-file -s <hash>` Retorna el tipo del objeto. El tamaño lo retorna en bytes.

#### 4.2.6 Creación de objetos mediante comandos de git

el comando `git hash-object` se usa para crear nuevos objetos, luego tenemos varias opciones adicionales:

```
echo "Hello, Git!" | git hash-object --stdin -w
```

La primera opción es para tomar la entrada como entrada estándar, la segunda es importante porque es la que hace se cree el objeto. También podemos crear objetos en git basados en archivos locales:

```
git hash-object <filename> -w
```

Una cosa importante de notar es que en git cuando almacenamos archivos del tipo blob, estos objetos no tienen un nombre de archivo. Como se podrá ver en los anteriores comandos, ninguno de ellos retorna el nombre del archivo puesto que este no se almacena. Otra cosa importante de notar es que tanto el tamaño como el tipo de objeto se almacenan dentro del mismo hash. La estructura con la que lo hace es la siguiente:

contenido + tipo de objeto + tamaño del objeto = hash. Entre el tipo de objeto más tamaño, y el contenido del objeto hay un delimitador. En esencia, el hash se genera a partir de lo siguiente:

```
blob 11\0Hello
```

El delimitador es \0, antes del delimitador tenemos el tamaño que para el ejemplo es 11 bytes, y antes el tipo de objeto seguido de un espacio. Luego del delimitador está el contenido del archivo.

#### 4.2.7 Tree

Este tipo de objeto es el que representa las direcciones y los directorios. Un objeto Tree puede tener tanto blobs como otros trees. La estructura es la misma de los demás objetos (tipo, tamaño, delimitador y contenido). En este caso el contenido será diferencial al de un blob:

```
100644 blob 57537e1d8fba7d80c5bcca8b04e49666b1c1790f .babelrc
100644 blob 602c57ffb51af99d6f3b54c0ee9587bb110fb990 .flow config
040000 tree 80655da8d80aaaf92ce5357e7828dc09adb00993 dist
100644 blob 06a8a51a6489fc2bc982c534c9518f289089f375 package.json
040000 tree fc01489d8afd08431c7245b4216ea9d01856c3b9 src
```

Como podemos ver un tree puede contener blobs y más trees, tenemos tres secciones importantes: el primer número representa los permisos, el segundo es el tipo de objeto, luego va el hash, y por último el nombre o directorio.

#### 4.2.8 Permisos de objetos de git

El primer número representa los permisos de los objetos de Git, estos permisos se pueden ver en la siguiente lista.

```
040000 Directorio
100644 Archivo regular no ejecutable
100664 Archivo de escritura de grupo no ejecutable normal
100755 Archivo ejecutable regular
120000 Link simbólico
160000 Gitlink
```

La razón de que existan estos permisos es porque los repositorios de git deben ser independientes de cualquier sistema de archivos del SO en el que está.

#### 4.2.9 Creación de objetos Tree

Teniendo un ejemplo de dos objetos blob, cada uno con su respectivo hash, podemos crear un archivo del tipo tree que nos proporcione apuntadores para cada uno de los blobs y que nos proporcione la información de los nombres de los archivos de los blobs. Si los dos archivos blobs tienen los siguientes hashes

```
284a47ff0d9b952bab8ccbae29b97b5beb700e82
814d2ecd90a29b25b12880623d82e727f9a650cb
```

Los cuales representan los archivos file1.txt y file2.txt; en este caso, el contenido del tree será el siguiente:

```
100644 blob 284a47ff0d9b952bab8ccbae29b97b5beb700e82 file1.txt
100644 blob 814d2ecd90a29b25b12880623d82e727f9a650cb file2.txt
```

Para crear un nuevo tree usamos el comando `git mktree`, primero creamos un objeto del tipo texto con el contenido de arriba y lo guardamos en cualquier carpeta.

### 4.2.10 Tres pilares importantes

Dentro de los repositorios tenemos y podemos identificar tres áreas fundamentales: directorio de trabajo (working directory), staging area o index, y git repository.

El staging area que también es llamado index es el área responsable de preparar los archivos para ser insertados en un repositorio limpio, y del mismo modo, prepara los archivos tomados del repositorio para ser puestos en el directorio del trabajo. El proceso en el que los archivos pasan por el área de staging es siempre obligatorio. Es el puente entre el directorio de trabajo y el repositorio de git. Si un objeto tree está representando el nombre de dos objetos blob, significa que está representando un directorio raíz. Es decir que se hace necesario describir otro tree que represente una carpeta con su respectivo nombre en la que estén alojados los dos archivos blob.

`git ls-files -s` es un comando que sirve para listar los archivos que se encuentran en el staging area. Si queremos enviar cualquier objeto tree desde el repositorio de git hasta el área de staging, usamos el comando `git read-tree <hash>`.

### 4.2.11 Git checkout index

Teniendo los dos objetos blob creados a mano y el objeto tree también creado a mano con los nombres de los anteriores blobs, y también teniendo estos dentro del staging area, se pueden añadir dentro del directorio de trabajo, que corresponde a la carpeta física (dentro del sistema de archivos de cada SO) en la que se ven los archivos. Esto se hace con el siguiente comando: `git checkout -index -a`. Con la opción `-a` decimos que agregue todos los archivos.

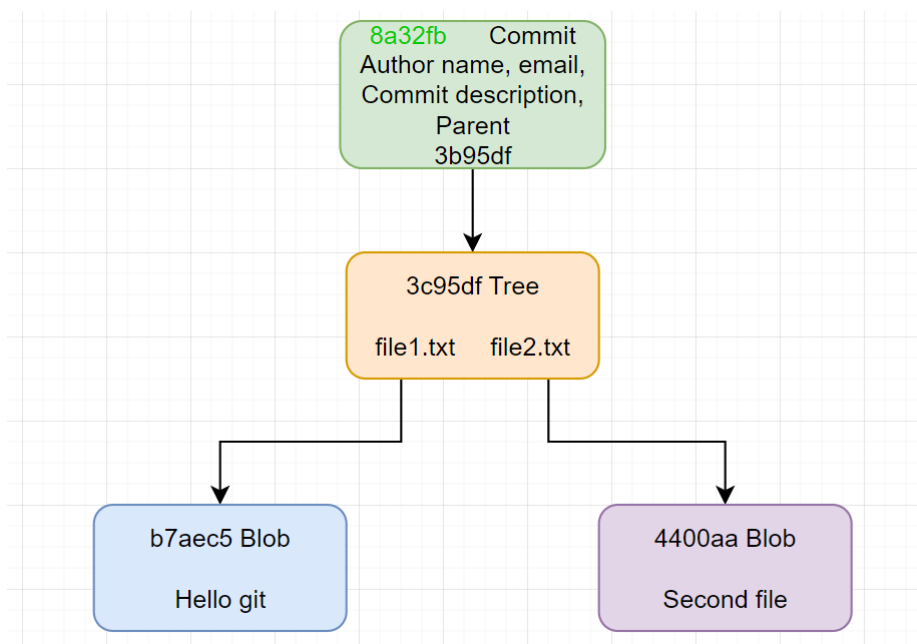
## 4.3 Operaciones básicas de Git

### 4.3.1 ¿Qué es commit?

Uno de los cuatro tipos de objetos principales es el denominado "commit". Lo primero de todo es que `commit` tiene la misma estructura que los otros tipos de objetos; es decir, que contiene la estructura de los objetos en git: un hash del tipo sha1 que consiste en tipo de objeto + tamaño + delimitador + contenido.

La diferencia está en que el contenido de un commit es el siguiente: nombre de autor, correo de autor, descripción de la versión, y como opcional, la versión padre. La confirmación de versión (commit) sirve esencialmente para guardar diferentes versiones de los proyectos. **Cada commit es una versión diferente del proyecto.**

La siguiente imagen muestra cómo son los apuntadores de cada objeto de git:



Como se puede ver, el archivo de versión (commit) es una especie de envoltorio para el objeto tree, y tiene un apuntador hacia el tree. Cada uno de los commits, puede ser llevado al directorio de trabajo para ver esa versión del proyecto. Los siguientes comandos sirven para establecer en git el nombre y dirección de correo electrónico:

```
git config --global user.name <name>
git config --global user.email <Email>
```

Y para leer la configuración establecida, usamos el siguiente comando: `git config --list`

### 4.3.2 Creación de las primeras versiones

En primer lugar, debemos estar pendientes de cuál es el estado del repositorio.

```
git status
```

Con el comando anterior podemos ver los cambios realizados para ser confirmados. También podemos ver si hay algún cambio sin seguimiento para añadir y posteriormente ser enviados/confirmados.

Una vez tengamos listos los cambios realizados para enviar, usamos el comando `git commit -m "comment"`. Con la opción `-m` podemos asignar un comentario a la versión. Es muy importante tener en cuenta que cuando hacemos confirmación de versión, estamos enviando información del 'staging area' al 'git repository', y cuando hacemos el proceso contrario (desde 'staging area' a 'working directory'), el proceso se llama 'checkout'.

El commit como archivo hash contiene lo siguiente:

- tree (es el hash del tree principal al que apunta el commit)
- parent (es el hash del commit anterior)
- Usuario autor de los cambios
- Usuario que confirmó el cambio
- Comentario

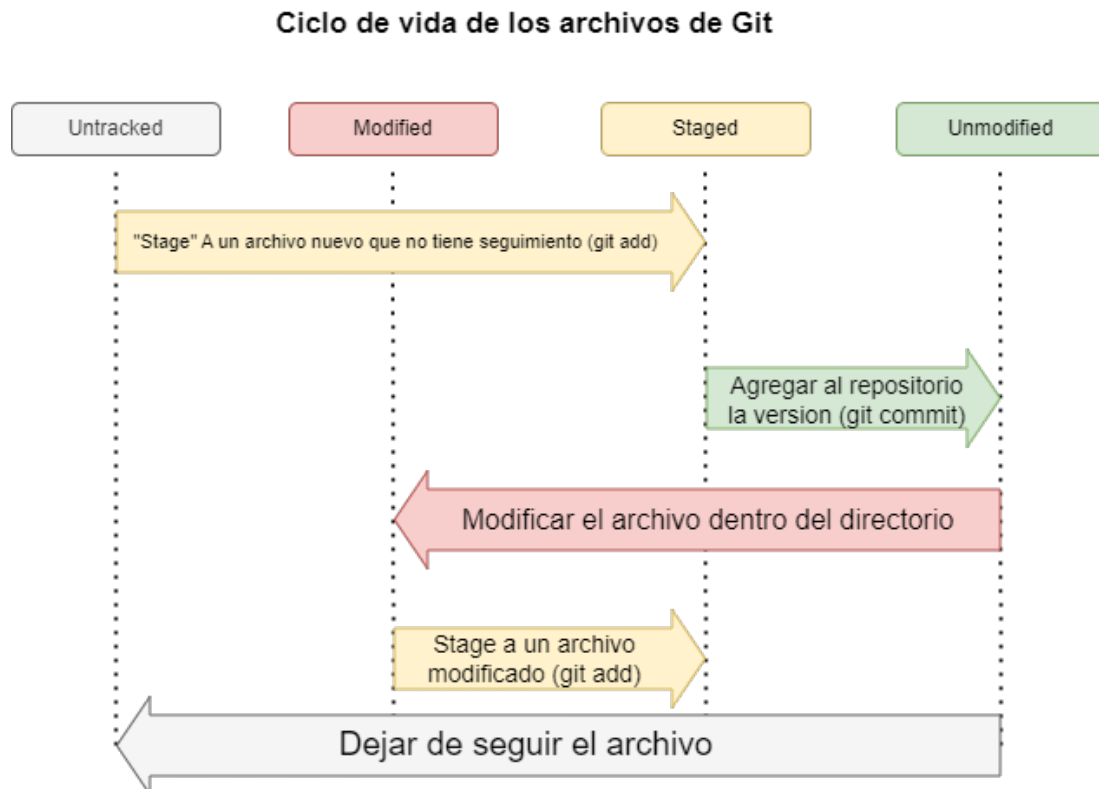
### 4.3.3 Comandos básicos de git

- `git status`
- `git add` con este se agregan archivos al area de staging
- `git commit` con este se escriben los cambios al repositorio ya como objetos
- `git log` con este se muestra el historial de cambios o commits
- `git checkout` este comando sirve para poner en el directorio actual (working directory) un commit o un branch específico.
- `git cat-file -p <hash>` Con este comando, como se mostró arriba, se visualiza el contenido del archivo correspondiente al hash ingresado.
- `git ls-files -s` Lista todos los archivos (blob) indicando el directorio donde están e indicando su hash.

Cuando se agrega un nuevo archivo al directorio, los archivos pueden tener cuatro estados diferentes:

- Untracked
- Modified
- Staged
- Unmodified

Si un archivo recién creado se adiciona al directorio, directamente está en el estado "untracked". En la siguiente figura se puede observar cómo va cambiando el estado de los archivos.



Para listar los archivos que están dentro del staging area, se usa el comando `git ls-files -s`. Al agregar archivos al área de staging, tenemos varias opciones

- `git add <name>` Agrega el archivo especificado a la zona de preparación.
- `git add -A` Agrega todos los archivos modificados, eliminados y nuevos al área de preparación. La opción `-A` incluye archivos en subdirectorios.
- `git add .` solo incluye los archivos en el directorio actual.
- `git add -u` Agrega todos los archivos modificados y eliminados al área de preparación, pero no los nuevos.
- `git add -p` Abre una sesión interactiva que permite agregar solo partes seleccionadas de los cambios realizados en un archivo.

Además podemos quitar archivos del área de preparación, mediante el comando `git rm --cached <name>`, esta última opción sirve para quitar un archivo en específico.

Una de las propiedades importantes de ver es que cuando realizamos un commit, este tiene un apuntador a su commit padre, es el hash de su commit padre. Según el tipo de commit tendrá uno u otro commit padre (más adelante se verá que para pull requests pueden haber punteros distintos).

#### 4.3.4 Historial de un archivo

Un comando útil para analizar la historia de un archivo en nuestro repositorio es el siguiente

```
git log --pretty=oneline <archivo_con_su_ruta>
```

Este comando retorna una lista de hashes que corresponden a los commits que han modificado dicho archivo:



```

13bde112178bbf94d9f83a2a14397c14d8cb973b UpdateBrowser
288cb6fee465de9e1bf682c7b47a25c4c75dd9e7 UpdateBrowser
2f7e1498f72e5d77b2773938f8516dd4c576b922 UpdateDictJson
23835e0a0d35796e708c50cfc71c523ef1b941d5 UpdateDictJson
f9dd3785bbc39a5e1ae9f8fed003f7b696f17f6b Se cambia apertura de navegador para form OVH
e1b1a7f0632ef745d02749468aa02eee016e62f9 Merge branch 'test2' of https://github.com/JhoAraSan/Proces
863641d145bdff2d4546a9b8f0328afed95d74ac Update code
47cf1a72d0ee58077a61558072c0866c46295547 cloudflare form bus ixed
673268edaff5d407e7de309c14e8a81829adf137 update
bdc232b7cb40c41c70eacfe7182d510145f26ce2 check bug
.
.
.

```

Como se puede ver, se muestra el hash del commit y a continuación se muestra el comentario realizado. El primer commit en la lista es el más reciente. El último de la lista será el commit que creó el archivo.

De forma alternativa, se puede ver solamente el commit creador del archivo en cuestión mediante el comando

```
git log --pretty=oneline --diff-filter=A -- .\Consola_3000\Consola3000.py
```

El cual devuelve

```
04bf774da85b9db1a059da4111887240ad2b3d4e renombramiento de carpeta a sugerencia de Sebastian
```

## 4.4 Las ramas

Como introducción a la sección, recordemos la capacidad de llevar un commit (screenshot de una versión del proyecto) al directorio actual. Esto se realiza mediante el comando "git checkout". En otras palabras, es algo así como saltar hacia una versión específica del proyecto.

Una definición general de lo que es una rama de github, es que es una referencia textual a un commit. Las siguientes son características de las ramas en git:

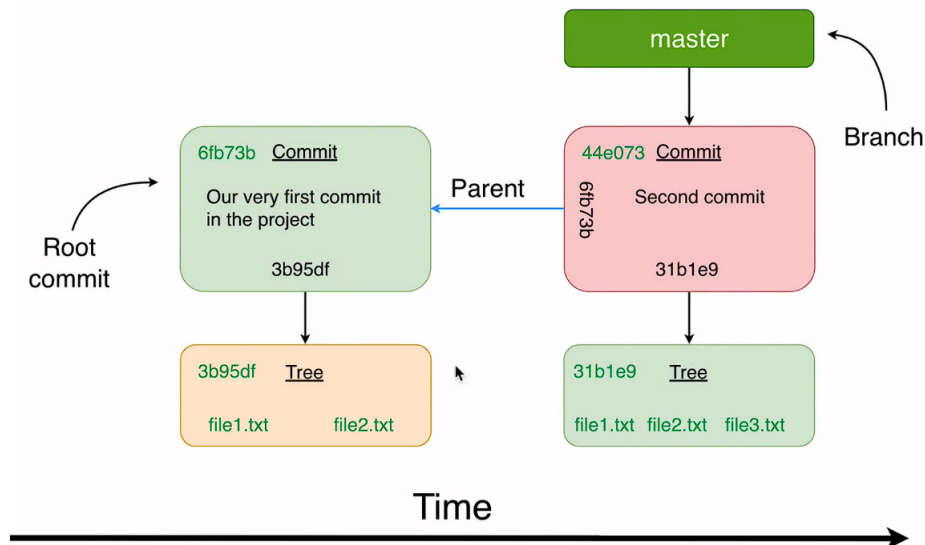
1. La rama por defecto es la master.
2. En un mismo repositorio pueden existir varias ramas.
3. Los apuntadores a todas las ramas se localizan en `.git/refs/heads`
4. Cada rama maneja sus propios commits.
5. El puntero de la rama se mueve automáticamente después de cada nuevo commit.
6. Para cambiar de branch se usa el comando `git checkout <branch>`

El puntero de la rama será el último commit realizado en dicha rama, el archivo es un texto que contiene el hash de dicho commit.

### 4.4.1 HEAD

El concepto de head es útil para especificar al sistema cuál es la rama en la que me encuentro actualmente. Básicamente HEAD es el apuntador que apunta hacia la rama/commit **actual**. Solamente existe un solo HEAD en cada proyecto.

1. El puntero se guarda en `.git/HEAD`
2. El puntero por defecto es `refs:/heads/master`
3. Para cambiar la referencia a una rama específica se usa `git checkout <branch>`
4. Para cambiar la referencia a un commit específico se usa `git checkout <sha1>`



Cada vez que se crea una nueva rama en un repositorio de Git, se crea una referencia a la cabeza (HEAD) de esa rama en el sistema de archivos de Git. Esta referencia se guarda en el directorio `.git/refs/heads/` dentro del repositorio.

Para la administración de las ramas disponemos de los siguientes comandos:

- `git branch` Lista todas las ramas locales
- `git branch <name>` Crea una nueva rama
- `git checkout <branch>` Se dirige a la rama especificada
- `git branch -d <name>` Borrar la rama especificada
- `git branch -m <old> <new>` Renombrar la rama especificada

Un comando muy útil para crear una rama nueva y dirigirse directamente a ella es la siguiente:

```
git checkout -b <branch name>
```

## Ejemplo

Para un repositorio de un proyecto cualquiera como ejemplo vamos a la carpeta `.git/refs/heads`.

Si se lista el contenido del directorio se obtiene la siguiente salida

```
Directory: C:\Users\seb-c\OneDrive\Documentos\Project_Process\Process\.git\refs\heads
```

Mode	LastWriteTime	Length	Name
1a---	3/25/2023 3:49 PM	41	master
1a---	9/6/2023 9:31 AM	41	speechGen
1a---	9/10/2023 4:00 PM	41	test2

El cual muestra que en el repositorio de ejemplo hay tres ramas: master, speechGen y test2. Si queremos ver el contenido del archivo `speechGen` se obtiene

```
866c48dd5d96c7ba7ae730dbd3dc85896bc6b576
```

Este es el hash correspondiente al **último** commit de esta rama. De aquí se puede concluir que la rama puede verse como un apuntador hacia el commit.

Para ver dónde se guarda el apuntador general HEAD hacia la rama (o commit) en el que se está actualmente. Vamos al directorio que guarda el puntero:

```
cd .git
cat HEAD
```

Se obtiene lo siguiente

```
ref: refs/heads/test2
```

El cual indica que en el momento de realizar el comando, el usuario estaba en la rama `test2`

## 4.5 Repositorios remotos

En esta sección se describirán algunas características no antes vistas sobre los procesos asociados a los repositorios remotos.

### 4.5.1 git diff

Este es un comando que puede ser útil para ver y evidenciar las diferencias entre un archivo modificado su anterior versión dentro de la consola. Al usar el comando podemos ver el hash provisional del nuevo archivo (el que tendría si se realiza el commit), las líneas agregadas- quitadas-modificadas.

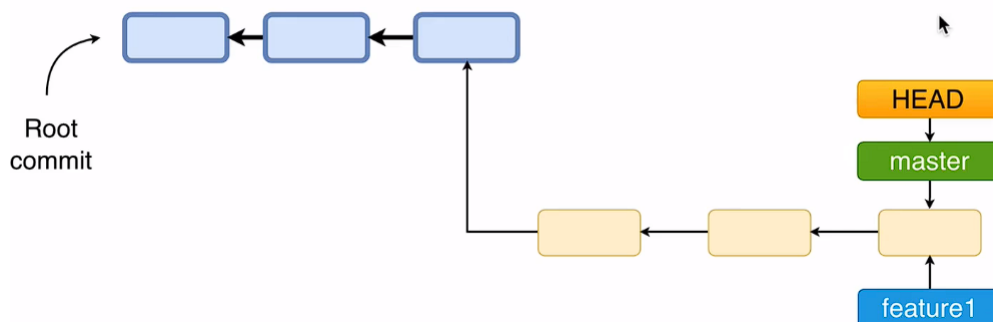
## 4.6 Fusión o combinación de ramas

Es importante tener clara la perspectiva de dónde está apuntando `HEAD`. De ello depende la información que vamos a obtener al llamar al comando `git log`. Si estamos visualizando commits anteriores, ese commit será el actual para la vista que tengamos en el momento.

Ahora teniendo en cuenta lo anterior, suponemos que hemos creado una rama para agregar cualquier especificación; hemos creado esa rama desde la rama principal. Luego hemos realizado cambios en dicha rama y hemos confirmado dichos cambios. Posteriormente volvemos a cambiar la vista hacia la rama principal y **desde esta rama traemos o unimos los cambios realizados en la rama secundaria**; ese es el proceso de fusión o combinación de ramas.

```
git merge <feature-branch>
```

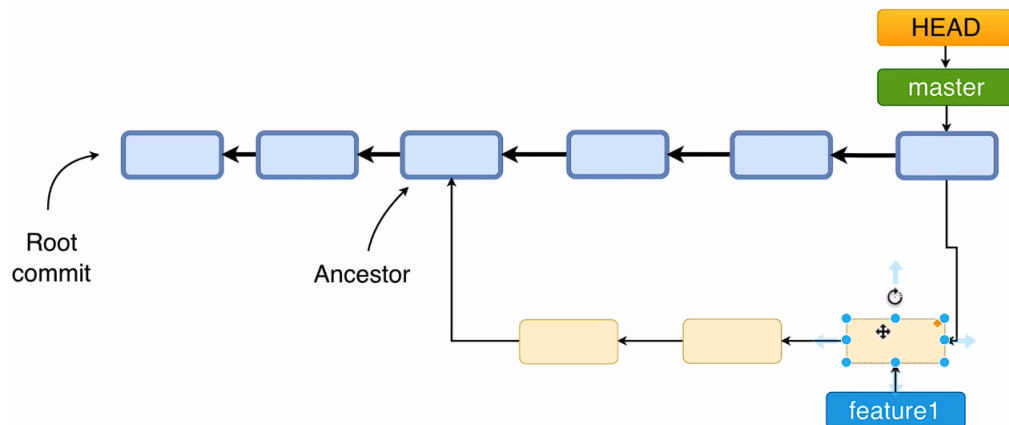
En esencia, cuando hacemos la fusión de ramas, lo que estamos realizando es un cambio del apuntador de la rama hacia la que fusionamos (la principal) para apuntar ahora al último commit realizado en la rama que estamos trayendo. Este caso aplica cuando en la rama principal no hay cambios después de haber creado la rama secundaria.



Supongamos ahora que tenemos nuestra rama principal, creamos una rama para trabajar en características secundarias, pero al mismo tiempo también realizamos cambios en la rama principal. Si en este momento queremos hacer una fusión de las ramas, ya no podemos simplemente cambiar el puntero de la rama principal;

ahora es necesario realizar una fusión de 3 direcciones.

En la fusión de tres direcciones tenemos tres versiones importantes: la versión ancestro, que corresponde a la última versión en común que tienen las dos ramas a unir; la última versión de la rama secundaria y la última versión de la principal. Como en la fusión anterior, también se debe ir a la versión de la rama receptora; se crea una nueva versión de fusión en esta rama; **dicha versión tendrá como versiones padres la última de la rama receptora y la última de la rama secundaria**. Si no existen conflictos entre archivos que se hayan modificado en ambas ramas, simplemente la nueva versión combinará los archivos nuevos.



Una vez realizado este proceso, se puede borrar la rama secundaria, lo cual significa borrar el apuntador de la rama, las versiones permanecen.

#### 4.6.1 conflictos de fusión

Los conflictos ocurren cuando se intenta fusionar dos ramas y en ambas se ha modificado el mismo archivo. Estos conflictos siempre deben ser arreglados manualmente. Si intentamos unir dos ramas y se generan conflictos, el estado actual del repositorio cambiará a tener dos caminos sin fusionar. Git le pedirá al usuario que corrija los conflictos y que confirme dichos cambios. Están las opciones de dejar los cambios de la rama principal, dejar los cambios de la rama secundaria, o de dejar ambos cambios.

Algo interesante de observar, es que en este momento se habrán creado tres objetos blob diferentes en el staging area. tres objetos que tienen el nombre del archivo que contiene el conflicto. El primero corresponde a la versión ancestro de ambas ramas, el segundo corresponde a la modificación de la rama principal, y el ultimo corresponde a la modificación de la rama secundaria. Existen varias formas de resolver estos conflictos; el primero se puede hacer mediante la consola:

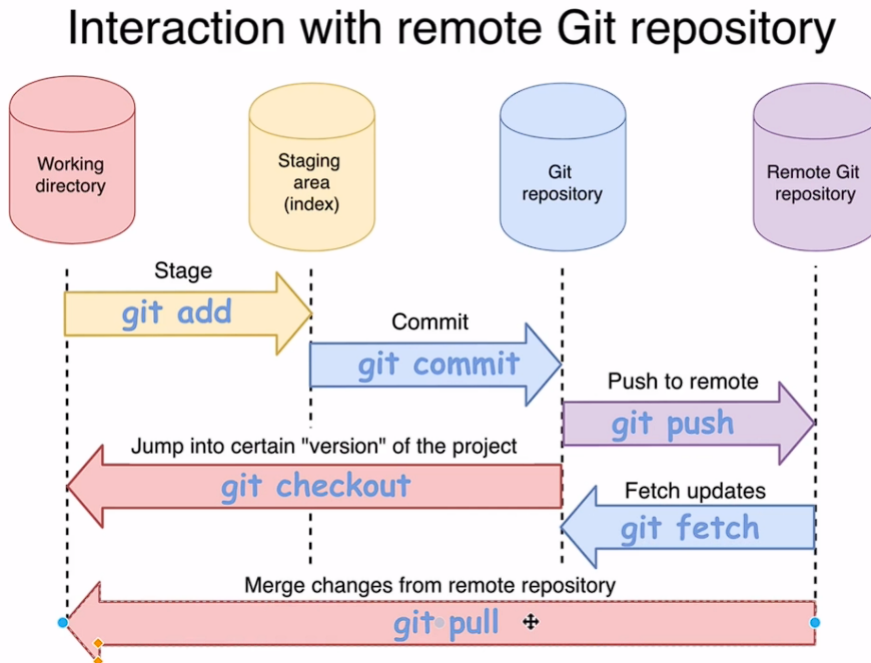
Abrimos el archivo que contiene los conflictos mediante nano por ejemplo y seleccionar manualmente cuál de la(s) líneas van a ser conservadas. guardar el archivo y de esta manera el o los conflictos habrán sido resueltos. También se tiene a opción de incluso volver a modificar el archivo si ninguna de las dos versiones es la que queremos. Finalmente cuando las modificaciones sean las deseadas, podremos concluir la fusión de las ramas mediante la confirmación (commit).

### 4.7 Comandos para los repositorios remotos

Dentro de los comandos más importantes para la clonación de los repositorios remotos, los siguientes son los más importantes:

## 4.7.1 Git push

Añadido a las tres áreas de los repositorios, tenemos un área adicional que corresponde al repositorio remoto. El primer comando envía toda la información desde el área de repositorio local y lo envía directamente al repositorio remoto. Solamente los cambios que están confirmados son los que efectivamente se ven reflejados en el repositorio remoto.



## 4.7.2 Git fetch y pull

Una vez el repositorio está actualizado, y es necesario pasar esa información al repositorio local, se pueden hacer dos comandos, el primero es `git fetch`: este comando es para enviar toda la información del repositorio remoto al repositorio local, pero sin enviarlo al área del directorio de trabajo. Por su parte, si queremos enviar directamente la información del repositorio remoto al directorio de trabajo usamos el comando `git pull`.

Para entender un poco la diferencia entre los dos comandos, supóngase que se crea una nueva rama en el repositorio remoto. Después de realizar `git fetch` dicha rama será creada en el repositorio local. En otras palabras, con `git fetch` básicamente estoy actualizando la información del repositorio remoto en mi repositorio local.

Por su parte, el siguiente ejemplo muestra el funcionamiento de `pull`:

1. Se clona el repositorio remoto
2. Se hace `checkout` a la rama master en el repositorio local
3. Se realizan cambios y se confirman en la rama master del repositorio remoto
4. Después de realizar `git pull` el repositorio local extraerá los cambios del remoto.
5. Git realiza la fusión de la rama master en el repositorio local
6. Tanto el área de staging como el directorio de trabajo se actualizan automáticamente luego de la fusión.

Cuando uno clona un repositorio remoto en uno local, git automáticamente crea un enlace entre ambos repositorios. Por defecto para el repositorio local, el nombre del repositorio remoto es `origin`.

Siempre que se clona un repositorio, git solamente crea una rama local con el mismo nombre de la rama por defecto del repositorio remoto. Por tanto, si tenemos dos ramas en el repositorio remoto, y clonamos este repositorio, en nuestro repositorio local solamente habrá una rama que es la rama principal del repositorio remoto.

Para poder traer una rama remota que no es la principal a nuestro repositorio local, solamente necesitamos hacer un checkout a dicha rama, de esta forma esta rama aparecerá en nuestro repo local.

Para los repositorios remotos, un comando importante y útil puede ser el siguiente `git remote show origin`. Con este comando podemos ver mucha más información sobre el repositorio remoto como las ramas remotas, las ramas locales, cuáles están configuradas o trackeadas, etc. El siguiente es un ejemplo de la información total de un repositorio remoto:

```
PS C:\Users\seb-c\OneDrive\Documentos\Project_Process\Process> git remote show origin
* remote origin
Fetch URL: https://github.com/JhoAraSan/Process.git
Push URL: https://github.com/JhoAraSan/Process.git
HEAD branch: master
Remote branches:
  master                tracked
  refs/remotes/origin/clases  stale (use 'git remote prune' to remove)
  refs/remotes/origin/test    stale (use 'git remote prune' to remove)
  refs/remotes/origin/virtual stale (use 'git remote prune' to remove)
  speechGen              tracked
  test2                  tracked
Local branches configured for 'git pull':
  master    merges with remote master
  speechGen merges with remote speechGen
  test2     merges with remote test2
Local refs configured for 'git push':
  master    pushes to master    (up to date)
  speechGen pushes to speechGen (up to date)
  test2     pushes to test2     (up to date)
```

En este ejemplo "stale" indica que la rama fue borrada del repositorio remoto, y se puede quitar de la lista con el comando `git remote prune origin`

Por su parte, cuando queremos sincronizar los cambios entre el repositorio remoto y el local, las ramas siempre harán un merge, es decir que se fusionarán y habrá que manejar los posibles conflictos que hayan entre la rama local y su correspondiente remota.

Para el siguiente ejemplo podemos listar todas las ramas en los repositorios local y remoto:

```
git branch -a

master
speechGen
* test2
remotes/origin/HEAD -> origin/master
remotes/origin/clases
remotes/origin/master
remotes/origin/speechGen
remotes/origin/test
remotes/origin/test2
remotes/origin/virtual
```

Como se puede ver, la línea `remotes/origin/HEAD -> origin/master` indica que en el repositorio remoto, la rama por defecto es la master.

### 4.7.3 Ramas rastreadas

Las ramas rastreadas son todas las ramas del repositorio remoto que tienen su correspondencia con la rama en el repositorio local. En otras palabras, son las ramas que aparecen tanto en el repositorio remoto como en el repositorio local.

Cuando se clona un repositorio remoto, solamente se crea la rama principal en el repositorio local. Estando en repositorio local se realiza el comando `git checkout` a una rama del repositorio remoto, y en ese momento se crea la rama rastreada. El comando `git branch -vv` muestra información de las ramas rastreadas:

```
git branch -vv
```

```
master      5d42019 [origin/master] Unificando y dejando la verdadera principal
speechGen 866c48d [origin/speechGen] first ver apps module in window DONE!
* test2     3c3e9ae [origin/test2] Cambio del nombre de la Appstore
```

Como se ve, se muestra el nombre, el hash y el comentario de las ramas que están rastreadas.

#### 4.7.4 Proceso pull

Para realizar un `git pull`, se combinan varios conceptos previamente discutidos:

1. Rama de reastreo local: Es necesario tener una rama local que esté rastreando una rama remota para poder realizar un `pull`
2. Funcionamiento del `merge`: Es necesario entender cómo funcionan las fusiones. Recordar que puede hacerse con el enfoque de avance rápido o con una fusión de tres vías.
3. `git fetch` antes del `pull`, `git` efectúa un `fetch` para obtener todos los cambios desde el repositorio remoto

El proceso de `git pull` se realiza en dos pasos:

Primero se ejecuta un `git fetch`, que toma todas las actualizaciones del repositorio remoto y las escribe en el repositorio local. Esto incluye todas las nuevas ramas y versiones creadas en el repositorio remoto.

Luego se ejecuta un `git merge`. Esta fusión es local, es decir, se hace en el repositorio local sin interactuar con el repositorio remoto. Durante este proceso, se utilizan dos ramas: la rama receptora será la rama local y la rama "feature" será la rama remota correspondiente. `Git` mezclará la rama remota en la rama local.

Es relevante notar que `Git` usa un término especial, "Fetch Head", en lugar del nombre de la rama remota durante este proceso. Como limitación, `git pull` actualiza solo la rama local que está actualmente en uso. No afecta ninguna otra rama local.

#### 4.7.5 Fetch head

Pongamos un ejemplo: sea un repositorio tal que al revisar las ramas tanto locales como remotas obtenemos el siguiente resultado:

```
git branch -a
```

```
* feature-1
master
remotes/origin/HEAD -> origin/master
remotes/origin/master
```

Vemos que tenemos las dos ramas remotas `master` y `feature-1` con sus correspondientes ramas locales. Si hacemos el siguiente código

```
git branch -vv
* feature-1   ccc9d7b [origin/feature-1]
master       f38cf54 [origin/master]
```

Vemos nuevamente la correspondencia entre las ramas. Escribiendo el comando `fetch` y luego `pull` tenemos lo siguiente:

```
git fetch -v
```

```
From https://github.com/yo/myrepo
= [up to date] feature-1 -> origin/feature-1
= [up to date] master   -> origin/master
```

```
git pull
Already up to date
```

```
git pull -v
```

```
From https://github.com/yo/myrepo
= [up to date] feature-1 -> origin/feature-1
= [up to date] master    -> origin/master
Already up to date
```

Con lo anterior podemos ver que pull siempre hace fetch antes de hacer algunos cambios adicionales para intentar fusionar las ramas.

Si vemos los archivos que están dentro de la carpeta `.git` podremos ver que hay un archivo llamado `FETCH_HEAD`, cuyo contenido sería el siguiente

```
cat FETCH_HEAD
73acf27141929dd1f890236317cb54009914c35a      branch 'test2' of https://github.com/JhoAr
629fab88a1c5954a9389c827f0e7b4829ce734ba      not-for-merge tag '1' of https://github.com/JhoAr
```

Este contenido corresponde a las ramas que están en el repositorio remoto. Si cambiamos de rama, al hacer este mismo comando, veremos en primer lugar la rama que está actualmente. Cuando hacemos `git pull`, Git primero ejecuta `"git fetch"`. Después del fetch se actualiza la lista de `.git/FETCH_HEAD` y la primera rama de esta lista será la rama actual. Finalmente Git ejecuta `git merge FETCH_HEAD` que busca la primera rama en el fetch head sin la etiqueta `"not-for-merge"` y la fusiona con la rama local actual.

#### 4.7.6 Git pull con modificación de repositorio remoto

Si después de hacer `git pull` el repositorio remoto es modificado, se crea un nuevo commit en el repositorio remoto. Git actualiza el repositorio y realiza la actualización del apuntador en el repositorio local, del commit antiguo al nuevo commit.

Ahora supongamos que se realizan cambios tanto en el repositorio remoto como en el repositorio local, y queremos traer o hacer pull al repositorio remoto. Tengamos en cuenta que al revisar con `git log`, veremos el último commit que acabamos de hacer, y anterior a ese veremos el último commit sincronizado; el que se supone que es el último commit del repositorio remoto. Sabemos que esta es una información falsa, pues este commit está desactualizado. Mediante el comando `git fetch`, vamos a actualizar la información del cambio realizado en el repositorio remoto, aunque Git ya creó los objetos dentro del repositorio local, estos aún no se ven en el directorio de trabajo porque aún no se ha fusionado con la rama local.

Después de esto podemos fusionar el cambio remoto con el cambio local. Mediante `git merge FETCH_HEAD`

#### 4.7.7 Subida de cambios al repositorio remoto

Ahora que tenemos cambios en el repositorio local que están ausentes en el repositorio remoto, es hora de poner la operación push en acción. Recordar siempre que es necesario tener permisos de escritura. Una vez estén sincronizados los repositorios, podemos ver que los apuntadores de los repositorios local (que está en la carpeta `.git/refs/head`) y la remota (que está en la carpeta `.git/refs/remotes/origin`) tienen el mismo commit de destino.

Si necesitamos crear una rama nueva, los pasos a realizar para que también se vea reflejada en el repositorio remoto es lo siguiente:

1. Se crea la nueva rama local mediante `git checkout -b nueva`
2. Se hacen los cambios correspondientes.
3. Se confirman los cambios con `commit`
4. Se sube la nueva rama al repositorio remoto con el comando `git push -v -u origin feature-2`



## 4.7.8 De rama local a rama nueva remota

Cuando se crea una rama nueva en el repositorio local y se requiere tener dicha rama en el repositorio remoto es necesario crear la rama nueva en ambos lados de forma manual. Si intentamos ejecutar el comando `push` a una rama nueva que no está en el repositorio remoto, obtendríamos el siguiente error:

```
fatal: The current branch nombre has no upstream branch.}
To push the current branch and set the remote as upstream, use
  git push --set-upstream origin nombre
```

el comando sugerido `git push --set-upstream origin nombre` efectivamente realiza la creación de la rama. Sin embargo, un comando equivalente y más corto es el siguiente

```
git push -v -u origin nombre
```

Y así queda creada la rama en el repositorio remoto con el mismo nombre que la rama local nueva.

## 4.7.9 Actualización de estados de ramas

Cuando una rama del repositorio remoto se elimina, es necesario realizar una actualización en el repositorio local. Si se ejecuta el comando `git branch -vv` se observará que aunque en el repositorio remoto ya no exista, la rama local todavía sigue a la rama remota. Incluso después de actualizar el repositorio mediante *git fetch*, vemos que la rama sigue siguiendo a la desaparecida rama remota. Para este caso se usa el siguiente comando

```
git remote update origin --prune
```

Esto asegura que el repositorio local reconozca la eliminación de la rama remota. Finalmente se ejecuta `git branch -D nombre`. Para eliminar también la rama local.

Ahora, también es posible eliminar la rama remota desde la consola local:

Recordemos que cuando se crea una nueva rama local, se debe especificar al momento de publicar el repositorio remoto:

```
git push -u origin nombre-rama
```

El comando para borrar una rama remota es

```
git push origin -d nombre-rama
```

Y luego se borra la rama local

```
git branch -D nombre-rama
```

### 4.7.9.1 comando `git show-ref`

Este comando es útil porque indica todas las referencias a las correspondientes versiones de las ramas tanto locales como remotas:

```
git show-ref
5d42019b30082902c9dd6dc8ebfcb5f63c75095d refs/heads/master
866c48dd5d96c7ba7ae730dbd3dc85896bc6b576 refs/heads/speechGen
4b8c370d8bbe8d2769cdb69e91d90fc9a2a7338e refs/heads/test2
5d42019b30082902c9dd6dc8ebfcb5f63c75095d refs/remotes/origin/HEAD
5d42019b30082902c9dd6dc8ebfcb5f63c75095d refs/remotes/origin/master
866c48dd5d96c7ba7ae730dbd3dc85896bc6b576 refs/remotes/origin/speechGen
4b8c370d8bbe8d2769cdb69e91d90fc9a2a7338e refs/remotes/origin/test2
d5c7ebc8c4cdf48f3395a092f5616a33f0aab274 refs/stash
```

Es un indicador muy útil para saber si un repositorio está debidamente actualizado. El comando puede especificar una sola rama solo añadiendo el nombre de la misma al comando.

## 4.8 Pull Requests

Una manera de definir las peticiones o solicitudes de integración "Pull requests" sería la siguiente: Una solicitud de integración es una propuesta de potenciales cambios en el repositorio. La idea principal detrás de trabajar con Git es desarrollar múltiples características de manera simultánea, usualmente en diferentes ramas y por diferentes personas. Cuando un colaborador, como Bob o Mike, está listo para aplicar cambios a la rama principal, inician comunicación con otros desarrolladores a través de "pull requests". Un "pull request" es simplemente una propuesta de cambios potenciales en el código. Estos cambios son "potenciales" porque después de una revisión por parte de otros desarrolladores, pueden ser rechazados o aprobados. Si se rechazan, el "pull request" se cierra y la rama correspondiente se elimina. El objetivo principal es aplicar los cambios para avanzar en el desarrollo.

El término "Pull Request" o "Merge Request" depende del contexto y del flujo de trabajo en desarrollo de software. En un entorno donde todos los desarrolladores trabajan en el mismo repositorio y tienen acceso de escritura, "Merge Request" sería más apropiado porque el objetivo es fusionar cambios en la rama principal tras la aprobación de revisores.

En cambio, en proyectos de código abierto con múltiples repositorios (uno principal y otros bifurcados), "Pull Request" es más adecuado. Aquí, un desarrollador que no tiene acceso de escritura al repositorio principal puede solicitar que el propietario del repositorio principal "jale" y revise los cambios de una rama en un repositorio bifurcado.

Por lo tanto, si los desarrolladores trabajan en el mismo proyecto, "Merge Request" es más apropiado. Si los desarrolladores crean bifurcaciones del repositorio y el propietario debe jalar cambios de esas bifurcaciones, entonces "Pull Request" es más adecuado.

### 4.8.1 Paso a paso del proceso de solicitudes de integración

El siguiente ejemplo expone de manera clara el proceso.

En un flujo de trabajo de desarrollo, dos desarrolladores, Mike y Bob, colaboran en un proyecto. Mike crea una nueva rama llamada "feature-one" en su computadora local y realiza cambios. Una vez satisfecho, sube estos cambios al repositorio remoto. A continuación, abre un "Pull Request" para iniciar el proceso de revisión por parte de otros colaboradores. Él puede iniciar el proceso de solicitud de integración una vez su rama esté en el repositorio remoto.

Mike solicita a Bob revisar esta solicitud. Él puede añadir dentro de la petición algunas descripciones de los cambios que realizó en su rama. Bob puede optar por descargar la rama para probar los cambios localmente o revisarlos directamente en línea. Tras la revisión, Bob puede añadir comentarios o solicitar cambios adicionales. Mike puede hacer los cambios necesarios y actualizar el Pull Request existente sin necesidad de crear uno nuevo. El Pull Request se actualiza automáticamente cuando él confirma los cambios en su rama y los sube al repositorio remoto. Cuando estas actualizaciones son subidas, a Bob se le notifica mediante correo electrónico sobre las nuevas versiones.

Una vez que Bob aprueba los cambios y se alcanza el número requerido de aprobaciones, se permite la fusión del Pull Request en la rama principal. Dependiendo de los permisos, uno de los desarrolladores o un administrador realiza la fusión.

Este proceso permite la colaboración efectiva y revisiones detalladas antes de que los cambios se integren en las ramas principales del proyecto. En resumen, el Pull Request es una herramienta central para revisar e implementar características en un entorno de desarrollo colaborativo.

Dentro de la página del repositorio remoto está la opción para crear una nueva solicitud de integración. Es importante que para que pueda haber una fusión, no pueden haber conflictos de fusión, por tanto cualquier conflicto de fusión debe resolverse antes de iniciara la solicitud de integración.

En la creación de la solicitud de integración se deber realizar una descripción del cambio, actualización, componente, característica, etc, realizado. Siempre es recomendable realizar la mejor descripción posible, ser muy claro y conciso con la descripción.

Una vez creada la solicitud, las personas pueden revisarla, añadir comentarios, ver los cambios que se han realizado, etc. Por ejemplo, dentro de la pestaña de archivos modificados, uno puede insertar un comentario sobre una línea de código específica.

Existen dos opciones para publicar el comentario: uno es añadir un solo comentario a la línea de código, y la otra es iniciar una review, que implica un grupo de líneas o una sección de código. Por otro lado está la opción de aprobar la solicitud dejando un comentario de la aprobación, dejar un simple comentario sin indicar aprobación o declinación, y sugerir más cambios para una nueva revisión.

Una vez la solicitud está aprobada, cualquier usuario con los permisos correspondientes puede fusionar la rama. Es importante notar que en algunas ocasiones Github por defecto permite a usuarios fusionar la rama incluso si no hay ninguna review.

# Chapter 5

## Datos

### 5.1 Modelo de datos

¿Qué es un modelo de datos?

Es un conjunto de tablas de datos que tienen relaciones y conexiones entre sí. Lo importante es que existan relaciones entre las diferentes tablas; relaciones de conexión entre datos como columnas o datos claves en común.

#### 5.1.1 Normalización de los datos

Es el proceso de organización de las tablas y las columnas en una base de datos relacional para reducir la redundancia y preservar la integridad de los datos.

- la eliminación de los datos para reducir el tamaño de las tablas y aumentar la velocidad y eficiencia.
- Minimizar las anomalías y los errores de las modificaciones de los datos.
- simplificación de las consultas y estructuración de la base de datos para análisis más útiles.

una buena manera de entender una base de datos normalizada es definiéndola de la siguiente manera: en una base de datos normalizada, cada tabla debe tener un **propósito único y específico**.

los modelos tienen generalmente dos tipos de tablas

1. Tablas de datos: contiene números o valores, típicamente a nivel granular, con una columna de identificación o Id que puede ser
2. Tablas de Vista: provee atributos descriptivos normalmente basados en texto sobre cada dimensión de la tabla. en este caso, este tipo de tablas provee mucha información sobre objetos como pueden ser "clientes" o "Productos".

#### 5.1.2 Expresiones de análisis de datos

Las expresiones de análisis de datos o DAX son un lenguaje de fórmulas de PowerBI. Con estas fórmulas se puede hacer lo siguiente:

- Añadir columnas con cálculos y mediciones al modelo, usando sintaxis intuitiva.
- Ir más allá de las capacidades de las formulas tradicionales, con funciones potentes y flexibles construidas específicamente para trabajar con modelos de datos relacionales.

Existen dos maneras de realizar acciones DAX: mediante creación de columnas nuevas, y mediante mediciones de un solo valor.

### 5.1.2.1 Columnas calculadas

Se pueden agregar columnas nuevas basadas en fórmulas a las tablas. Cosas importantes a remarcar:

1. No hay referencias a celdas, todas las columnas calculadas se referencian a tablas enteras o a columnas enteras
2. Las columnas calculadas generan valores por cada fila, los cuales son visibles dentro de las tablas en la vista de datos.
3. Las columnas calculadas entienden el conexto de filas, lo que significa que son muy útiles para definir propiedades que estén basadas en la información de cada fila.

Como un tip adicional y útil, use las columnas calculadas para **filtrar datos** más que para crear valores numéricos.

#### Ejemplo

```
Parent = IF(Customer_Lookup[TotalChildren]>0, "Yes", "No")
```

En este ejemplo se puede usar un if condicionante para agregar una columna nueva de control o de filtro.

### 5.1.2.2 Mediciones

En este caso son usados para generar nuevos valores calculados.

1. Igual que la anterior, no hay forma de referenciar celdas aisladas; solo para tablas enteras.
2. Estos valores no pueden ser visibles dentro de las tablas, solamente se pueden ver en elementos de visualización como cartas, o matrices.
3. Estos cálculos siempre estarán regidos por el contexto del filtro; es decir, el valor se recalcula siempre que sea aplicado algún filtro en el elemento de visualización que se agregue o configure.

A manera de regla general, use este tipo de cálculo cuando una única fila no puede entregarle el valor solicitado. En otras palabras, cuando necesite más de una fila.

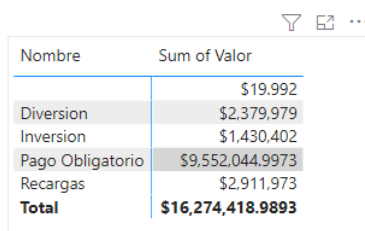
### 5.1.2.3 Mediciones implícitas y explícitas

Las mediciones implícitas están en los elementos de visualización cuando uno arrastra un campo numérico a dicho elemento. Ahí es cuando se puede seleccionar el modo de agregación (suma, media, mínimo, máximo, etc).

Las mediciones explícitas, son por el contrario las que se definen de manera literal con la sintaxis de DAX

Recordando que las mediciones son evaluadas con base en el contexto de filtrado, según el filtro aplicado el resultado de la medición sera distinta.

Por ejemplo, para la siguiente imagen, el valor sombreado es la suma del campo "Valor" está canculado con base en el siguiente contexto de filtro: IDLookup[Nombre]="Pago Obligatorio"



Nombre		Sum of Valor
		\$19,992
Diversion		\$2,379,979
Inversion		\$1,430,402
Pago Obligatorio		\$9,552,044.9973
Recargas		\$2,911,973
<b>Total</b>		<b>\$16,274,418.9893</b>

CALCULATE(Measure, DATEADD(Calendar[Date], -1, MONTH)) Aquí la función DATEADD devuelve un rango de fechas desplazado en la cantidad y con intervalos puestos en sus argumentos  
CALCULATE(Measure, DATESINPERIOD(Calendar[Date], MAX(Calendar[Date]), -1, DAY))

#### 5.1.2.4 Sintaxis de las expresiones DAX

Las sentencias de las expresiones se pueden separar en tres partes diferentes: nombre de la medición realizada, nombre de la función, y las referencias o argumentos de las funciones, estas pueden referirse al nombre de la tabla con su respectiva columna. También se puede estar refiriendo a una medición anteriormente definida. Como tip de utilidad, cuando nos vayamos a referir a referencias de columnas, usamos el nombre: Table[Column]. Y para referencias a mediciones, solo usamos el nombre de la medición: Measure.

Algunos operadores importantes los podemos ver en la siguiente tabla

Operador	Significado	Ejemplo
+	Suma	2 + 7
-	Resta	5 - 3
*	Multiplicación	2 * 6
/	División	2 / 4
^	Exponente	5 ^ 5
=	Igual a	[City] = "Boston"
>	Mayor	[Quantity] > 10
<	Menor que	[Quantity] < 10
>=	Mayor o igual	[Unit_Price] >= 2.5
<=	Menor o igual	[Unit_Price] <= 2.5
<>	Diferente a	[Country] <> "Mexico"
&	Concatena dos caracteres o cadenas para formar una nueva	[City] & " " & [State]
& &	AND	([State]="MA") & & ([Quantity]>10)
	OR	([State]="MA")    ([State]="CT")
IN	Crea una condición lógica OR basada en una lista dada.	'Store Lookup'[State] IN "MA", "CT", "NY"

Las funciones también las podemos clasificar en diferentes categorías:

##### 1. Math and stats

Tenemos aquí las funciones matemáticas de agregación más básicas y también algunas iteradoras:

- SUM
- AVERAGE
- MAX/MIN
- DIVIDE
- COUNT/COUNTA
- COUNTROWS
- DISTINCTCOUNT
- SUMX
- AVERAGEX
- MAXX/MINX
- RANKX
- COUNTX

##### 2. Logical functions

Estas funciones generalmente tienen la tarea de retornar información sobre valores, con base en una expresión condicional dada. usualmente o mayoritariamente basadas en condicionales "if":

- IF
- IFERROR
- AND
- OR
- NOT
- SWITCH

- TRUE
- FALSE

### 3. Text functions

Son funciones que sirven para manipular texto o formatos de control para fechas, horas o números

- CONCATENATE
- FORMAT
- LEFT/MID/RIGHT
- UPPER/LOWER
- PROPER
- LEN
- SEARCH/FIND
- REPLACE
- REPT
- SUBSTITUTE
- TRIM
- UNICHAR

### 4. Filter functions

Estas son funciones que están basadas en tablas relacionadas y para filtrar funciones para cálculos dinámicos.

- CALCULATE
- FILTER
- ALL
- ALLEXCEPT
- RELATED
- RELATEDTABLE
- DISTINCT
- VALUES
- EARLIER/EARLIEST
- HASONEVALUE
- HASONEFILTER
- ISFILTERED
- USERELATIONSHIP

### 5. Date and time functions

Estas son funciones especializadas en fechas y horas, así como operaciones inteligentes avanzadas

- DATEDIFF
- YEARFRAC
- YEAR/MONTH/DAY
- HOUR/MINUTE/SECOND
- TODAY/NOW
- WEEKDAY/WEEKNUM
- DATESYTD
- DATESQTD
- DATESMTD

- DATEADD
- DATESINPERIOD

Un par de funciones muy útiles para el manejo de las fechas son `weekday/weeknum()` y `eomonth()`. La primera función retorna el número del día de la semana de la fecha que se ponga como argumento. Por defecto, el número 1 será el día domingo, pero esto se puede cambiar mediante una opción. La segunda retorna la fecha correspondiente al último día del mes, más o menos un número especificado de meses.

Una de las funciones más importantes o útiles puede ser `RELATED` la cual retorna valores relacionados directamente con cada fila con base en la relación que haya con otras tablas. La sintaxis de la función es:

La función `DATEADD` Devuelve la fecha que resulta de restar o sumar la cantidad especificada en el argumento.

`=RELATED(ColumnName)`

el nombre de la columna es la columna de la cual se requiere extraer la información o el dato. Como tip es recomendable tratar de no crear nuevas columnas basadas en esta función, en algunas osiones esto puede ser redundante; es más recomendable usar este tipo de funciones dentro de otras como iteradores del tipo `SUMX`.

`DATEADD(Calendar[date], -1, MONTH)`

La anterior devuelve la fecha restada en un mes

Teniendo en cuenta la tabla de gastos personal, si usamos la siguiente función:

`YTD Gastos = CALCULATE(SUM(Gasto[Valor]), DATESYTD(Calendario_Lookup[Fecha]))`

obtenemos la suma acumulativa de los gastos y se "renueva" en año nuevo. Observar como las barras van aumentando y el valor acumulado en el año.

Start of Month	Sum of Valor	YTD Gastos
Friday, July 01, 2022	\$1,385,890	\$1,385,890
Monday, August 01, 2022	\$2,677,236.256	\$4,063,126.256
Thursday, September 01, 2022	\$2,809,979.404	\$6,873,105.66
Saturday, October 01, 2022	\$2,622,030.812	\$9,495,136.472
Tuesday, November 01, 2022	\$2,756,051.46	\$12,251,187.932
Thursday, December 01, 2022	\$5,171,846.0376	\$17,423,033.9696
Sunday, January 01, 2023	\$5,508,035.1006	\$5,508,035.1006
Wednesday, February 01, 2023	\$826,739.9978	\$6,334,775.0984
<b>Total</b>	<b>\$23,757,809.068</b>	<b>\$6,334,775.0984</b>

### 5.1.3 Parámetros 'what if'

Este tipo de parámetros son esencialmente parámetros DAX preconfigurados que producen valores dentro de un rango dado. Este tipo de medidas pueden ser muy útiles por ejemplo para estudios de pronósticos o para estudios de posibles escenarios. para más información revisar la lección 98 del curso.

## 5.2 SQL

### 5.2.1 Conceptos

#### 5.2.1.1 Query

Es una porción de código que induce a la computadora a ejecutar una serie de operaciones y que entregará la salida requerida.

SQL Es un lenguaje de programación declarativo no procedural. No se focaliza tanto en el procedimiento como en cuál es la tarea que se está solicitando. Su sintaxis se compone principalmente de



- Lenguaje de definición de datos
- Lenguaje de manipulación de datos
- Lenguaje de control de datos
- Lenguaje de control de transacciones

CREATE object\_type object\_name

CREATE TABLE object\_name (column\_name data\_type)

CREATE TABLE sales (purchase\_number INT data\_type) creará una tabla llamada sales con una columna llamada purchase number del tipo entero.

ALTER TABLE sales  
ADD COLUMN date\_of\_purchase DATE;

Se modifica la tabla agregando una columna nueva.

DROP TABLE customers; borra la tabla entera

RENAME TABLE customers TO customer\_data; cambia el nombre de la tabla.

TRUNCATE borra los datos enteros de una tabla, pero la tabla sigue existiendo

Ahora algunas palabras reservadas de DML (lenguaje de manipulación de datos)

SELECT .. FROM sales selecciona lo indicado de la tabla de ventas. Se usa para extraer información de la tabla

INSERT INTO sales (purchase\_number, date\_of\_purchase) VALUES(1, '2017-10-11'); añade datos a la tabla

UPDATE sales  
SET date\_of\_purchase = '2017-12-11'  
WHERE purchase\_number = 1;

Cambia directamente una entrada de la tabla

DELETE FROM sales  
WHERE  
purchase\_number = 1;

Sentencias para control de datos

GRANT type\_permission ON database\_name.table\_name TO 'username'@'localhost'

Otorga permisos determinados a los usuarios

REVOKE type\_permission ON database\_name.table\_name TO 'username'@'localhost'

Quita los permisos.

Finalmente los comandos de control de transacciones

COMMIT solamente funciona cuando se hacen cambios del tipo INSERT, DELETE, o UPDATE.

UPDATE customers  
SET last\_name = "Johnson"  
WHERE customer\_id = 4  
COMMIT;

ROLLBACK permite devolver al estado anterior de commit

## 5.2.2 Pasos para crear una database y usarla

```
create database if not exists Sales;  
use Sales
```

EL lenguaje SQL no es sensible a mayúsculas-minúsculas, ni para los nombres de los diferentes objetos, ni para las solicitudes. Esto quiere decir que `sales` y `Sales` serán iguales para SQL

## 5.2.3 Tipos de datos

Siempre es necesario especificar el tipo de datos que se insertará en cada columna de la tabla.

### 5.2.3.1 Cadenas de caracteres

Existen varios tipos de cadenas string.

1. **caracter:** `CHAR`, tiene un tamaño de almacenamiento fijo y depende de qué tamaño sea declarado. `CHAR(5)` tendrá un tamaño de 5 bytes aunque el string tenga menos de 5 símbolos.
2. **caracter variable:** `VARCHAR` en este caso el tamaño de la variable no está fijo, así un `VARCHAR(5) = 'bob'` tendrá un tamaño de 3 bytes.
3. **ENUM** Es un tipo de variable que contiene un conjunto definido total de cadenas. Por ejemplo, para el género, cuando solamente hay dos opciones para seleccionar, se puede declarar la variable `ENUM('M', 'F')`.

Una cadena de caracteres `char` puede tener un tamaño máximo de 255 bytes. Un `varchar`, por su parte, puede tener un tamaño máximo de 65535 bytes. El procesamiento de las variables `char` es más rápido y eficiente que el de las `varchar`, razón por la cual a veces es más conveniente declarar las primeras.

### 5.2.3.2 Enteros

La siguiente tabla resume los diferentes tipos de enteros que se pueden declarar y sus tamaños máximos así como los valores máximos que pueden tomar

Tipo	Tamaño	Valor mínimo (con signo/sin signo)	Valor máximo (con signo/sin signo)
TINYINT	1	-128/0	127/255
SMALLINT	2	-32 768/0	32 767/65 535
MEDIUMINT	3	-8 388 608/0	8 388 607 / 16 777 215
INT	4	-2 147 483 648 /0	2 147 483 647 / 4 294 967 295
BIGINT	8	-9 223 372 036 854 775 808 / 0	9 223 372 36 854 775 807 / 18 446 744 73 709 551 615

## 5.2.4 Variables de puntos flotantes y puntos fijos

En este contexto la precisión de un número se refiere a la cantidad de dígitos que hay en el mismo. Por ejemplo, `10.523` tiene una precisión de 5. Por su parte, la escala de la variable se refiere a la cantidad de dígitos que hay después del punto decimal. en el ejemplo anterior, la escala es de 3.

Es importante saber la diferencia entre datos de punto fijo y datos de punto flotante. Los datos de punto fijo son los que representan valores exactos. Existen dos tipos de variables para los puntos fijos:

`DECIMAL(5, 3)`  
`NUMERIC`

Para el ejemplo `decimal(5, 3)` cualquier número insertado en esta variable contendrá esta estructura. Si se inserta un `10.5` el número en realidad será `10.500`, y si se agrega un número con más decimales, entonces se redondeará para que pueda entrar en la variable, se saltará una advertencia.

Un buen ejemplo de uso de este tipo de datos son los valores monetarios: Salarios, gastos, etc. `NUMERIC(p, s)` donde `p` representa la precisión y `s` representa la escala.

Por su parte, los datos de coma flotante se usan para aproximar valores solamente y tiene como objetivo equilibrar el alcance y la precisión, Simplemente aproxima el valor y guarda esa aproximación. En este caso tenemos los siguientes tipos de variables para los números de coma flotante.

FLOAT  
DOUBLE

float tiene un tamaño de 4 bytes. Es de precisión sencilla y cuenta con un máximo de 23 dígitos.  
double tiene un tamaño de 8 bytes. Es de precisión doble y cuenta con un máximo de 53 dígitos.

### 5.2.5 Otros tipos de datos

- DATE el formato es YYYY-MM-DD. La hora no hace parte de esta representación
- DATETIME El formato es YYYY-MM-DD HH:MM:SS[.fraction]. El rango de la hora es 0 - 23:59:59.999999
- TIMESTAMP Es el conteo en segundos desde una fecha inicial, sirve como punto de comparación entre dos fechas y establecer la diferencia en tiempo
- BLOB Significa binary large object. Se puede usar para guardar archivos de diferentes tipos: objetos binarios de gran tamaño. como fotos.
- 

### 5.2.6 Creación de tablas

```
CREATE TABLE table_name
(
    column_1 data_type constraints,
    .
    .
    .
    column_n data_type constraints
);
```

Ejemplo:

```
CREATE TABLE Sales
(
    purchase_name int not null primary key auto_increment,
    date_of_purchase date not null,
    customer_ID int,
    item_code varchar(10) not null
);
```

la sentencia auto\_increment asigna el número de ID a esta columna y la aumenta automáticamente.

```
CREATE TABLE Customers
(
    customer_ID int not null primary key auto_increment,
    first_name varchar(255) not null,
    last_name varchar(255) not null,
    email_address varchar(255),
    number_of_complaints int
);
```

### 5.2.7 Restricciones de SQL

Las restricciones son especificaciones de reglas o límites que se imponen a cada una de las columnas de la tabla. Los términos "primary key" (clave primaria) y "foreign key" (clave foránea) son fundamentales para entender cómo se relacionan las tablas entre sí y cómo se mantiene la integridad de los datos.

### 5.2.7.1 primary key

Una clave primaria es un campo (o conjunto de campos) que identifica de manera única cada fila en una tabla de una base de datos. Las características principales de una clave primaria son:

1. **Unicidad:** Ningún valor duplicado es permitido en una clave primaria. Cada fila debe tener un valor único para la clave primaria.
2. **No nula:** Una clave primaria no puede tener valores nulos. Cada fila debe tener un valor para la clave primaria.
3. **Identificación:** La clave primaria se utiliza para identificar de manera exclusiva una fila en la tabla.

Una forma de asignar una clave primaria a una tabla es la siguiente:

```
CREATE TABLE Sales
(
purchase_number int auto_increment,
date_of_purchase date,
customer_ID int,
item_code varchar(10),
primary key (purchase_number)
);
```

Creamos las demás tablas con sus respectivas claves primarias:

```
create table customer
(
customer_id int,
first_name varchar(255),
last_name varchar(255),
number_of_complaints int,
primary key c(customer_id)
);
```

### 5.2.7.2 Clave foránea

Una clave foránea es un campo (o conjunto de campos) en una tabla que se utiliza para referenciar la clave primaria de otra tabla. La relación entre la clave foránea y la clave primaria es lo que permite establecer relaciones entre tablas. Las características principales de una clave foránea son:

1. **Correspondencia** con la clave primaria: Una clave foránea en una tabla corresponde a una clave primaria en otra tabla.
2. **Integridad referencial:** La clave foránea ayuda a mantener la integridad referencial entre las tablas, asegurando que la relación entre ellas sea válida. Por ejemplo, si una fila en una tabla A hace referencia a otra fila en una tabla B, entonces la fila referenciada debe existir en la tabla B.
3. **Valores nulos:** A diferencia de las claves primarias, las claves foráneas pueden tener valores nulos, lo que indica que no hay una relación con otra tabla.

En el contexto de bases de datos relacionales, los términos "tabla padre" y "tabla hija" son fundamentales para entender cómo se estructuran y relacionan los datos. Estos conceptos están intrínsecamente vinculados a las relaciones entre tablas, en particular a través del uso de claves primarias y foráneas.

**Tabla Padre:** Una tabla padre, en una relación de base de datos, es aquella que tiene una clave primaria referenciada por otra tabla. La característica clave de una tabla padre es que proporciona el registro principal o la fuente de datos que otras tablas referencian. Las propiedades de una tabla padre son :

1. **Posee una Clave Primaria:** Es esencial que tenga una clave primaria, que es un identificador único para cada fila de la tabla .

2. **Independencia:** La tabla padre puede existir en la base de datos sin la necesidad de tener una tabla hija asociada. Sus registros no dependen de la existencia de registros en otra tabla.
3. **Fuente de Referencia:** Otras tablas (tablas hijas) hacen referencia a la tabla padre a través de sus claves foráneas, estableciendo así una relación.

**Tabla Hija:** Una tabla hija es aquella que contiene una clave foránea que hace referencia a la clave primaria de otra tabla (la tabla padre). Las características de una tabla hija son

1. **Posee una Clave Foránea:** Contiene al menos una clave foránea que establece una relación con la clave primaria de la tabla padre.
2. **Dependencia Referencial:** La existencia de registros en la tabla hija generalmente depende de los registros en la tabla padre. Por ejemplo, no se puede insertar un registro en la tabla hija si no existe un registro correspondiente en la tabla padre, a menos que la clave foránea permita valores nulos.
3. **Integridad Referencial:** La clave foránea asegura que las relaciones entre la tabla hija y la tabla padre sean válidas y coherentes.

La forma de definir una clave foránea es

```
create table sales
(
  purchase_number int auto_increment,
  date_of_purchase date,
  customer_id int,
  item_code varchar(10),
  primary key(purchase_number),
  foreign key(customer_id) references customers(customer_id)
);
```

Donde la referencia se hace a la tabla que tenga esa clave como la primaria. Hay una restricción adicional llamada `on delete cascade`. Esta cláusula indica al sistema que si se elimina en la tabla padre una entrada o fila, entonces todas las entradas de la tabla hija que hagan referencia a esa clave también deberán ser borradas.

Se pueden modificar las características de las tablas sin necesidad de borrarlas para añadirlas nuevamente, mediante `alter`

```
alter table name
add foreign key;
drop primary key
```

De la misma forma se puede eliminar cualquier restricción o característica.

### 5.2.7.3 Clave única

La restricción `UNIQUE KEY` es un tipo de restricción que se puede aplicar a una o más columnas de una tabla para asegurar que cada fila de esa columna (o combinación de columnas) tenga un valor único dentro de la tabla. Es decir, no se pueden tener dos o más filas con el mismo valor en la(s) columna(s) marcada(s) como `UNIQUE`. Un ejemplo clásico de esta restricción es una columna de correos electrónicos; que no necesariamente constituye la clave principal, pero no puede ser duplicado, aunque pueden haber filas con valores en blanco en esta columna.

Para borrar una clave única de una tabla se debe usar

```
drop index name;
```

La instrucción `DROP INDEX` para eliminar una clave única `UNIQUE KEY` puede parecer inicialmente confusa, pero tiene sentido si consideramos cómo se implementan las claves únicas internamente en los sistemas de gestión de bases de datos.

**Implementación de Claves Únicas como Índices** Cuando se define una UNIQUE KEY en una tabla, la mayoría de los sistemas de bases de datos automáticamente crean un índice único para esa clave. Este índice único es lo que efectivamente garantiza la restricción de unicidad, asegurando que no se puedan insertar valores duplicados en la columna o conjunto de columnas designadas. pero ¿Por Qué Se usa DROP INDEX?

1. **Índices Únicos Subyacentes:** Dado que las claves únicas son implementadas internamente como índices únicos, la eliminación de una restricción de clave única implica la eliminación del índice asociado.
2. **Consistencia con Otros Tipos de Índices:** Los RDBMS suelen tratar las claves únicas de manera similar a otros índices (como los índices normales o los índices de texto completo). Por lo tanto, se utiliza un comando común DROP INDEX para eliminar cualquier tipo de índice, ya sea que haya sido creado para una clave primaria, una clave única, o para optimización de consultas.
3. **Simplificación de la Sintaxis SQL:** Utilizar un comando común para eliminar índices, independientemente de su tipo, simplifica la sintaxis del lenguaje SQL y hace que el manejo de la base de datos sea más uniforme y predecible.

**Consideraciones Adicionales:** Es importante mencionar que la sintaxis exacta para eliminar una clave única puede variar entre diferentes sistemas de bases de datos. Algunos sistemas pueden ofrecer una sintaxis que permite eliminar directamente una clave única sin referirse explícitamente al índice. Por su parte, eliminar una clave única (y por lo tanto, el índice único asociado) debe hacerse con cuidado, ya que altera las restricciones de integridad de la tabla y puede afectar el rendimiento de las consultas.

#### 5.2.7.4 Por defecto

La restricción default sirve para asignar un valor particular a cualquier fila de una columna; una forma de establecer un valor predeterminado para los casos en que no se especifica un valor particular al momento de añadir filas en la tabla. `number_of_complaints int default 0`. Para modificar una columna y configurar su valor por defecto, se puede mediante la siguiente forma:

```
alter table name
change column old_name new name new constraint;
alter column col_name drop constraint;
```

Las sentencias ALTER TABLE ... CHANGE COLUMN y ALTER TABLE ... ALTER COLUMN son utilizadas para modificar las características de las columnas, pero tienen propósitos y comportamientos distintos.

CHANGE COLUMN se usa principalmente para cambiar el nombre de la columna, y también permite cambiar el tipo de datos de la columna y agregar o modificar restricciones. Por su parte, ALTER COLUMN se usa para modificar restricciones o configuraciones de la columna ; específicamente para cambiar aspectos como el valor por defecto o para modificar la capacidad de aceptar valores nulos. A diferencia de change column, alter column no se usa para cambiar el nombre de la columna o su tipo de datos.

Hay una tercera, llamada MODIFY COLUMN: Usado para cambiar el tipo de datos de una columna y modificar o agregar restricciones, pero sin cambiar el nombre de la columna.

#### 5.2.7.5 NOT NULL

Esta restricción para una columna impide que una entrada nueva esté vacía; es decir, debe tener algún valor siempre.

### 5.2.8 Buenas prácticas

Es evidente la importancia de realizar siempre un código limpio y bien entendible, organizado, cuyas secciones sean fácilmente intercambiables. Al momento de declarar nombres, asegurarse de que sean nombres cortos, con significado acorde a su función, que brinde información sobre sí mismo, que sea pronunciable. La convención clásica indica que aunque SQL no distingue mayúsculas de minúsculas, es buena práctica usar mayúsculas para escribir palabras reservadas y tipos de instrucciones, y minúscula para los nombres, nunca usar espacios, y separar palabras por guión bajo.

### 5.2.9 Manipulación de datos

Una de las declaraciones más importantes en el mundo de SQL es SELECT.

### 5.2.9.1 SELECT

Es una declaración que permite extraer una fracción del conjunto de datos. Permite obtener datos de las tablas. Básicamente se trata de una instrucción que se usa para solicitar datos de la base de datos. La sintaxis es la siguiente:

```
SELECT col_1, col_2, ...  
FROM table_name;
```

Hay una cláusula dentro de la declaración denominada WHERE, la cual permite establecer una condición sobre la cual se puede especificar qué parte de los datos se requiere obtener

```
SELECT col_1, col_2, ...  
FROM table_name  
WHERE condition;
```

La condición puede ser cualquier valor de verdad: `WHERE first_name = 'Denis'`. Los operadores de verdad son

- =
- AND
- OR
- IN
- NOT IN
- LIKE
- NOT LIKE
- BETWEEN... AND..
- EXISTS
- NOT EXISTS
- IS NULL
- IS NOT NULL

Uno de interés de los anteriores es el LIKE, utilizando

```
SELECT * FROM table WHERE first_name LIKE('seb%')  
SELECT * FROM table WHERE first_name LIKE('%an')
```

Indica una búsqueda de los nombres que empiecen por 'seb' ('seb%') o que terminen con 'an' (%an). Si se usa %ak% se selecciona todas las que tengan 'ak' en cualquier parte. Si se usa por ejemplo 'Mar\_', entonces solo se buscará todo lo que contenga 'Mar' y **un solo caracter más**. Por parte de las fechas, SQL es capaz de realizar comparaciones entre fechas son operadores simples, por ejemplo la comparación

```
hire_date > '2000-01-01'
```

La sentencia `SELECT DISTINCT` tiene la capacidad de retornar solamente valores únicos.

### 5.2.9.2 Funciones agregadas

Se puede obtener mucha información de las tablas a través de las funciones de agregación

1. COUNT
2. SUM
3. MIN
4. MAX
5. AVG

La sintaxis es la siguiente

```
SELECT COUNT(col_name)
FROM table_name
```

Las operaciones de verdad se deben hacer directamente con las tablas:

```
SELECT
    COUNT(DISTINCT col_name)
FROM
    table_name;
```

Las funciones de gregación ignora los elementos NULL a menos que se le indique lo contrario. Por su parte, seleccionamos operaciones especializadas mediante la forma

```
SELECT
    COUNT(*)
FROM
    table_name
WHERE
    col_name > 100000;
```

**Ordenación** Podemos ordenar el resultado con base en cualquier columna, para esto usamos ORDER BY col\_name:

```
SELECT
    *
FROM
    table_name
ORDER BY first_name;
```

Por defecto el ordenamiento es ascendente, para configurar lo contrario usamos

```
SELECT
    *
FROM
    table_name
ORDER BY first_name DESC;
```

Se pueden poner varias columnas para ordenar, la primera tendré la prioridad.

```
SELECT
    *
FROM
    table_name
ORDER BY first_name other_col ASC;
```



**GROUP BY** Todos los resultados de SQL pueden ser agrupados de acuerdo a uno o más campos específicos. La sentencia debe ser escrita inmediatamente después de las condiciones dadas por **WHERE** si están, y justo antes de la cláusula **ORDER BY**. El comando agrupa las filas que contienen los mismos valores en columnas especificadas en conjuntos de resumen. Estos agrupamientos pueden usarse para realizar operaciones de agregación.

```
SELECT
    COUNT(first_name)
FROM
    employees
GROUP BY first_name;
```

Este filtro devuelve una lista con el número de veces que cada nombre se repite en toda la lista. Se puede añadir el nombre para que quede más completo:

```
SELECT
    first_name, COUNT(first_name)
FROM
    employees
GROUP BY first_name;
```

De manera más compacta, escribimos un resumen de una solicitud para las tablas mediante **SELECT**:

```
SELECT col_name(s)
FROM table_name
WHERE conditions
GROUP BY col_name(s)
ORDER BY col_name(s)
```

**alias** Se usa para renombrar las columnas que extraigamos mediante alguna operación

```
SELECT
    first_name, COUNT(first_name) AS name_freq
FROM
    employees
GROUP BY first_name;
```

**HAVING** Se utiliza a menudo junto con **GROUP BY** para refinar condiciones, la estructura es

```
SELECT col_name(s)
FROM table_name
WHERE conditions
GROUP BY col_name(s)
HAVING conditions
ORDER BY col_name(s)
```

Es el equivalente al **WHERE** pero aplicado al bloque **GROUP BY**. Un ejemplo es el siguiente

```
SELECT
    first_name, COUNT(first_name) AS name_count
FROM
    employees
GROUP BY first_name
HAVING COUNT(first_name) > 280
ORDER BY first_name
```

Es usual encontrarse en situaciones en las que no es claro la diferencia de usar **HAVING** y **WHERE**.

La cláusula **WHERE**

1. Filtrado de Filas: **WHERE** se utiliza para filtrar filas individuales antes de que se realice cualquier agrupación de datos (**GROUP BY**).

2. Operaciones en Datos Crudos: Funciona directamente sobre los datos crudos (raw data) de las tablas. Por ejemplo, puedes usar **WHERE** para filtrar registros basados en condiciones específicas de las columnas de la tabla
3. No Aplicable a Funciones de Agregación: No puedes usar **WHERE** para filtrar basándote en el resultado de una función de agregación como **SUM()**, **AVG()**, etc.

La cláusula **HAVING**

1. Filtrado de Grupos: **HAVING** se utiliza para filtrar grupos creados por la cláusula **GROUP BY**.
2. Operaciones en Datos Agrupados: Funciona sobre el resultado de las funciones de agregación. Es útil cuando necesitas aplicar condiciones a un conjunto agrupado de registros.
3. Uso Posterior a **GROUP BY**: Se usa después de **GROUP BY** para filtrar grupos según una condición de agregación.

Un ejemplo en el que se puede usar ambos: Extraer una lista de todos los nombres que se encuentren menos que 200 veces. Los datos se deben referir a las personas que fueron contratadas después de primero de enero de 1999.

```
SELECT
    first_name, COUNT(first_name) AS names_count
FROM
    employees
WHERE
    hire_date > '1999-01-01'
GROUP BY first_name
HAVING COUNT(first_name) < 200
ORDER BY first_name DESC;
```

Por último, se puede limitar el número de resultados mediante la cláusula **LIMIT**.

```
SELECT *
FROM salaires
ORDER BY salary DESC
LIMIT 10;
```

### 5.2.9.3 Inserción de datos **INSERT**

Recordemos la sintaxis para insertar datos nuevos:

```
INSERT INTO table_name (col_1, ..., col_n)
VALUES (val_1, ..., val_n)
```

Es posible no especificar la lista de columnas que se insertan, pero en este caso es necesario siempre poner todos los valores (es decir, no omitir ninguno) y agregarlos en el mismo orden en que salen en la tabla.

Utilizando la palabra **INTO** se pueden agregar datos de columnas de una tabla en otra, basándose en alguna condición si es necesario. Puede ser útil en la creación de tablas duplicadas para relizar algún tipo de prueba.

```
INSERT INTO table_2 (col_1, ..., col_n)
SELECT col_1, ..., col_n
FROM table_1
WHERE condition;
```