Contents

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 embarago, 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)
```

Saltará 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 impotante 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 definicion dentro de las líneas subordinadas al `if` anterior solamente existirán dentro del `if`. En python esto no pasa, cualqueir variable que este definida aquí estará dentro del ambito en que se encuentre el `if`.

1.1.1.2 Cómo modificar una variable global

Una cosa importante dentro del tema de los entornos de que siempre que tenemos dos ámbitos diferentes; por ejemplo el entorno global y un entrono 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 exlícitamente 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 recuente el uso de este método de cambio de variables globales, porque se presta para confusiones y facilita de ciera manera los erroes. 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 fórma:

```
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 linea 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 dede 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 numeros 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 oredenamiento, 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() Elimina todo el contenido del diccionario

dic.copy() Retorna una copia del diccionario, sirve para asignar a otra variable y copiar el diccionario.

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, opene
```

- `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 caracter 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. Pr 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 la 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:
    xxxxxxx
    xxxxxx
except (exception_type1):
    xxxxxx
    xxxxx
except (exception_type2):
    xxxxxx
    xxxxx
    .
    .
    .
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 `AssertError`. 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.

1.6 Sobre desarrollo de Software

Cuando se requiere crear un Software, similar a como se planea la construcción de una cada, el primer paso siempre es tener un esquema que qué es exactamente lo que se quiere construir. Cuál es el objetivo principal que dicho Software quiere cumplir. Después de este paso, viene la fase de diseño; está compuesta por los desarrolladores y los arquitectos. Se determina cómo se trabajará, los lineamientos, etc. Una vez el diseño está finalizado, viene la parte del código; la implementación de aquello que se quiere implementar. Cada sub equipo va a realizar pruebas y tests de cada componente. Una vez todos los componentes están listos, es el momento de realizar la unción de dichos componentes, y se realizan las pruebas de integración, pruebas de funcionalidad. Cuando todo está listo, se viene la fase de producción, operación y mantenimiento. Esto implca que el usuario empezará a utilizar el Software, y es cuando pueden venir requerimientos como cambios, mejoras, etc. Las fases se pueden resumir como sigue

- Requierimientos
- Diseño.
- Implementación.
- Verificación.
- Operación y manteniemiento.

En muchas ocasiones puede presentarse situaciones en las que los desarrolladores o los arquitectos pueden malinterpetar los requerimientos del usuario, este malentendido puede extenderse a las fases posteriores. Esto hace importante tener muy en cuenta diferentes modelos que permitan disminuir al máximo este tipo de situaciones. Un concepto importante de algo que parece ser más que un modelo es el llamado 'Agile'.

Se trata de un enfoque de desarrollo que se enfatiza en la flexibilidad, la colaboración y el desarrollo iterativo. Este desarrollo implica la división del proyecto en varias iteracione que típicamente tienen una

duración de una a cuatro semanas. cada iteración se concentra en entregar un incremento en la producctión del trabajo. Esto permite un arealimentación más frecuente y la habilidad de adaptarse y realizar cambios en caso de ser necesario. Agile también promueve la colaboración cercana entre los desarrolladores y el cliente. El cliente está mas involucrado en el proceso de desarrollo, proveyendo retroalimentación y clarificando los requerimientos. También reconoce que los requerimientos y las prioridades pueden cambiar a lo largo del tiempo. En lugar de tratar de predecir y planear cada detalle, la idea es abrazar los cambios y ajustar los planes a medida de las nuevas informaciones. Esto permite una gran flexibilidad, y la habilidad de responder a las necesidades cambiantes de los clientes. Los miembros de los equipos colaboran de cerca, cimparten responsabilidades y toman decisiones de manera colectiva. Siempre se enfocan en el mejoramiento continuo.

1.6.1 Requerimientos

El hecho de que un Software sea un objeto intangible, hace bastante más complicado comunicar ideas de manera exacta, sobre verdaderamente qué es lo que se pretende y delimitar los requerimientos de un software. Un requerimiento tiene esencialmente dos definiciones. El primero es definido como un proceso, en el cual se elabora una idea compartida sobre el problema que existe y eventualmente la potencial solución al mismo. Se construye un conjunto de descripciones de alto nivel de cada parte que compone el probema. El principal objetivo es elaborar un documento que pueda describir detalladamente qué es lo que el sistema deberá hacer y qué es lo que el sistema no deberá hacer. Es impotante tener en cuenta más el 'qué' que el 'cómo', se quiere determinar el comportamiento que tendrá la solución sin tomar decisiones prematuras que puedan afectar la habilidad de diseñar la solución. Este diseño no se realiza todavía en este paso. El segundo concepto de especificación de requerimiento, es el producto de este proceso; la documentación que sale como producto de este proceso.

La especificación de los requerimientos son muy importantes debido a dos aspectos particulares: Por la parte de ingeniería, la importancia recae en el hecho de que así se evita cometer diversos errores que pueden desencadenar en pérdidas de tiempo. Está demostrado que un mayor porcentaje de tiempo invertido en el proceso de especificación de requerimientos da como resultado un mejor porcentaje de costos por imprevistos.

1.6.2 Modelo WRSPM

El modelo WRSPM, también conocido como el modelo mundo-máquina, es un modelo que permite esquematizar y entender los requerimientos del usuario y determinar las especificaciones necesarias de Software para resolver el problema. El modelo consiste en cinco elementos: W (world), R (requirements), S (specifications), P (program), y M (machine). Las suposiciones del mundo son aquellas cosas que ya están dadas por sentadas dentro del universo del probema. Los requerimientos son los objetivos del usuario, aquello que el usuario quiere lograr. Las especificaciones 'S' definen cómo el sistema va a cumplir esos requerimientos. El programa ya está inmerso en el conjunto del sistema, y es, de hecho, el cógido o conjuntos de códigos escritos por los programadores; el programa que cumplirá las especificaciones. Finalmente M es la máquina; el conjunto de hardware que compone la solución.

En este sistema tenemos cuatro variables interesantes: e_h, e_v, s_v, y s_h. e_h son los elementos del entorno que están ocultos al sistema, fuera del sistema, pero aún nos preocupa. Un ejemplo puede ser la tarjeta de crédito que el usuario necesita para poder retirar del cajero. El conjuinto e_v es el de las partes visibles para el sistema en el entorno. En nuestro ejemplo, son los datos generadeos al leer la cinta magnética de la tarjeta de crédito, y el número PIN introducido. EN esencia, cualquier dato que puede ser leído o introducido en el sistema, pues este lo puede leer y es visible. Los s_v son los elementos del sistema que están visibles en el entorno. Esto puede comprender los botones del cajero, la información en pantalla, etc. Finalmente, los s_h son los elementos del sistema que no son visibles para usuarios; que están escondidos internamente en el código, en el hardware, y demás.

1.6.2.1 Ejemplo WRSPM

Un ejemplo para ilustrar los elementos que componen el modelo. Sea un monitor de paciente, capaz de leer signos vitales como fercuencia cardíaca, pulso, presión arterial, etc. El deseo u objetivo del monitor, es tener un sistema de alerta que notifique a la enfermera si el corazón del paciente se detiene. Esto da como requerimiento real el siguiente: si el corazón del paciente se detiene, se debe avisar a la enfermera. Eso se traduce dentro del sistema de la siguiente manera: si el sonido de un sensor cae por debajo de un umbral establecido, se activará

una alarma. Un elemento clave es analizar una de las posibilidades de la parte 'W' del entorno. Se da por sentado que si la alarma suena, siempre habrá una enfermera que escuche y entienda que el corazón se detuvo. Este y muchos otros elementos que se asumen del entorno (y que por tanto no hacen parte del sistema) deben ser cuidadosamente estudiados y tenidos en cuenta dentro de la elaboración de los requerimientos y posteriores especificaciones.

1.6.3 Arquitectura de Software

Tal como sugiere el concepto de arquitectura en construcciones de edificios, un arquitecto es una clase de interfaz entre el cliente, lo que quiere; y el contratista, el implementador, la persona que construye. De manera similar, también es importante mencionar que el tipo de arquitecto debe cambiar y es diferente en función del tipo de producto que se desea diseñar. Como ejemplo en construcciones, el arquitecto de un rascacielos es muy diferente a un arquitecto de una represa, o el de un reactor nuclear. Una definición de arquitectura de software sería la siguiente: Se define como la estructiura de cada componente del software y la manera en la que estos componentes se relacionan entre sí para formar el software como tal. Un elemento clave de este concepto cae en el elemento de particionar el software en partes más pequeñas, independientes, funcionales y con un valor empresarial; de modo que puedan ser fácilmente integrados entre sí para conformar el sistema completo.

1.6.3.1 Modelos de arquitectura

Algunos modelos arquitectónicos como los siguientes son bastante usados e implementados en la industria

- pipe and filter: Este modela de manera secuencial lo que se podría denominar filtros (o transformación) de datos que se conectan a otros filtros mediante los conectores (pipes).
- client-server: Este modelo puede ser fácilmente ejemplificado mediante sistemas basados en internet, como servicios web, sistemas bancarios en internet, etc.
- layers: Es una forma de separar la estructura en diferentes capas independientes entre sí, pero que se correlacionan de manera que el sistema funciona correctamente. Cada capa puede ser modificada sin afectar el funcionamiento del resto de las capas.
- blackboard: Este modelo es un estilo arquitectónico para datos compartidos. Se trata de un sistema o módulo central (puede ser un programa compartido, o una fuente común de información) y un conjunto de componentes que precisan de este módulo central para operar, ya sea mediante la consulta de datos, o mediante procesamiento compartido.

1.6.3.2 Proceso en la arquitectura

El proceso de diseño de una arquitectura se puede desglosar a tres preocupaciones principales:

- Estructura del sistema. Se refiere a cómo el sistema se descompone en estos varios subsistemas principales, y cómo estos se comunican entre sí.
- Modelamiento de control. Es la forma en la que la arquitectura realiza un modelo de las relaciones de control entre las diferentes partes del sistema
- Descomposición modular. Es la forma en que se identifican las particiones de los subsistemas

Otra cosa importante a nivel arquitectónico es cómo podemos evaluar la calidad de un software. Los sisguientes son algunos atributos de calidad que se tienen en cuenta para el Software:

1. Desempeño
2. Confiabilidad

Chapter 2

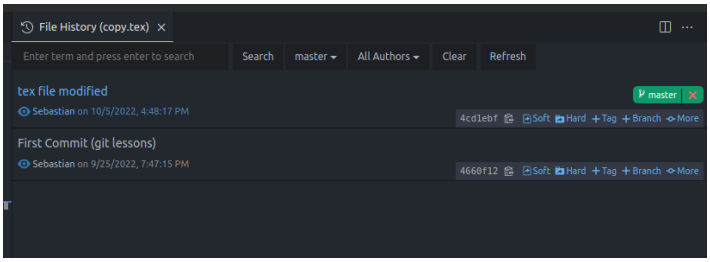
Git and github

2.1 Algunas extensiones de VSCode para Git

2.1.1 Git History

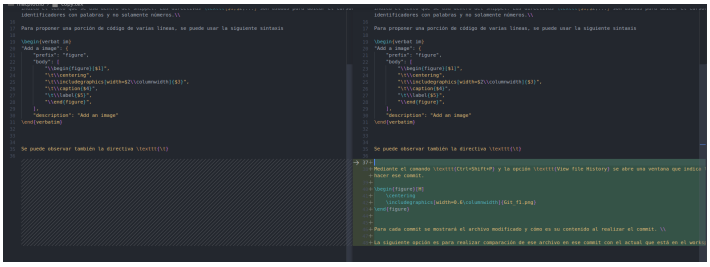
Es una extensión que sirve para visualizar los cambios históricos que se han hecho a los diferentes archivos.

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.

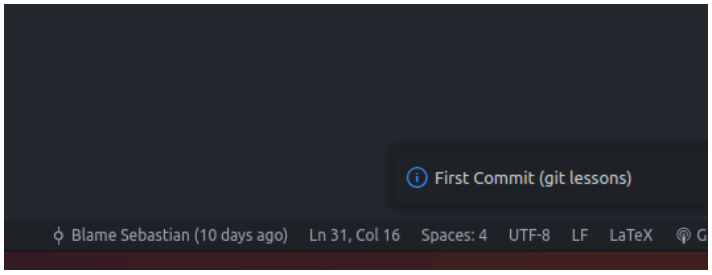
La siguiente opción es para realizar comparación de ese archivo en esa versión con la actual que está en el espacio de trabajo.



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

2.1.2 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



2.1.3 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.

2.2 Fundamentos de Git: cómo funciona por dentro

2.2.1 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`.

2.2.2 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.

2.2.2.1 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.

2.2.2.2 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.

2.2.2.3 Commit

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

2.2.2.4 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.

2.2.3 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.

2.2.4 Hash

Al realizar el comando de la seccion anterior, vimos que ek string "`Hello, Git`" generó un hash `b7aec520dec0a7516c18eb4c68b`
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 funccióon 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 fuciones 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 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 $16^{40} \approx 1.46\ 10^{48}$. Por otro lado podemos havernos la pregunta de cual es la posibilidad de que dos archivos diferentes produzcan el mismo hash?

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

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

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.

2.2.5 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.

2.2.6 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 promera opvion es para romar la intrada como entrada estandar, 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 archibos 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. Luefo del delimitador está el contenido del archivo.

2.2.7 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.

2.2.8 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.

2.2.9 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 file `file2.txt`; en este caso, el contanido 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.

2.2.10 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 pasa por el area 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 esta 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 archibvos 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 area de staging, usamos el comando `git read-tree <hash>`.

2.2.11 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 ateniendo 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 agrege todos los archivos.

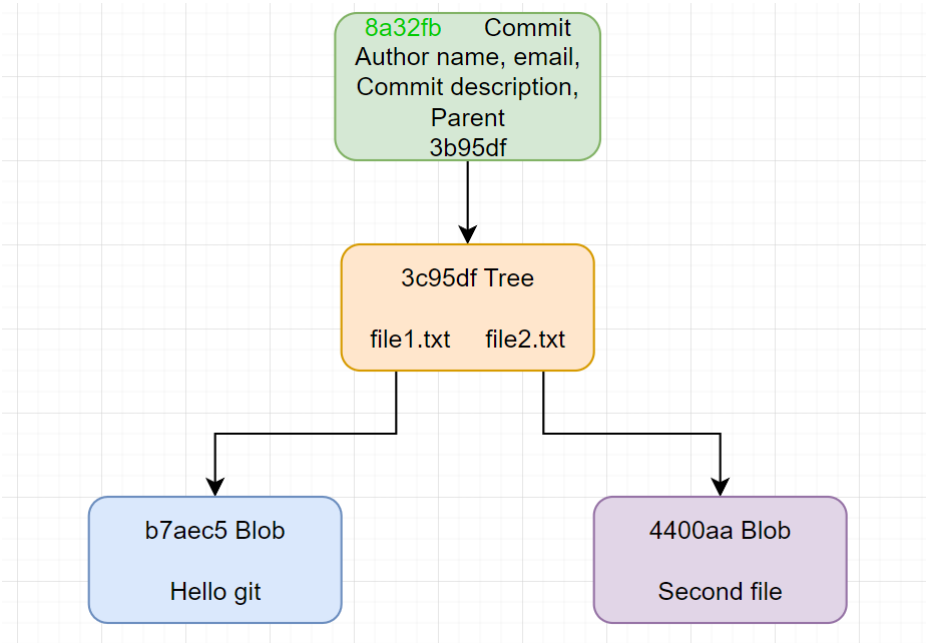
2.3 Operaciones básicas de Git

2.3.1 ¿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 es tipo de objeto + tamaño + delimitador + contenido.

La diferencia esta 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`

2.3.2 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

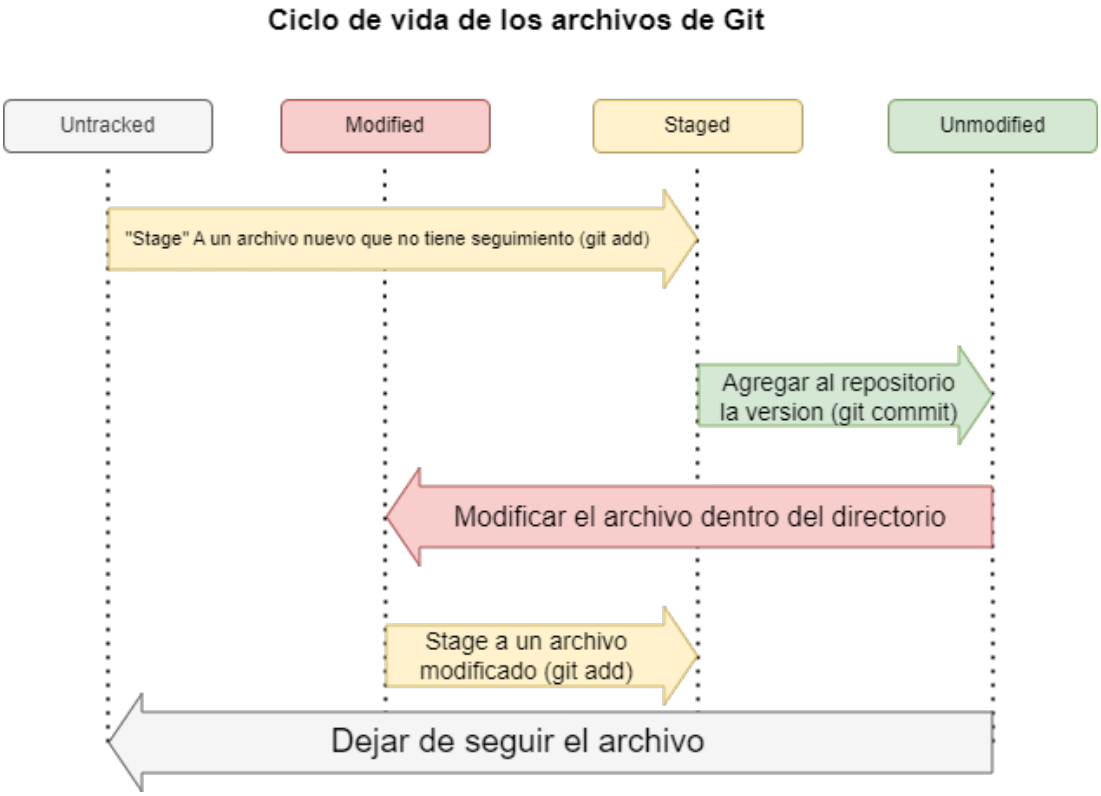
2.3.3 Comandos básicos de git

- `git status`
- `git add` con este se agregan aarchivos al area de staging
- `git commit` con este se 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 varis 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 area 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 este tendrá uno u otro commit padre (más adelante se verá que para pull requests pueden haber punteros distintos.

2.3.4 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/Proc
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
```

2.4 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 especifica 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:

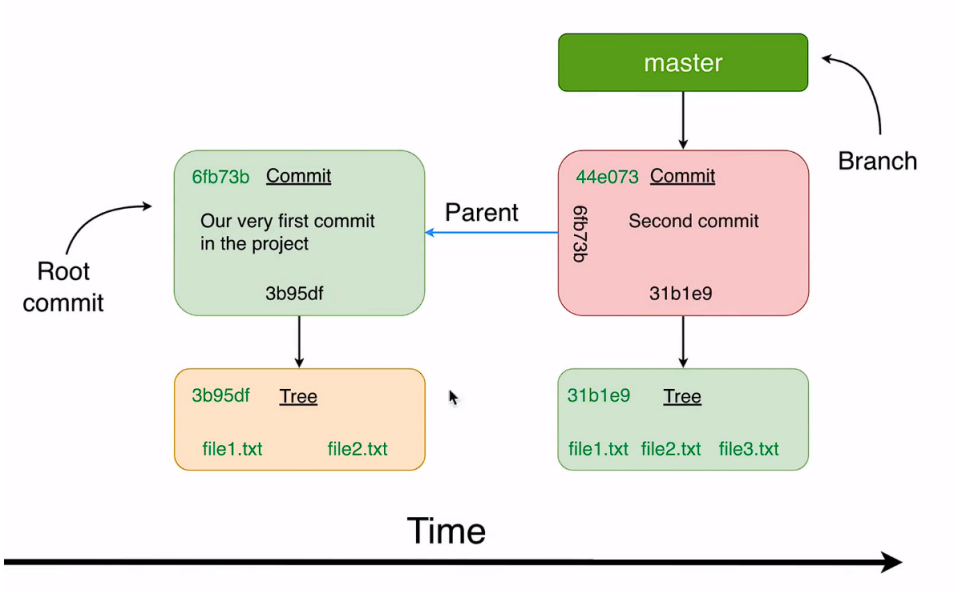
1. La rama por defecto es la master.
2. En un mismo repositorio pueden existir varias ramas.
3. Los apuntadores a todas las ramas se localizan en `.git/refs/heads`
4. Cada rama maneja sus propios commits.
5. El puntero de la rama se mueve automáticamente después de cada nuevo commit.
6. Para cambiar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

2.4.1 HEAD

El concepto de head es util 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
----                 -------------         ------ ----
la---           3/25/2023  3:49 PM             41 master
la---            9/6/2023  9:31 AM             41 speechGen
la---           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`

2.5 Repositorios remotos

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

2.5.1 git diff

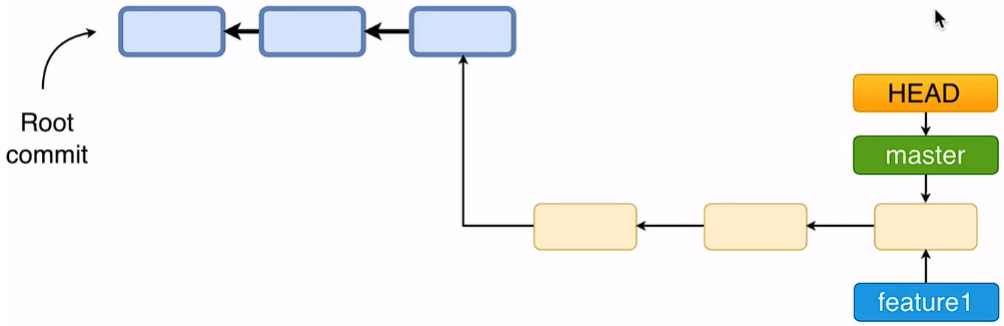
Este es un comando que puede ser útil para ver y evidenciar las diferencias entre un archivo modificado su anterior versión dentro de la consola. Al usar el comando podemos ver el hash provisional del nuevo archivo (el que tendria si se realiza el commit), las líneas agregadas- quitadas-modificadas.

2.6 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.

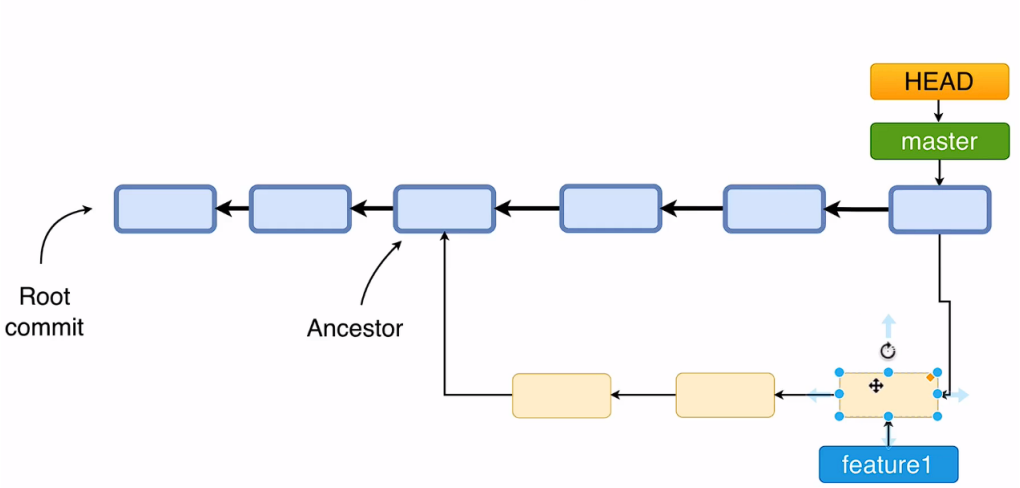
```
git merge <feature-branch>
```

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.

2.6.1 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:

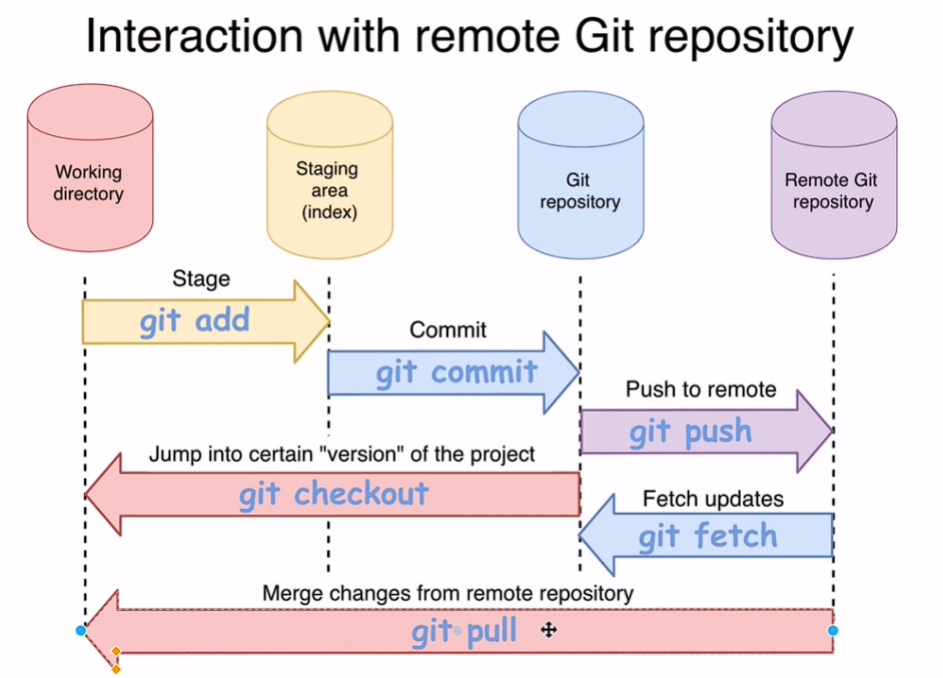
Abrimos el archivo que contiene los conflictos mediante `nano` por ejemplo y seleccionar manualmente cuál de la(s) líneas van a ser conservadas. guardar el archivo y de esta manera el o los conflictos habrán sido resueltos. También se tiene a opción de incluso volver a modificar el archivo si ninguna 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).

2.7 Comandos para los repositorios remotos

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

2.7.1 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.



2.7.2 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 stagig como el directorio de trabajo se actualizan automáticamente luego de la fusión.

Cuando uno clona un repositorio remoto en uno local, git automaticamente 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 unformación 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.

2.7.3 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.

2.7.4 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 fudión 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 tona todas las actualizaciones del repositorio remoto y las escribe en el repositorio local. Esto incluye todas las nuevas reamas y versiones creadas en el repositorio remoto.

Luego de 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.

2.7.5 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/JhoAraS
629fab88a1c5954a9389c827f0e7b4829ce734ba        not-for-merge   tag '1' of https://github.com/JhoAra
```

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". Despues 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.

2.7.6 Git pull con modificacion de repositorio remoto

Si despues 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`

2.7.7 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 c0ommit 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`

2.7.8 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.

2.7.9 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
```

2.7.9.1 comando `git show-ref`

Este comando es putil 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.

2.8 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.

2.8.1 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 3

Essentials for Cibersecurity

La guía de manejor de incidentes en la seguridad de sistemas NIST 800-61 indica un proceso de respuesta ante incidentes.

El primer paso para cualquier respuesta a incidentes es la preparación. Es muy importante tener la seguridad y la certeza de cómo se debe actuar ante un incidente. Es necesario tener muy bien establecido un proceso de respuesta par acad atipode incidente. Una vez se presenta el incidente, el siguiente paso para el manejo es la detección y el análisis; cosas como determinar el nivel de criticidad del incidente, cuál es el tipo de impacto que potencialmente se tiene con este incidente, si se trata de un positivo real o un falso positivo. El siguiente paso es la contención, erradicación y recuperación; la contención implica el aislamiento, de ser posible, de los dispositivos afectados. Es necesario a veces volver al paso anterior de análisis y detección, porque dentro de la mitigación se pueden determinar nuevos elementos no detectados anteriormente y ser analizados. Finalmente el último paso es la actividad post-incidente: esta está estrechamente relacionada con el primer paso de preparación porque en este paso se obtienen nuevos elementos y lecciones que se implementan en la preparación par evitar nuevos incidentes.

3.0.1 Herramientas de gestión

Es importante mencionar las herramientas más implementadas dentro de un equipo de operaciones de seguridad.

3.0.1.1 EDR

El significado de EDR es (Endpoint Detection and Response). Es una tecnología dieñada para monitorear continuamente los endpoint (que son dispositivos finales como computadoras, teléfonos y dispositivos IoT) con el objetivo de realizar detección y respuesta ante diferentes incidentes de seguridad. Es una tecnología que recopila datos de actividad de los enspoints, los analiza para identificar patrones de amenazas y si se detecta una amenaza, responder automáticamente en tiempo real para contener o eliminar la amenaza y notificar al personal de seguridad. Las principales funcionalidades que puede desempeñar un EDR son las siguientes:

- Monitoreo y análisis en tiempo real
- Detección de amenazas
- Respuesta automática
- Análisis forense

3.0.1.2 XDR

El XDR significa Respuesta y detección extendida. La principar diferencia con EDR es su alcance y capacidad de integración. EDR se centra exclusivamente en la protección y respuesta a incidentes en los endpoints. XDR amplía el enfoque al integrar la detección y erspuesta a través de últiples coponentes de la infraestructura, incluyendo endpoints, redes, servidores, aplicaciones den la nube, etc. Con un análisis enfocado en la correlación de datos y análisis de eventos en toda la infraestructura, y una respuesta coordinada y atutomatizada.

3.0.1.3 SIEM

Security Information and Event Management. Es un sistema centralizado cuyo objetivo es recolectar, correlacionar y analizar todos los datos como logs y eventos de todos los incidentes de seguridad. Se centra en la agregación de datos y posterior análisis. Esta tecnología implementa reglas predefinidas y personalizables para correlacionar eventos y detectar posibles incidentes de seguridad. Genera alertas en tiempo real cuando se detectan eventos que coinciden con las reglas de correlación permitiendo una respuesta rápida a incidentes.

3.0.1.4 SOAR

Es una tecnología cuyo objetivo es implementar automatizaciones y mejorar la eficiencia y eficacia de las operaciones. Permite la integración de múltiples herramientas y sistemas de seguridad, facilitando la coordinación de flujos de trabajo de seguridad a través de diferentes sistemas y equipos. Implementa la automatizacón de procesos repetitivos y rutinarios, tales como la recopilación de datos, anáñlisis de eventos y respuesta inicial a incidentes.

Normalmente los equipos de SOC en grandes empresas se dividen en dos sub equipos más pequeños: Equipo azul y equipo Rojo. El equipo azul se encarga mayormente de realizar las tareas más típicas de analista de SOC: monitoreo de seguridad, respuesta a incidentes, análisis forense, seguimiento de amenazas, etc. Por otro lado, el equipo Rojo está más enfocado en la evaluación de vulnerabilidades, testeo de penetración, ingenieía social, y en general, simulación de procedimientos, técnicas y tácticas de adversarios.