Introducción a PyQGIS

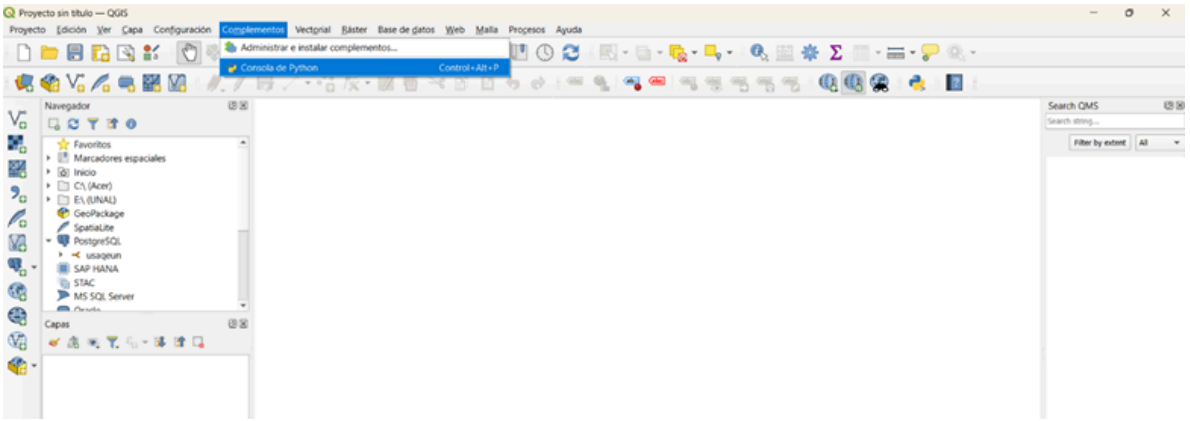
La capacidad de manipular y analizar datos geográficos es una habilidad fundamental en el mundo actual. QGIS se ha consolidado como una herramienta indispensable para profesionales y estudiantes en este campo. Sin embargo, para llevar el análisis y la gestión de información geográfica a un nivel superior, es esencial ir más allá de la interfaz gráfica. Aquí es donde PyQGIS entra en juego, ofreciendo una combinación poderosa de QGIS y Python, un lenguaje de programación reconocido por su facilidad de aprendizaje y versatilidad.

PyQGIS es, en esencia, la puerta para dotar a QGIS de "superpoderes", permitiendo la automatización de tareas repetitivas y la creación de herramientas personalizadas que se adaptan a necesidades específicas. Dominar PyQGIS no es meramente adquirir una habilidad técnica; representa una oportunidad para optimizar significativamente el tiempo dedicado a proyectos, manejar grandes volúmenes de datos con mayor eficiencia y desarrollar soluciones GIS innovadoras que trascienden las limitaciones de las funciones predefinidas. La posibilidad de automatizar tareas complejas transforma la forma en que se abordan los proyectos. Por ejemplo, procesar cientos de archivos de forma manual podría llevar horas o días, pero con un script de PyQGIS, esta tarea se reduce a minutos

Preparando tu Entorno: La Consola de Python en QGIS

Antes de sumergirse en la escritura de código, es fundamental comprender dónde se escribe y ejecuta PyQGIS dentro de QGIS. El software integra una "Consola de Python", que funciona como un laboratorio interactivo donde se pueden probar comandos de PyQGIS de manera inmediata.

Para acceder a esta consola, los usuarios deben navegar en QGIS al "Menú Complementos > Consola de Python". Una vez abierta, la consola se presenta con tres áreas principales: el área de entrada de comandos, donde se introduce el código; el área de salida, que muestra los resultados de la ejecución y cualquier mensaje de error; y el editor de scripts, diseñado para escribir y gestionar códigos más extensos. Al introducir un comando y presionar Enter, el resultado o cualquier error se visualiza de inmediato



Explorando tus Datos GIS con Código

Para que PyQGIS pueda realizar operaciones útiles, primero necesita "ver" y acceder a los datos geográficos cargados en QGIS. Esto implica aprender a referenciar el proyecto QGIS actual y las capas que contiene. Es como proporcionar a PyQGIS las "coordenadas" para localizar y manipular los mapas y sus componentes.

Accediendo a Capas y Proyectos

El punto de partida para interactuar programáticamente con QGIS es el objeto *iface*. Este objeto actúa como el "control remoto" principal, permitiendo el acceso a casi todos los elementos visibles en la interfaz de QGIS, desde el mapa hasta las capas. A través de *iface*, se puede interactuar con el proyecto actual, las capas cargadas y la vista del mapa.

Para trabajar con las capas, es necesario obtener una referencia al proyecto QGIS que se tiene abierto. Esto se logra utilizando *QgsProject.instance()*, que devuelve una instancia del proyecto actual, indicando a PyQGIS que opere con el mapa que se está visualizando

Carga de Capas Vectoriales

Cargar capas vectoriales es una de las primeras cosas que aprendemos a hacer en QGIS, y con PyQGIS, podemos hacerlo de manera programática, lo que es genial para automatizar procesos. Hay varias formas de hacerlo, dependiendo del tipo de archivo y de cómo queramos manejar la capa en nuestro proyecto.

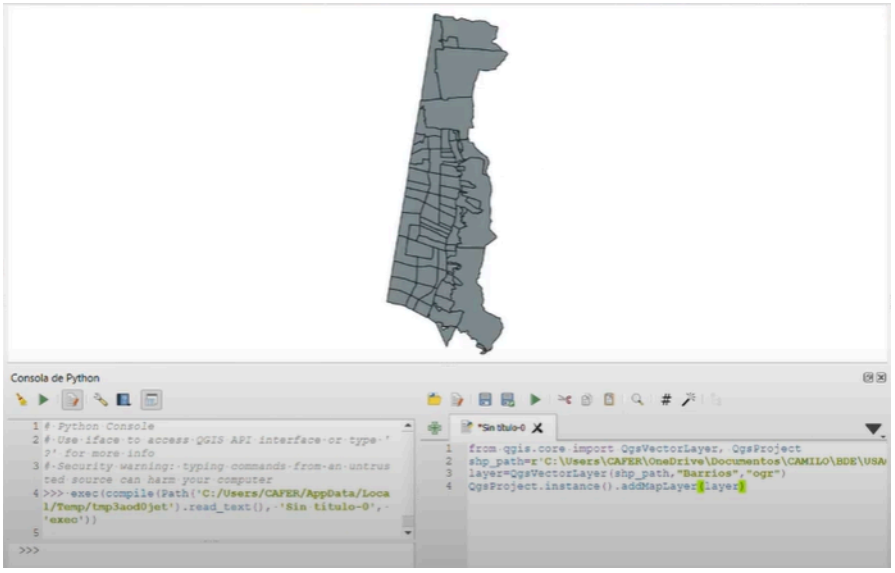
Método QgsProject.instance().addMapLayer()

A veces, no queremos que la capa se añada directamente a la interfaz de QGIS al crearla, sino que preferimos crear el objeto *QgsVectorLayer* primero y luego añadirlo explícitamente al proyecto. Esto es muy útil cuando trabajamos con capas temporales o generadas en memoria.

El objeto *QgsProject.instance()* es la representación del proyecto QGIS actual. Es como el cerebro central de nuestra sesión de QGIS, y nos permite interactuar con todas las capas cargadas y la configuración del proyecto.

Esta forma de añadir capas nos da más control, especialmente cuando generamos datos sobre la marcha o necesitamos manipular la capa antes de que se muestre en el lienzo del mapa.

En el ejemplo practico podemos evidenciar como mediante el script de Python podemos adicionar la capa de barrios de la localidad de Usaquén



from qgis.core import QgsVectorLayer, QgsProject
shp_path=r'C:\ruta\de\la\capa.shp'
layer=QgsVectorLayer(shp_path, "Barrios", "ogr")
QgsProject.instance().addMapLayer(Layer)

!!!Es importante que en el script donde creamos la ruta de la capa '*shp_path'* pongamos la ruta verdadera donde tengamos guardado el archivo¡¡¡

Cuando llamamos a *addVectorLayer()*, QGIS hace un montón de cosas en segundo plano: verifica la ruta de la capa, intenta abrirla con el proveedor de datos que le indicamos, la añade al proyecto con el nombre que le damos, y se asegura de que se redibuje el mapa para que podamos ver la nueva capa.

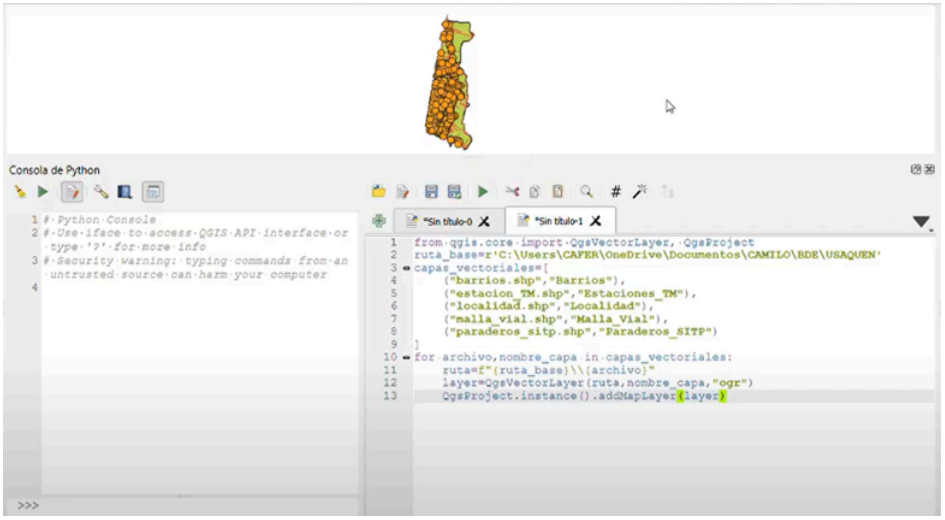
Esta función necesita tres parámetros principales:

1. ***uri (Uniform Resource Identifier):*** Este es el camino a tu archivo o la cadena de conexión a tu base de datos. ¡Ojo! No siempre es solo la ruta del archivo. Puede ser una cadena más compleja que especifique una subcapa dentro de un archivo (como un GeoPackage) o detalles de conexión a una base de datos.
2. ***baseName:*** El nombre que aparecerá en el panel de capas de QGIS. Puedes elegir el que quieras.
3. ***providerKey:*** El tipo de proveedor de datos. Para la mayoría de los formatos vectoriales comunes como Shapefiles y GeoPackages, el proveedor es *"ogr"*. Otros proveedores pueden ser *"postgres"* para bases de datos PostgreSQL, *"delimitedtext"* para archivos CSV, etc..

La función *addVectorLayer()* devuelve un objeto *QgsVectorLayer,* que es la representación de nuestra capa en Python. Es importante guardar este objeto en una variable para poder interactuar con la capa más tarde.

Cargar varias capas vectoriales

Para optimizar el proceso de añadir varias capas vectoriales, es prácticamente el mismo proceso que añadir una sola capa, pero para no hacerlo capa por capa, lo que procede es crear una lista con las capas que pensamos añadir, donde ira especificado el nombre del shape, y el nombre que aparecerá en el panel de capas de QGIS, y mediante un ciclo for añadiremos capa por capa




```
from qgis.core import QgsVectorLayer, QgsProject
ruta_base=r'C:\ruta\de\la\carpeta\donde\estan\los\shapes'
capas_vectoriales=[
        ("barrios.shp", "Barrios"),
        ("estacion TM.shp", "Estaciones TM"),
        ("localidad.shp", "Localidad"),
        ("malla vial.shp", "Malla Vial"),
        ("paraderos_sitp.shp", "Paraderos_SITP")
]
for archivo,nombre_capa in capas_vectoriales:
        ruta=f" (ruta_base)\\(archivo)"
        Layer=QgsVectorLayer (ruta,nombre_capa, "ogr")
        QgsProject.instance().addMapLayer(layer)
```

Carga de Capas Ráster

Cargar capas ráster es muy similar a cargar capas vectoriales, pero con algunas diferencias clave en los parámetros y el proveedor de datos.

Método QgsProject.instance()

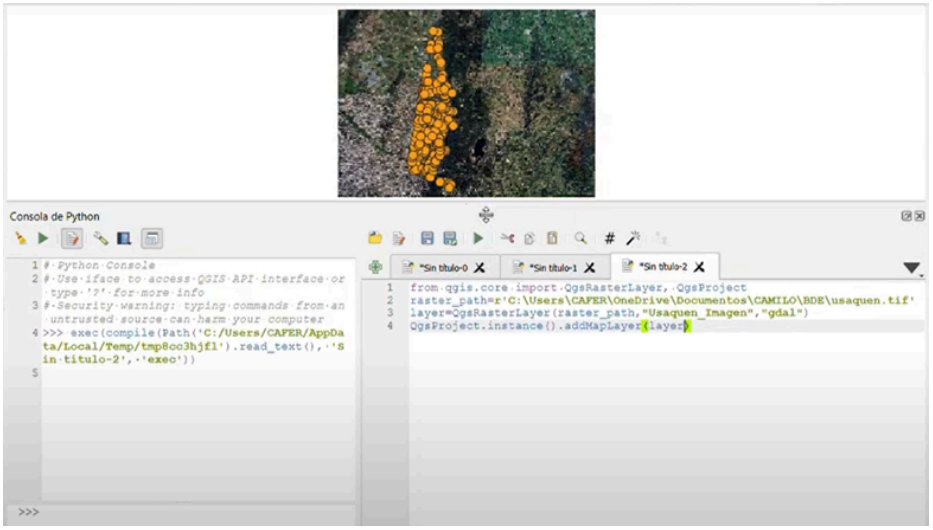
Cuando llamamos a *addRasterLayer()*, QGIS hace un montón de cosas en segundo plano: verifica la ruta de la capa, intenta abrirla con el proveedor de datos que le indicamos, la

añade al proyecto con el nombre que le damos, y se asegura de que se redibuje el mapa para que podamos ver la nueva capa.

Esta función necesita tres parámetros principales:

1. ***uri (Uniform Resource Identifier):*** Este es el camino a tu archivo o la cadena de conexión a tu base de datos.
2. ***baseName:*** El nombre que aparecerá en el panel de capas de QGIS. Puedes elegir el que quieras.
3. ***providerKey:*** El tipo de proveedor de datos. Para la mayoría de los ráster, especialmente los GeoTIFF, el proveedor es *"gdal"*

La función *addRasterLayer()* devuelve un objeto *QgsRasterLayer,* que es la representación de nuestra capa en Python. Es importante guardar este objeto en una variable para poder interactuar con la capa más tarde.



from qgis.core import QgsRasterLayer, QgsProject

raster_path=r'C:\ruta\del\raster.tif'

layer=QgsRasterLayer(raster_path, "Usaquen_Imagen", "gdal")

QgsProject.instance().addMapLayer(Layer)

!!!Es importante que en el script donde creamos la ruta de la capa '*raster_path'* pongamos la ruta verdadera donde tengamos guardado el archivo¡¡¡

Verificación de la validez de la capa

Tanto los objetos *QgsVectorLayer* como *QgsRasterLayer* tienen una función *isValid()*. Usar esta función es una buena práctica de programación. Nos permite asegurarnos de que la capa se cargó correctamente antes de intentar realizar cualquier operación con ella. Si un archivo no existe, está corrupto o la ruta es incorrecta, *isValid()* devolverá False, lo que nos ayuda a depurar nuestros scripts y evitar errores inesperados.

Guardar Proyectos QGIS

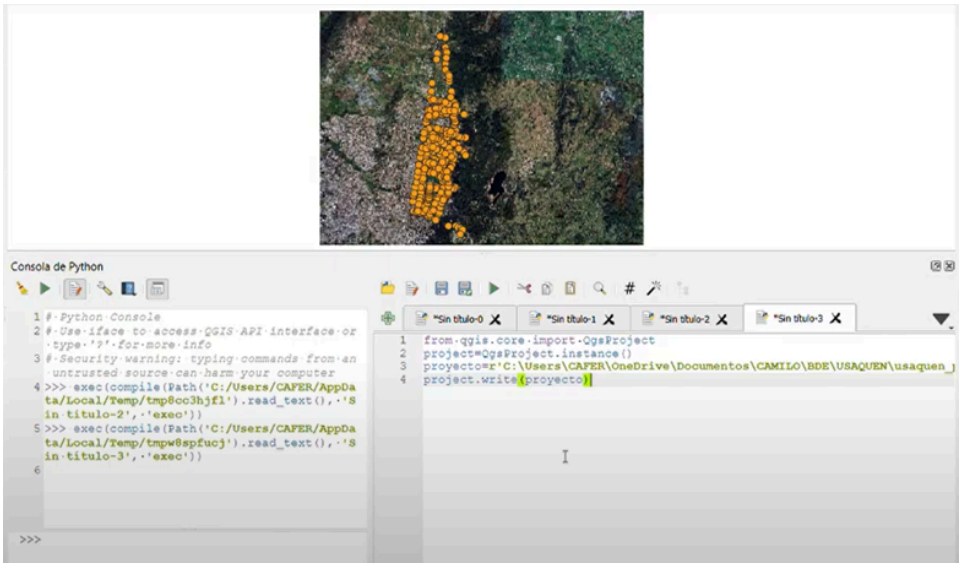
Guardar el proyecto es esencial para conservar nuestro trabajo. PyQGIS nos permite guardar el proyecto actual, ya sea desde cero o en una nueva ubicación.

El objeto *QgsProject.instance()* es, de nuevo, clave aquí. Este objeto representa el estado completo de tu proyecto QGIS, incluyendo todas las capas añadidas, sus propiedades, la simbología, las vistas de mapa y los diseños de impresión.

Guardar un proyecto desde cero

Cuando iniciamos un nuevo proyecto en QGIS, su título es "Untitled Project" (Proyecto sin título) hasta que lo guardamos. Para guardar este proyecto por primera vez, simplemente llamamos al método *write()* del objeto *QgsProject* y le pasamos la ruta completa y el nombre del archivo *.qgs* donde queremos guardarlo.

El método write() devuelve True si la operación fue exitosa y False si hubo algún problema. Esto es muy útil para saber si nuestro script funcionó como esperábamos



from qgis.core import QgsProject
project=QgsProject.instance()
proyecto=r'C:\ruta\donde\queremos\guardar\el\proyecto.qgs'
Project.write(proyecto)

Guardar un proyecto en una nueva ubicación

Si ya tenemos un proyecto abierto y queremos guardarlo con un nuevo nombre o en una ubicación diferente (equivalente a "Guardar como..."), usamos el mismo método *write()*,

pero especificando la nueva ruta. Si no se proporciona una ruta, *project.write()* simplemente guarda los cambios en el archivo de proyecto actual.

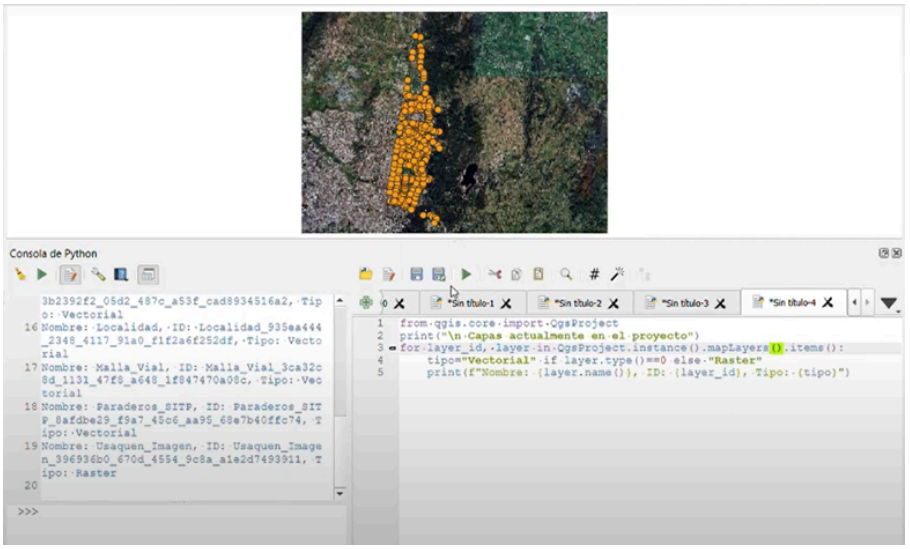
Esta flexibilidad para guardar proyectos en diferentes ubicaciones es fundamental para la gestión de versiones y para asegurar que los resultados de nuestros scripts se almacenen de manera organizada, sin sobrescribir el trabajo original

Visualización de Características de Capas (Nombre e ID)

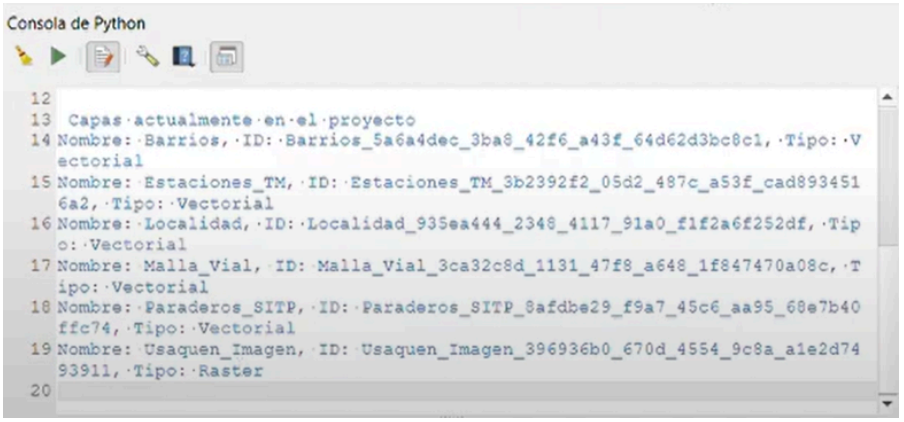
Una vez que tenemos nuestras capas cargadas, a menudo necesitamos acceder a sus propiedades, como su nombre o su ID único, para poder referirnos a ellas en nuestro código.

Listar todas las capas y sus propiedades

A veces, es útil obtener una lista de todas las capas cargadas en el proyecto y sus nombres o IDs. El objeto *QgsProject.instance().mapLayers().items()* nos da acceso a todos los objetos de capa en el proyecto actual.

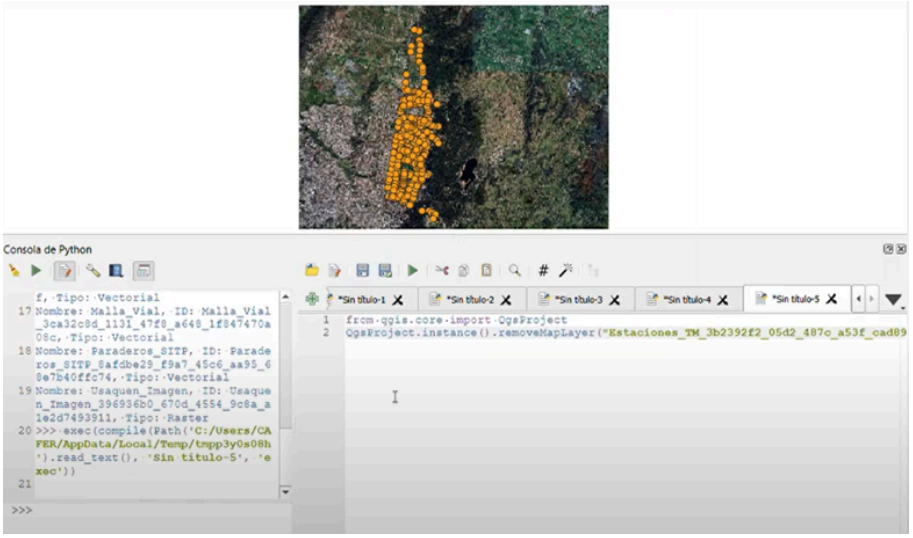


from qgis.core import QgsProject
print("\n Capas actualmente en el proyecto")
for layer.id, layer in QgsProject.instance().mapLayers().items() :
tipo="Vectorial" if layer.type() == 0 else "Raster"
print(f"Nombre: {layer.name()}, ID: {layer_id}, Tipo: {tipo}")



Esta capacidad de listar y acceder a las propiedades de las capas es fundamental para escribir scripts que interactúen dinámicamente con el contenido del proyecto QGIS, permitiendo automatizar tareas que de otro modo requerirían la interacción manual del usuario con el panel de capas.

Como el eliminar una capa utilizando su ID mediante la función *removeMapLayer()*



from qgis.core import QgsProject
QgsProject.instance().removeMapLayer("ID de la capa que vayamos a eliminar")

Características de Capas Ráster

Así como con las capas vectoriales, podemos obtener mucha información sobre nuestras capas ráster usando PyQGIS. Esto incluye el número de bandas, la extensión espacial, el Sistema de Referencia de Coordenadas (CRS) y el tipo de datos de los píxeles.

Para acceder a estas propiedades, trabajamos con el objeto *QgsRasterLayer* que obtenemos al cargar la capa, y a menudo, con su *dataProvider()*. El *dataProvider()* es el componente que realmente sabe cómo leer los datos del archivo ráster.

Conteo de bandas

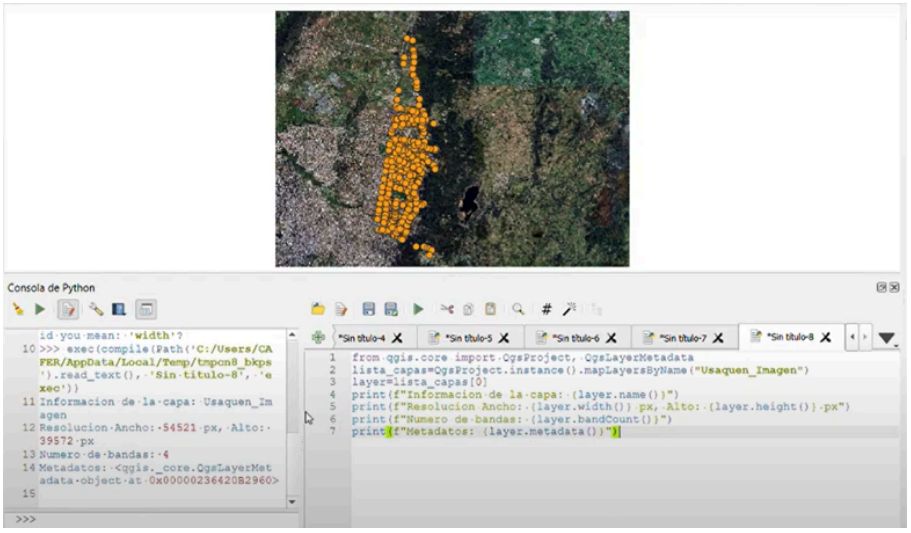
Los rásteres pueden tener una o varias bandas (por ejemplo, rojo, verde, azul, infrarrojo). Podemos saber cuántas bandas tiene una capa ráster con el método *bandCount()*.

Extensión de la capa

La extensión de una capa ráster define su área geográfica cubierta. Se obtiene con *extent()*, que devuelve un objeto QgsRectangle. Para verlo de forma legible, usamos *toString()*.

Tamaño en pixeles de la capa

El tamaño de la capa se conoce mediante la cantidad de pixeles que tiene en cuanto a lo ancho y lo largo de la imagen, esto lo podemos identificar mediante *.width()* y *.height()*



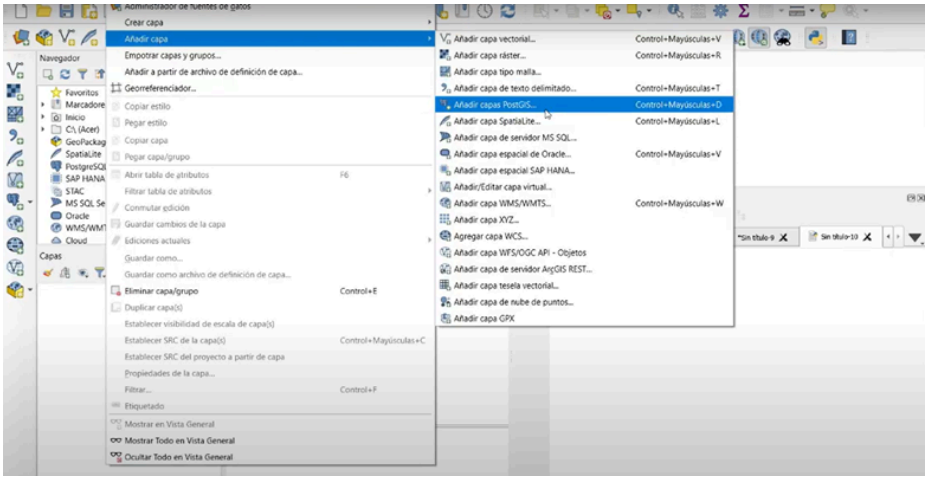

```
from qgis.core import QgsProject, QgsLayerMetadata
lista_capas = QgsProject.instance().mapLayersByName('Usaque_Imagen')
layer = lista_capas[0]
print(f"\n Información de la capa: {layer.name()}")
print(f" Resolución: Ancho = {layer.width()} px, Alto = {layer.height()} px")
print(f"Extensión (QgsRectangle): {layer.extent()}")
print(f"Número de Bandas: {layer.bandCount()}")
print(f"Metadatos: {rlayer.metadata().toString()}")
```

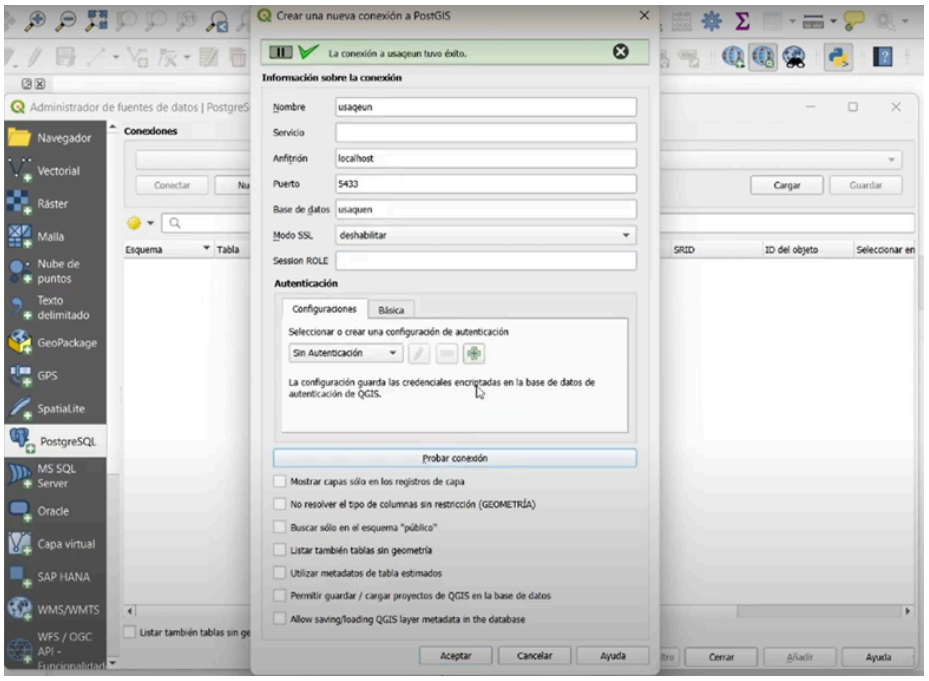
Carga de Capas desde PostgreSQL

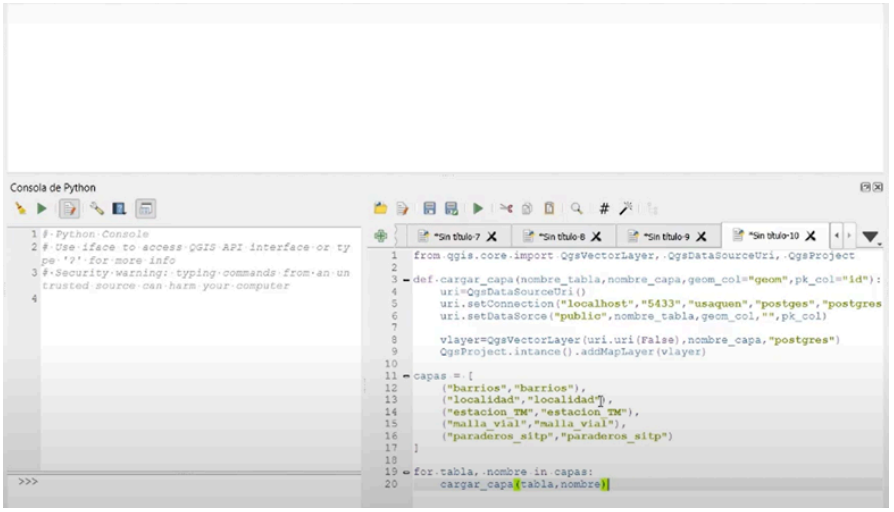
Cargar capas directamente desde una base de datos PostgreSQL con extensión PostGIS es una capacidad muy potente de PyQGIS, especialmente cuando trabajamos con grandes volúmenes de datos o en entornos multiusuario. Usamos la clase *QgsDataSourceUri* para construir la cadena de conexión a la base de datos.

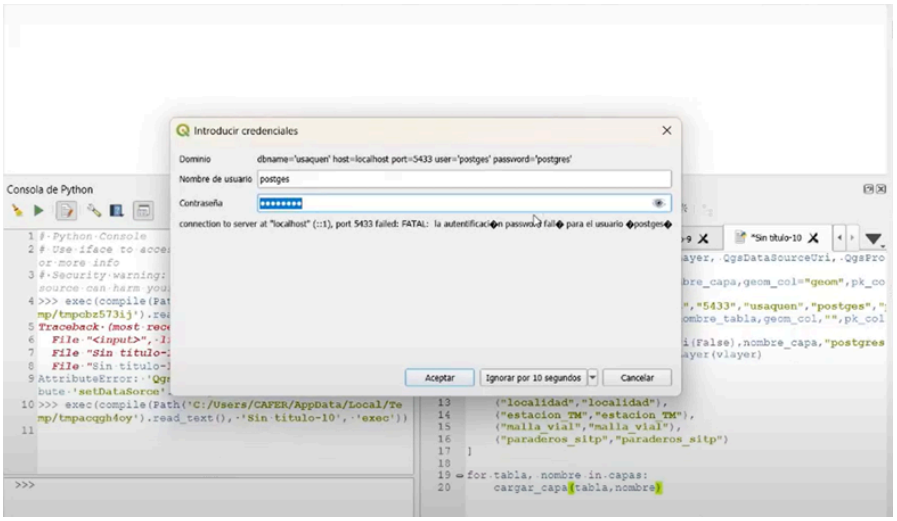
Conexión básica a PostGIS

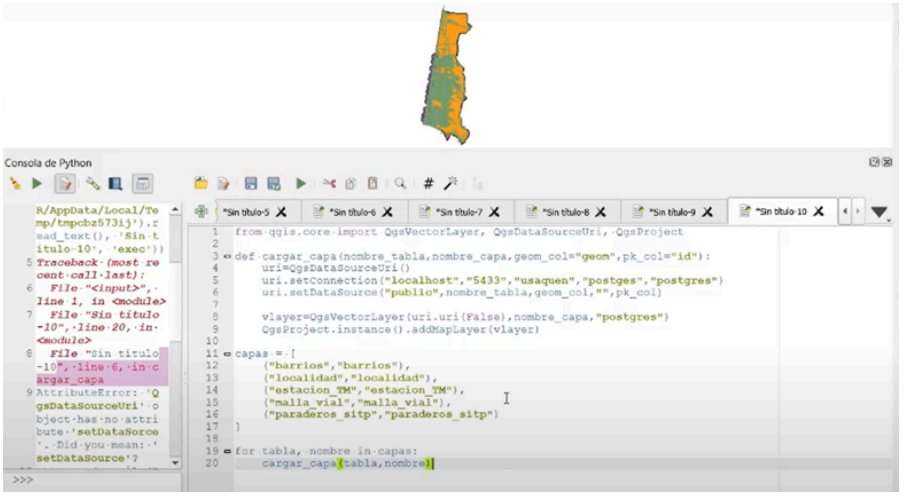
Primero, necesitamos configurar los parámetros de conexión a la base de datos: el host, el puerto, el nombre de la base de datos, el usuario y la contraseña. Luego, especificamos el esquema, la tabla y la columna de geometría.












```
from qgis.core import QgsVectorLayer, QgsDataSourceUri, QgsProject
def cargar_capa(nombre_tabla, nombre_capa, geom_col="geom", pk_col="id"):
        uri = QgsDataSourceUri()
         uri.setConnection("localhost", "5433", "usaquen", "postgres", "postgres")
```

```
        uri.setDataSource("public", nombre_tabla, geom_col, "", pk_col)

        vlayer = QgsVectorLayer(uri.uri(False), nombre_capa, "postgres")
        QgsProject.instance().addMapLayer(vlayer)

capas = [
        ("barrios", "barrios")
   ("localidad", "Localidad"),
   ("estacion_TM", "Estaciones TM"),
   ("malla_vial", "Malla Vial"),
   ("paraderos_sitp", "Paraderos SITP"),
]

for tabla, nombre in capas:
   cargar_capa(tabla, nombre)
```

Los documentos necesarios como los archivos shape, la imagen raster y la base de datos postgresql se encuentran en los documentos adjuntos, esto con el fin de ir practicando según los ejemplos plasmados en este manual