Tutorial Level 0

April 2023

Contents

1 Install PyCharm **2**

2 Install Miniconda **2**

3 Set up Hub environment and pip install (Cerc-Hub) **2**

4 Basic geometry import **3**

5 Export back to obj **4**

6 Supported formats **6**

6.1 XML 6
6.2 JSON 7
6.3 GeoJSON 7
6.4 CityGML 7
6.5 OBJ 7
6.6 Rhino 7

A Configure Python Interpreter **8**

The tutorial level_0 is the first document of the tutorial series. In this tutorial, we will import a geometry file, create a simple city, explore the details of buildings in the city (for example, floor area), and export the city back to the obj format. Before anything, a Python editor and Miniconda should be installed. Details are explained in subsections 1 and 2. Next, creating an environment and installing Hub are discussed. Supported formats in the Hub are introduced and briefly explained in the last part.

1 Install PyCharm

You will need a python editor in order to import the existing Hub source code and write your own python code. Whilst this is a personal choice, we recommend PyCharm Community Edition, an excellent open-source python editor.
Run the installer and follow the installation instructions for PyCharm; you may change a few options, but the default ones should be fine.

2 Install Miniconda

You can find the latest Miniconda installer links to different operator systems in this link. After downloading the file, click on .exe file and follow the instruction. To test Miniconda's installation, open a terminal window and run the **conda list** command. A list of installed packages appears if it has been installed correctly.

3 Set up Hub environment and pip install (Cerc-Hub)

To setup a conda environment named Hub, open a command prompt (in Windows) and type:
conda create -n Hub python=3.9 This step is shown in figure 1
This command will create a new conda environment called Hub with a specific Python version (3.9). To verify this step, type the following command: **conda env list.** Run the code and a list of available environments including Hub is printed.

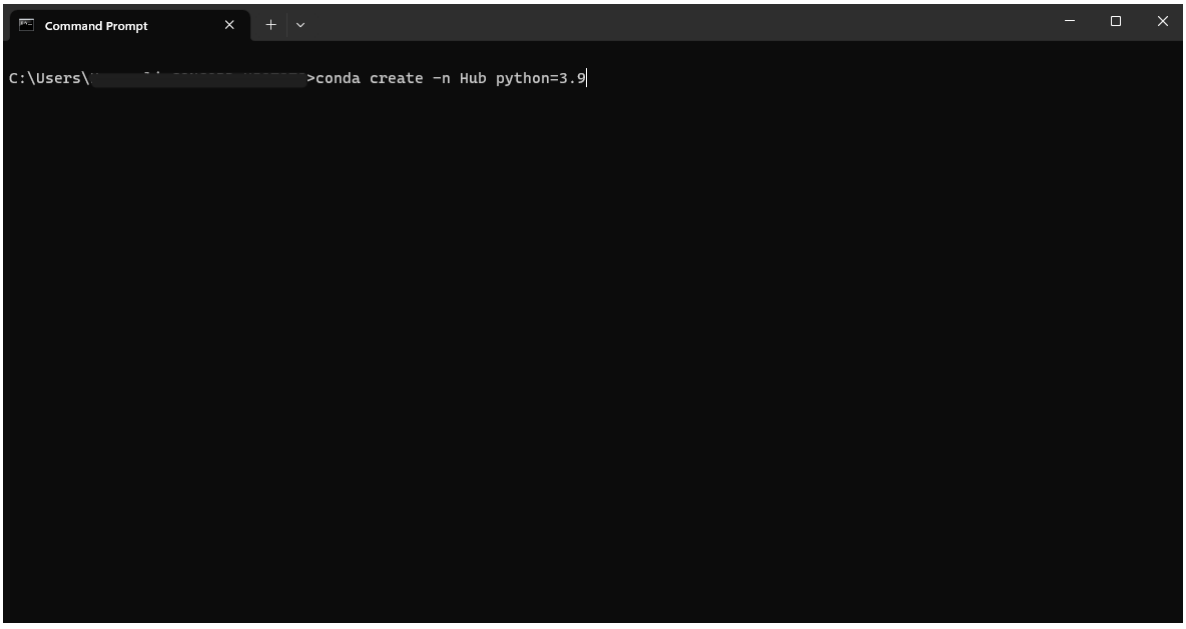

Figure 1: Caption

We need to activate the newly created environment (Hub). Run the following code in the command prompt: **conda activate Hub**. As shown in 2, the result will activate Hub by printing *(Hub)* at the start of the next line. If this didn't work, contact Guillermo (guillermo.gutierrezmorote@concordia.ca)

or Koa (kekoa.wells@concordia.ca) for some help.

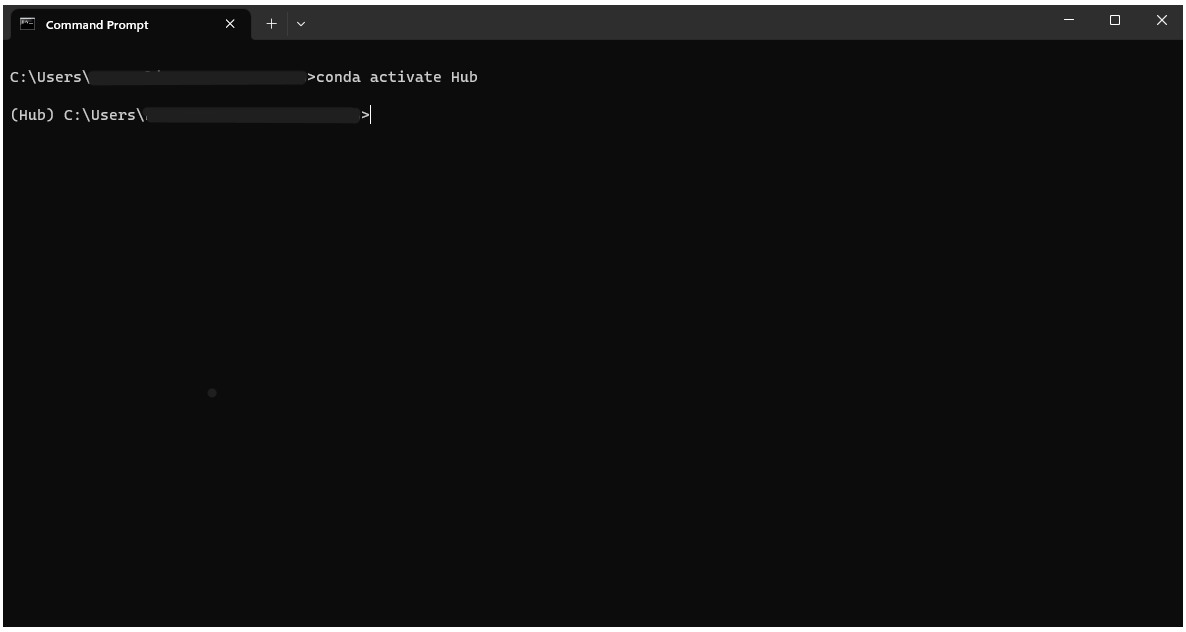

Figure 2: Caption

Having the Hub virtual environment activated, we can install Cerc Hub. To do that, run the following line of codes in the terminal:
pip install -U cerc-hub This command will take a minute and as a result, all the packages needed to work with the Hub will be installed. This command should be executed with no errors. In case of any difficulties. please reach out to Guillermo (guillermo.gutierrezmorote@concordia.ca).

Now, we can create a python file in PyCharm in the Hub environment. First, open PyCharm and select *File > New Project*. Then, in the opened window, specify the new project's location, and as shown in figure 3, from *Previously Configured Interpreter > Interpreter* select **Hub**. Click on Create button. The new project will be opened, and you can verify the Hub environment activation by looking at the bottom right corner, where Hub is printed. If you want to manually set up the environment, refer to the appendix A.

4 Basic geometry import

Within the same python file, we can start creating the first use case of Hub. A city instance could be created using a GML file of buildings. The GML file needs to be downloaded on the local machine, preferably in the same directory as python file. To continue, please download the GML file provided on Cerc gitea. To create the city, firstly, the **Geometry factory** should be imported (figure 4). Hub has several importers that are responsible for importing data to the Hub from multiple sources. Run the code, and it should be executed with no errors.

The downloaded GML file is defined as a variable called *gml_file*. To assign the GML file to the gml_file variable, we need to pass the path of the GML file. The code is shown in figure 4 for the file available in the same directory as python.

Next, a variable, for example, a city, is defined as the name of the city we want to create. The *"Geometry Factory"* class loads different data formats (for example, citygml, obj, gpandas, geojason, etc) and retrieves data from a given file using specific geometric modules. The *"Geometry Factory"* class needs different arguments; for example, the first one, which is obligatory is the file type. In our example, the file type is 'citygml'. Other arguments are file path, year of the construction field and so on. In this example, we need to specify the file type and file path as shown in figure 5 using the code:
city = GeometryFactory('citygml', gml_file).city

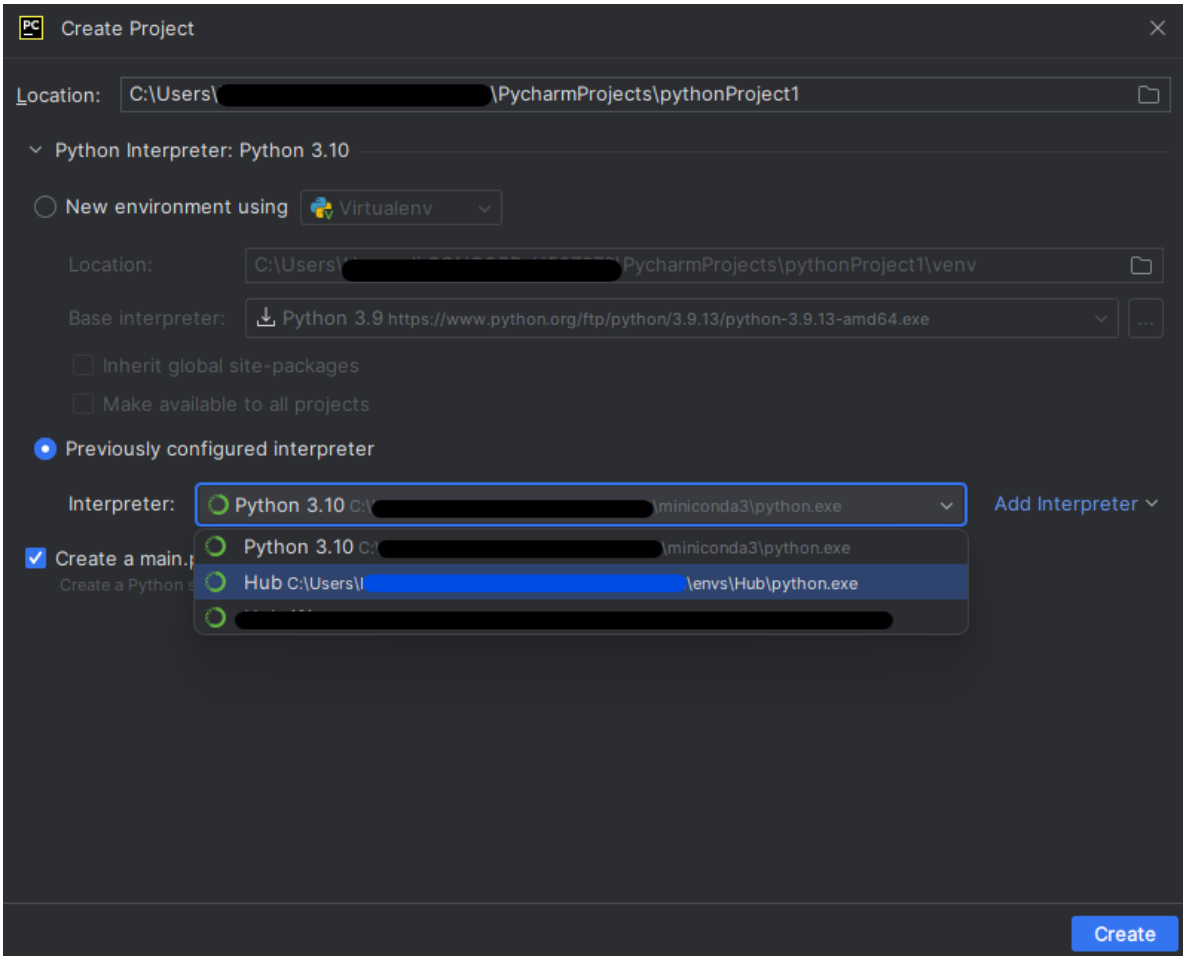

Figure 3: Caption

Running the code will create an instance of the city but won't return anything. By writing a simple for loop (over the buildings in the city) and calling the *"buildings"* function, a variety of information, including construction year, floor area, volume, and other attributes of the buildings, are printed as shown in figure 5. You can open the gml file and verify the printed information.

5 Export back to obj

The imported gml file could be exported to obj format. To do this, firstly, the *export factory* should be imported (similar to what we did for the import factory). Then the following code will generate the obj file and save it in the same directory. 'obj' is the format of the output file, the city is the name of the variable we need to export, and the last part specifies the path.

ExportsFactory('obj', city, './').export()

As you can see in figure 6, the exported obj file has now appeared in the left-hand part (Unknown.obj). You can click on the file and investigate it. It doesn't provide meaningful information; therefore, we can derive information using an online obj viewer. This website provides a simple obj viewer tool. You need to upload the obj file, and it will provide the 3D view of the object.

The *Export factory class* is able to export the given city's geometry to different formats; for example, citygml, stl, obj and sra.

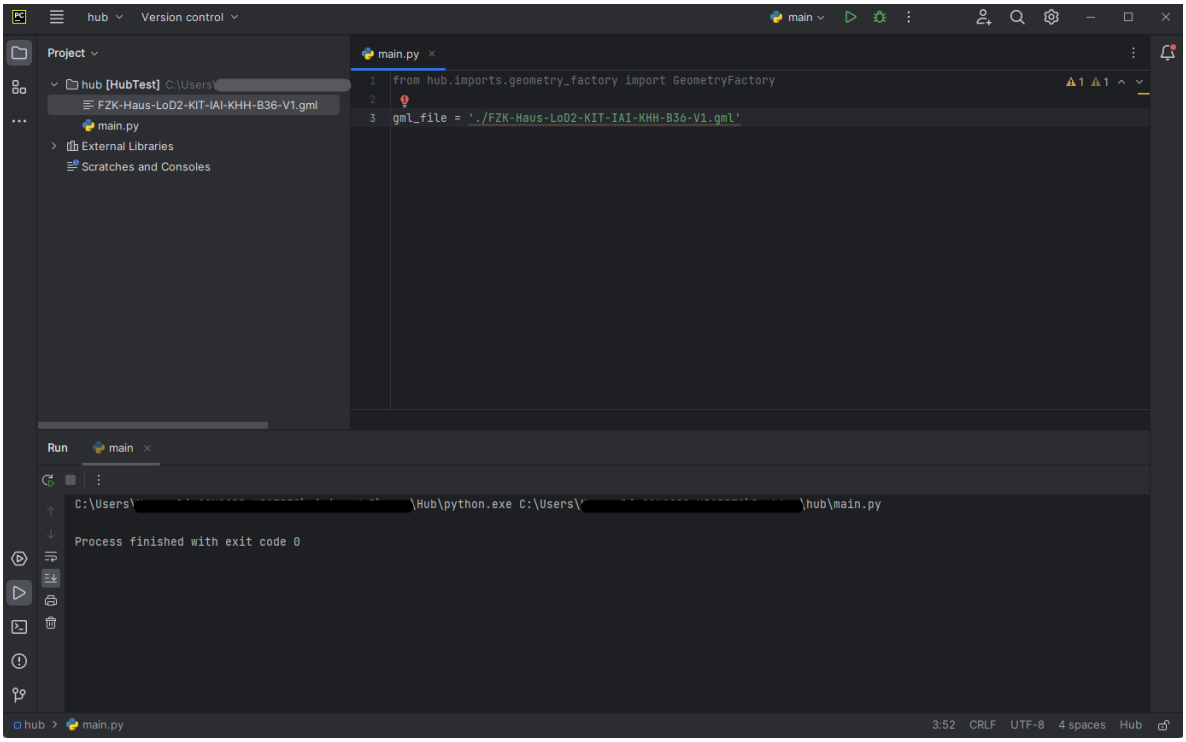

Figure 4: Caption

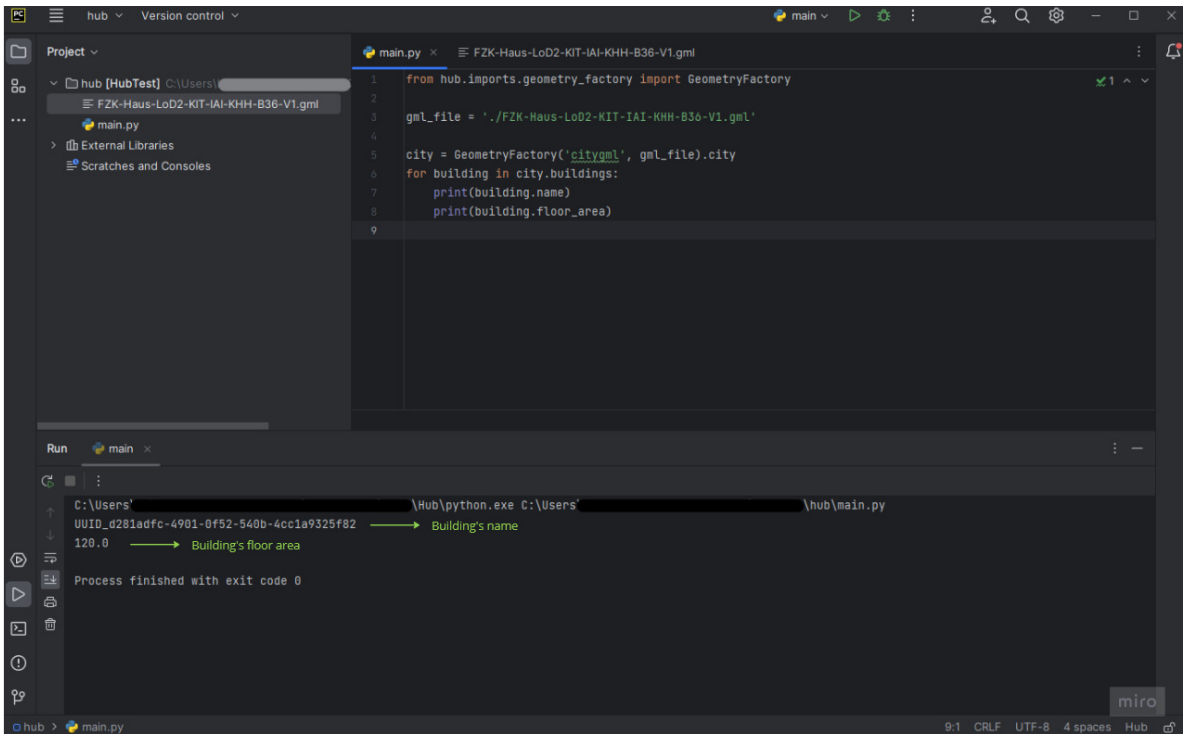

Figure 5: Caption

*

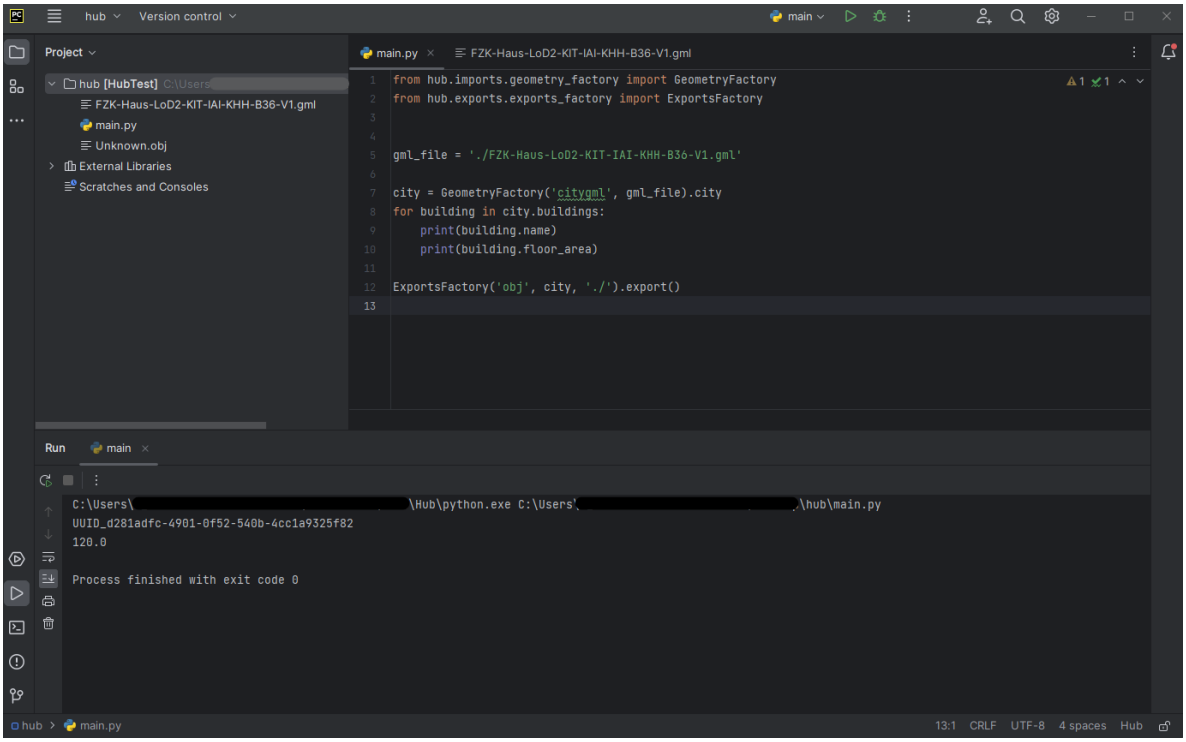

Figure 6: Caption

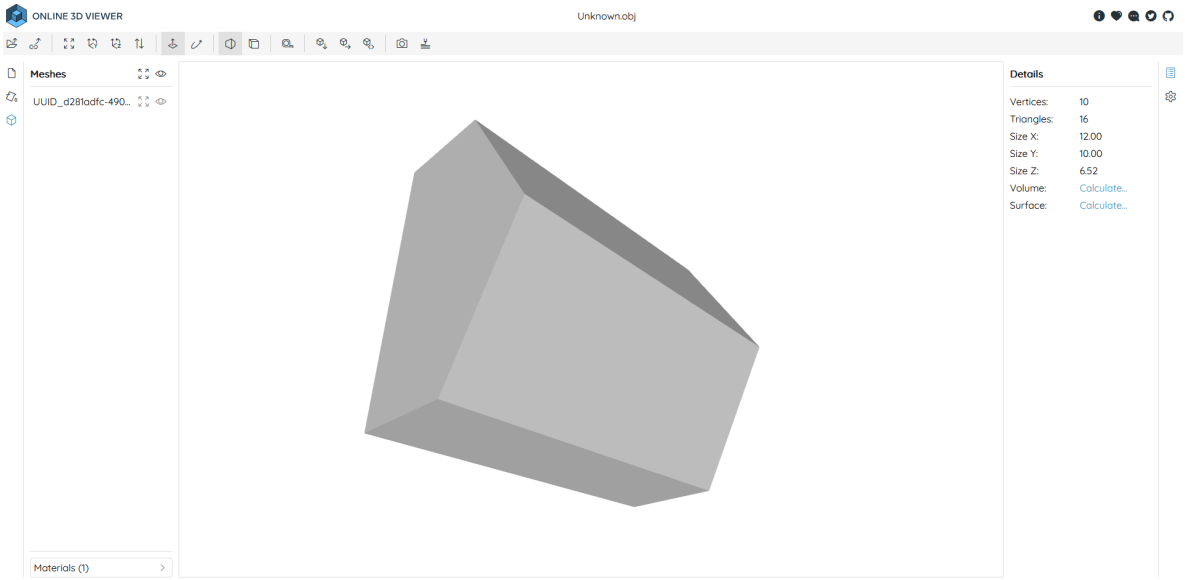

Figure 7: Caption

6 Supported formats

Hub supports various types of data formats, and with exporters, it is able to export the city to different formats. Supported formats and a brief introduction to each of them are provided in this section.

6.1 XML

The Extensible Markup Language (XML) is a simple text-based format for representing structured information: documents, data, configuration, books, transactions, invoices, and much more. It was

derived from an older standard format called SGML (ISO 8879), in order to be more suitable for Web use. XML is one of the most widely-used formats for sharing structured information today: between programs, between people, between computers and people, both locally and across networks. For example, Hub imports construction and material information defined by NREL in XML format.

6.2 JSON

JSON (JavaScript Object Notation) is a lightweight data interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language-independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.
JSON is built on two structures; A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array. And an ordered list of values. In most languages, this is realized as an array, vector, list, or sequence. For example, Hub imports construction and material information defined by NRCAN in JSON format.

6.3 GeoJSON

GeoJSON is an open standard format designed to represent simple geographical features and their non-spatial attributes. It is based on the JSON format. The features include points (addresses and locations), line strings (streets, highways and boundaries), polygons (countries, provinces, tracts of land), and multi-part collections of these types. GeoJSON also includes some additional properties that are specific to geographic features, such as coordinate reference system (CRS) information, which defines the spatial reference system used by the data.

6.4 CityGML

The CityGML standard defines a conceptual model and exchange format for representing, storing, and exchanging virtual 3D city models. It facilitates the integration of urban geodata for a variety of applications for Smart Cities and Urban Digital Twins, including urban and landscape planning; Building Information Modeling (BIM); mobile telecommunication; disaster management; 3D cadastre; tourism; vehicle & pedestrian navigation; autonomous driving and driving assistance; facility management, and; energy, traffic and environmental simulations.
One of the key features of CityGML is its support for multiple levels of detail (LODs), which allows for the representation of urban models at different levels of complexity and scale. For example, a CityGML model may include a simple representation of a building at LOD1, which includes just the basic geometry and location information, or a more detailed representation at LOD2 or LOD3, which includes additional attributes such as building use, materials, and textures.

6.5 OBJ

the OBJ format is a file format used for storing 3D object models. It stores models as a series of vertices, faces, surface normals, and texture coordinates, and is a simple and flexible format widely supported by 3D graphics software packages. However, it does have some limitations, such as the lack of support for animations and larger file sizes compared to other formats.

6.6 Rhino

Rhino format is a proprietary file format used by the Rhinoceros 3D modeling software, which is popular in various industries such as architecture and industrial design. It supports various types of geometry, including points, curves, surfaces, and solids and non-geometric data. Rhino format is compatible with other CAD and 3D modelling software applications, and Rhino also supports other file formats such as DWG, DXF, OBJ, STL, and FBX for exchanging data with other software applications and workflows.

A Configure Python Interpreter

So far, miniconda and Pycharm have been installed, a new environment to work with hub has been created, the cerc hub source code has been cloned, and we can create the first python file.

To activate *"Hub"* virtual environment in Pycharm, follow the steps below:

1. Click on *"File"*, and select *"Settings..."* from the drop-down menu. The Settings window will appear. From the panel on the left, select *"Project: hub>Python Interpreter"* and you will see the following window:

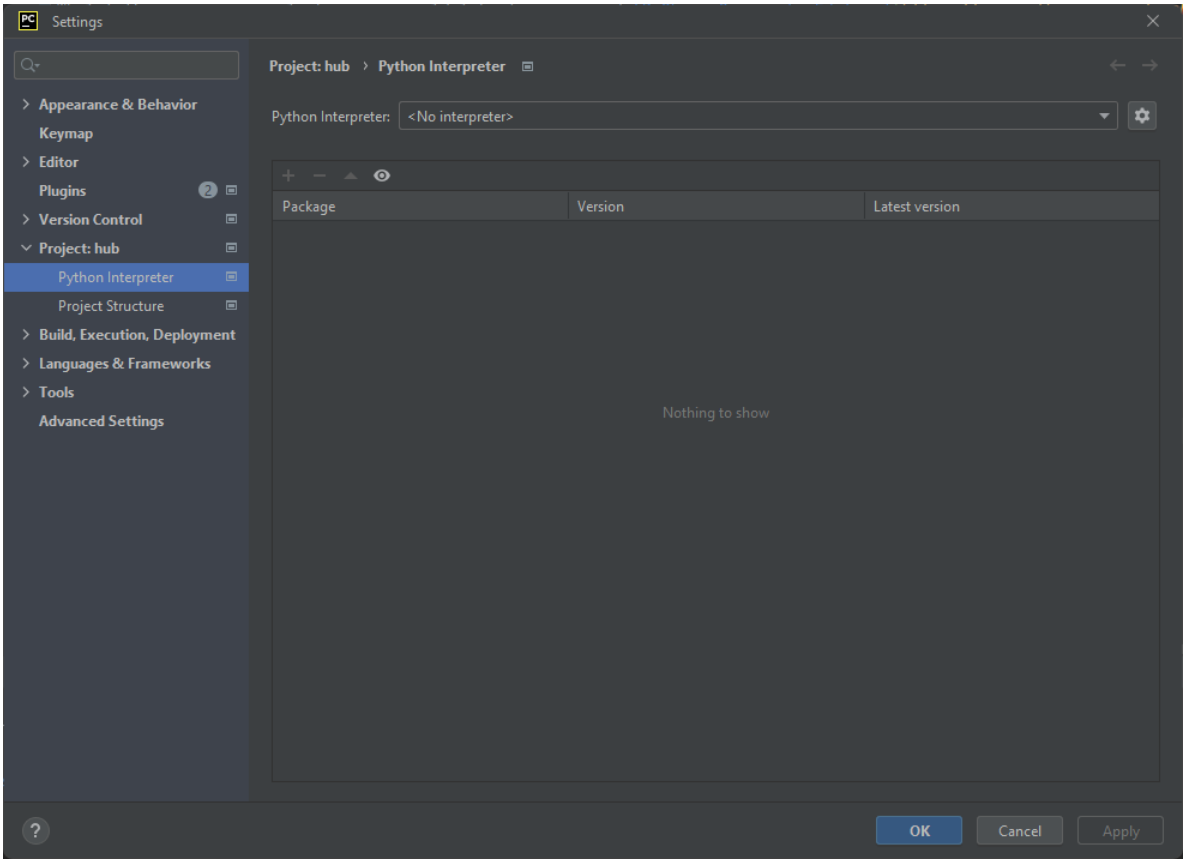

Figure 8: Caption

2. Click on the little gear on the right and click *"Add..."*. In the new window, click on *"Conda Environment"*, select *"New environment"* and the *Python version 3.9*.

You should end with the following window (10)

Click *"ok"* to close it.

5. Go to the Terminal (a tab in the bottom) to finish this process. You should see *_C:\ Users\ Pilar\ PycharmProjects\ hub >* preceded either by *_(base)_* or *_(hub)_*. If non of those are there, be sure that you have Miniconda installed and included in the path. If not, do it, restart Pycharm, and come back to this point.

If the environment you are working in is *_(base)_*, type "conda activate" and the project's name, in this case, **"conda activate hub"**. Click enter, and the environment should change to *_(hub)_*. If this didn't work, contact Guillermo (guillermo.gutierrezmorote@concordia.ca) or Koa (kekoa.wells@concordia.ca) for some help.

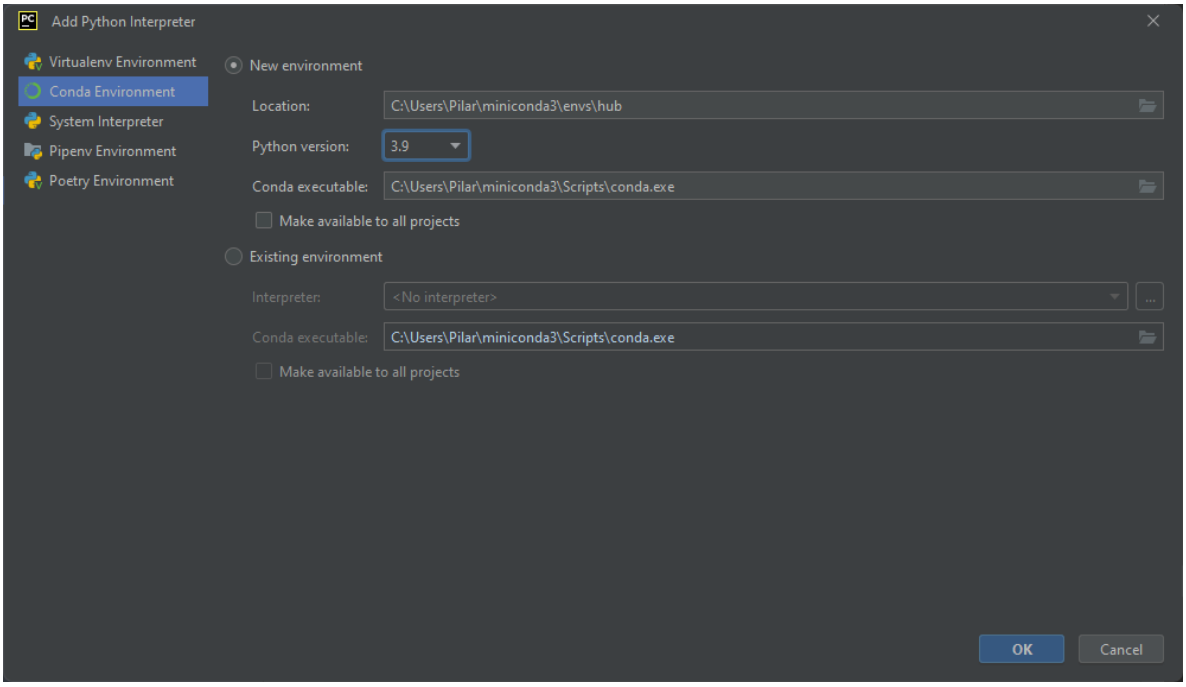

Figure 9: Caption

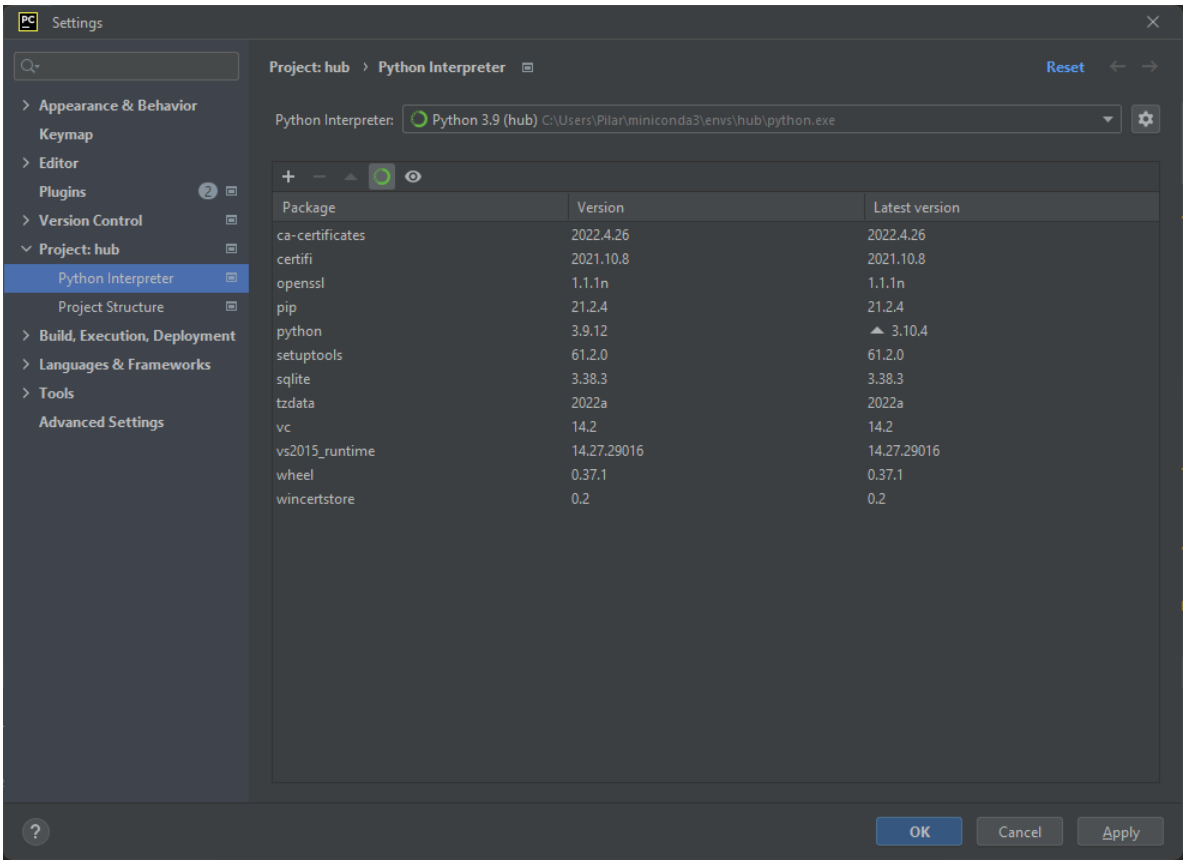

Figure 10: Caption

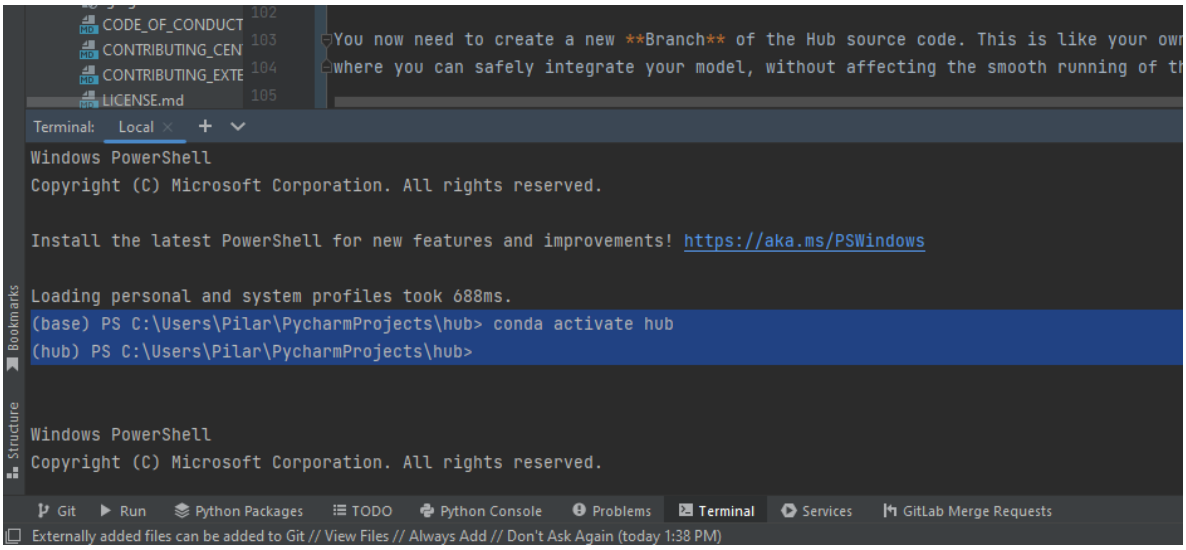

Figure 11: Caption