SQLITE + SMATH GUIDE

(By: Jordan Jevon, 2023)

(V: 2.60)

This tutorial would include:

1. Why SQLite
2. Creating database using python
3. SQLite plugin intro
4. Making a useful method + snippet to automate repetitive input
5. Making an app to help edit database value

1. Why SQLite

This is a rather straight foward ones, but the easiest reason is that most civil engineers would not care about what the heck is a database and would just simply goes to excel or any other method of storing sets of information although knowing a little bit of programming would help out tremendously (I got you there didn't I). From all the relational databases available (MySQL, PostGres, etc), SQLite is by far the easiest to learn. In fact (as of 2023), it has become quite popular in several data storing system

However we not need to care about those things as long as we can take what SQLite has to offer and apply it to our own major (I am taking structural engineering as my major, so the rest of you should bear with the given examples lol). The end goal here is to make a database using SQLite. This database could then be used in excel, python/jupyter, SMath Studio, etc. In this example, I would guide you to build a usable method in SMath studio such as in **Fig 1**, which is the one I used in my group's bridge engineering report.

Author's personal plugin involves several quick prop, see **Fig 1**.

$b_f := WF(WF\_OPTION\_1, "B") = 201\ mm$

Fig 1. Method for usage

E. Snippet

We would use some additional stocked data from one of the author's fault. This has been built since 2020. For now, it would stay in Jordan's personal fault (see **Fig 3**). One would reduced the amount of time looking down a particular set of table such as in **Fig 2**, by implementing either an .xml based data or sql injection.

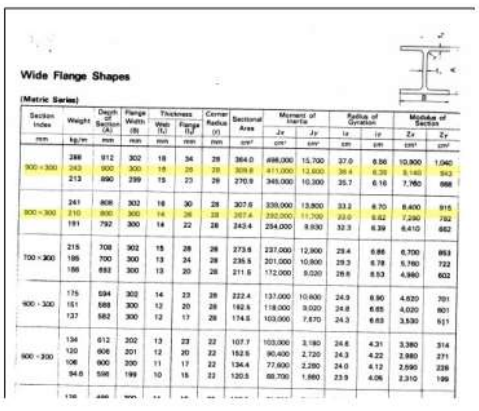


Fig 2. Manual PDF Table

Fig 3. Snippet usage

Fig 1. Snippets Containning Several Databases

The end result would save one much more time since you could modify it to your own needs. I would only layout the general rules and plausible workflow on a very basic level. Therefor, here's what you should do beforehand:

1. Try to learn basic python (understanding of list and dictionary is a must)
2. Try to learn basic SQLite syntax (**CREATE, INSERT, UPDATE, WHERE, DROP** is arguably the most important ones)
3. Understand the 'correct way' to use SMath Studio (Don't use SMath Studio as if it's Microsoft Word). You'd learn the appropriate way of using SMath studio through seeing other people's work (SMath Studio's Forum).

2. Making Database Using Python

In order for you to make a SQLite database, you would need to install python (You just need to watch one or two videos at most since installing python is a really straight foward procedure). You could actually install SQLite on it's own, but it would be quite a waste since python comes with SQLite by default and you would get a glimpse of pythonic feeling when working with SQLite via python. After installing python, I would recommend you to also install VS Code and it's python extensions. By default, python comes with own IDE (IDLE), which is a great IDE (but the lack of auto completion is a hastle), but I would recommend you to try out VS Code since it's the best all rounders

So here's the recap of what you should do:

1. Install Python
2. Install VS Code
3. Install VS Code Python Extension (See **Fig 3**)
4. Install SQLite Studio 3 (This would help if you have no prior experience interacting with terminal)

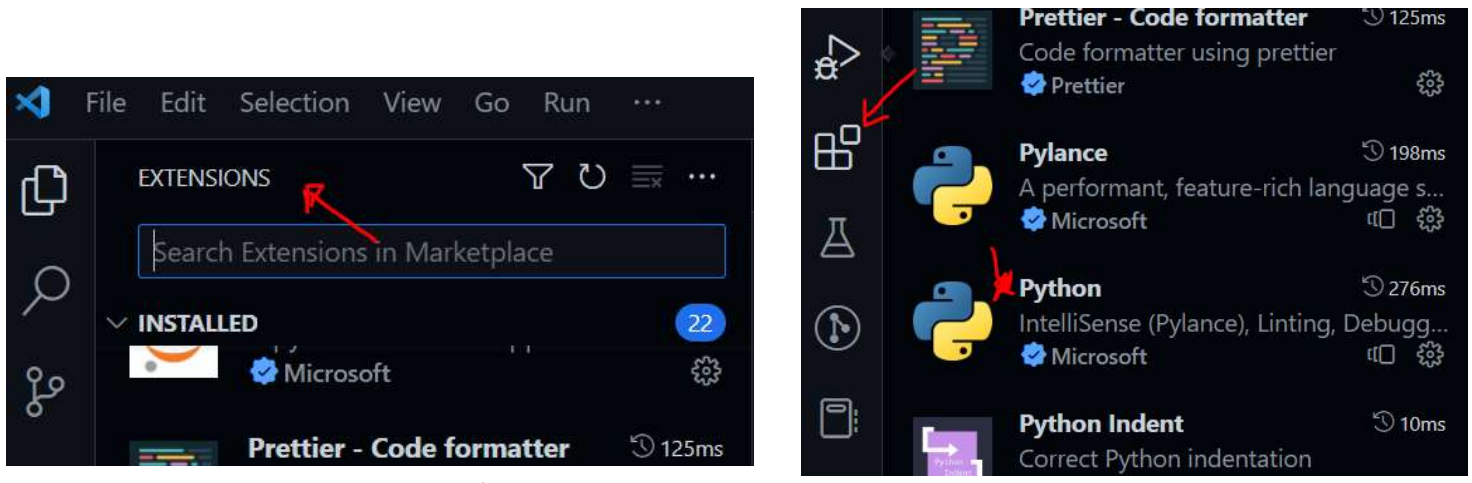


Fig 3. Python extension in VS Code

We would then open a folder and create a python file by adding the .py behind the file's name (See **Fig 4**)

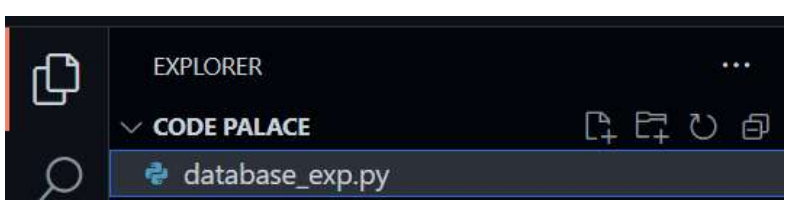


Fig 4. New python file

Note:

The name of the python file doesn't matter since we are only using python to create the database. It is not linked to the database that would be created later on (So feel free to give it any name as long as you put .py at the end). From this point onwards, I recommend making pseducodes, flowcharts or whatever you might need to familiarize yourself with SQLite + Python workflow. (I hate using flowcharts and would rather make pseudocodes instead of flowcharts, that's why you won't see any in my tutorial(s)).

First, import sqlite3 at the beginning of the file (**Fig 5**). The import statement would load a bunch of method that would be used later on. In this case, all sqlite3 commands are loaded when you execute the import statement.

Fig 5. importing sqlite3 module

Create a connection. We create our database using the ***.connect("database_name.db")*** method. Put the desired database name in a string (If you don't know what the heck I am refering to, just look at the Figures from this point onwards; JK: Please also read the statement. I have tried to make it as simple as I could). The important thing here is that you have to put your database name in a "**string**" with **.db** as it's end.

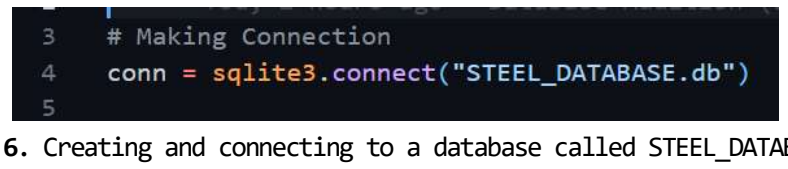


Fig 6. Creating and connecting to a database called STEEL_DATABASE.db

Create a cursor. We would then create a cursor so it would parse the entered command/query to sqlite3. (Remember that we are using python as a sort of factory to run SQLite commands. (See **Fig 7** and **Fig 8**) You would be seeing more of this stuffs when you're creating your own programme.

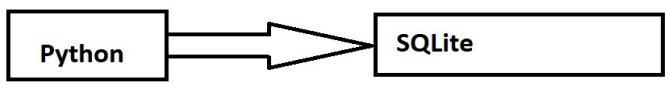


Fig 7. A rather good analogy of what this cursor does

```
# Creating cursor
cursor = conn.cursor()
```

Fig 8. Delcaring cursor to represent connection's cursor method

Commit and close the connection. This is the equivalent thing of saving and closing a word or an excel document. (See **Fig 9** and **Fig 10**). We would keep this command since we would logically speaking, save and close every update that we'd make.

Fig 9. Commiting changes to the database

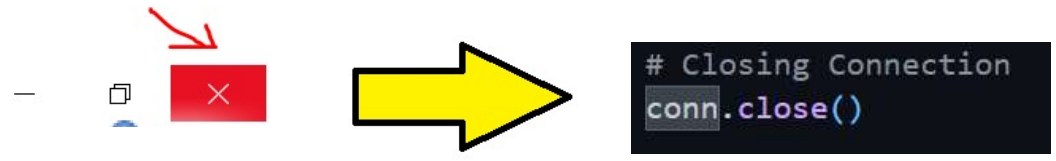


Fig 10. Closing the connection

You would then run the python file by pressing **Ctrl + F5**. (Alternatively, you could also install code runner). The terminal would state the created full path if you succeed in doing so. (See **Fig 11** for all statement declared so far and **Fig 12** for the terminal message). Keep in mind that we would run python by using **Ctrl + F5**. Since we don't need to compile the code as in C/CPP, you are prone to various errors. You could use ChatGPT to help you find you find the errors in your code (that's how I personally use ChatGPT).

```python
import sqlite3

# Making Connection
conn = sqlite3.connect("DEMO_DATABASE.db")

# Creating cursor
cursor = conn.cursor()

# Commit Database
conn.commit()

# Closing Connection
conn.close()
```

Fig 11. Full Code

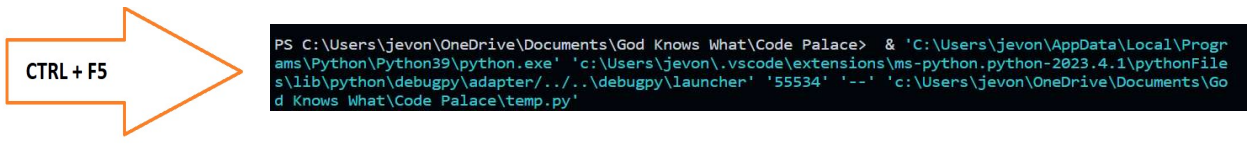


Fig 12. Terminal Message Indicating Nothing went Wrong

You would see the created database in the same folder as the ones where the python code is ran (See **Fig 13**).This **.db** file is consisted of binaries or sometype of format that can't be open using a "normal" means. We would only see a blank file since we haven't put anything (yet).

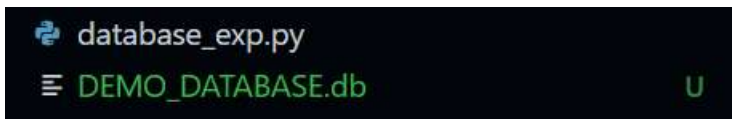


Fig 13. Created database

In this example, I would put the data in **Fig 14** and **Fig 15** into the created database. (You could try crazier stuff depending on your specifics needs, as the title implies, I am assuming that you are a beginner). In this example, we would put steel profile in the **DEMO_DATABASE.db**. The table in **Fig 14** and **Fig 15** is taken from Mr.Dewobroto's book (the only Indonesian based text book in my collections).

Tabel 6.15 Parameter profil *I* – Beam (F_y 240 MPa)
Sumber : PT. Krakatau Wajatama (sesuai Standar JIS G3192-2005)

Notasi	$d \times b_f \times t_w \times t_f$	Berat	Z_x	ϕM_p	ϕM_r	BF	L_p	L_r	I_x	ϕV_n
d x berat	mm	kg/m	cm^3	kN-m	kN-m	kN	m	m	cm^4	kN
I100x8	100x50x4.5x6.8	8.34	40	9	5	1.4	0.6	3.0	172	64
I120x11	120x58x5.1x7.7	11.1	64	14	8	2.1	0.7	3.3	331	88
I140x14	140x66x5.7x8.6	14.3	96	21	12	2.9	0.8	3.6	579	114
I160x18	160x74x6.3x9.5	17.9	137	30	18	3.9	0.9	3.9	944	146
I160x18	180x82x6.9x10.4	21.9	188	41	25	5.0	0.9	4.2	1,460	179
I200x26	200x90x7.5x11.3	26.2	251	54	33	6.2	1.0	4.5	2,162	216

Fig 14. I-Beam Profile (Dewobroto, 2016)

Tabel 6.12 Parameter *Wide Flange* (F_y 240 MPa)
Sumber : PT. Krakatau Wajatama (sesuai Standar JIS G3192-2005)

Notasi		$d \times b_f \times t_w \times t_f$	Berat	Z_x	ϕM_p	ϕM_r	BF	L_p	L_r	I_x	ϕV_n
d	kg/m	mm	kg/m	cm^3	kN-m	kN-m	kN	m	m	cm^4	kN
W100	9	100x50x5x7	9.3	42	9	5	1.4	.6	3.1	178	72
W125	13	125x60x6x8	13.1	74	16	10	2.3	.7	3.5	394	108
W150	14	150x75x5x7	14.0	98	21	13	3.5	.9	3.2	642	108
	21	148x100x6x9	21.0	150	32	20	3.	1.2	5.4	981	128
W175	18	175x90x5x8	18.0	152	33	20	4.6	1.1	3.8	1,172	126
W200	18	198x99x4.5x7	18.0	170	37	23	5.7	1.1	3.6	1,498	128
	21	200x100x5.5x8	21.0	200	43	27	6.1	1.1	3.9	1,761	158
	30	194x150x6x9	29.9	296	64	40	5.1	1.9	6.5	2,585	168
W250	25	248x124x5x8	25.1	305	66	41	8.6	1.4	4.3	3,378	179
	29	250x125x6x9	29.0	352	76	47	9.2	1.4	4.6	3,893	216

Fig 15. W-Beam Profile (Dewobroto, 2016)

We would begin by Creating Three Seperate tables. Two for the profile data and one for the unit. The reason for this is quite simple. SMath Studio works by combining Constant (Number) and Units. You could find this .xml sourcefile in your SMath installation folder (See **Fig 16.**)

Fig 16. SMath input components

We would begin by creating the first table, which is the I profile table. Note that the order of you making the table would not affect anything since relational database are built upon data relation (unlike the fix placement found in excel etc.). In this

case, we would not be using or assigning certain columns to be the primary key since this case doesn't call for it. We would just make a table with all the parameters in **Fig 14.** (See **Fig 17**)

```
# Creating Table 1: I Profile
cursor.execute("""
                CREATE TABLE i_data(
                    section_name TEXT,
                    weight REAL,
                    Z_x REAL,
                    phi_Mp REAL,
                    phi_Mr REAL,
                    BF REAL,
                    Lp REAL,
                    Lr REAL,
                    I_x REAL,
                    phi_Vn REAL
                )
                """)
```

Fig 17. Creating i_data table of Fig 14.

We would then ran this code using **CTRL** + **F5**. We would then see some blue-ish messages in the terminal (**Fig 18**)

```
ams\Python\Python39\python.exe' 'c:\Users\jevon\.vscode\extensions\ms-python.python-2023.4.1\pythonFile
s\lib\python\debugpy\adapter/../..\debugpy\launcher' '55534' '--' 'c:\Users\jevon\OneDrive\Documents\Go
d Knows What\Code Palace\temp.py'
PS C:\Users\jevon\OneDrive\Documents\God Knows What\Code Palace>  c:; cd 'c:\Users\jevon\OneDrive\Docum
ents\God Knows What\Code Palace'; & 'C:\Users\jevon\AppData\Local\Programs\Python\Python39\python.exe'
'c:\Users\jevon\.vscode\extensions\ms-python.python-2023.4.1\pythonFiles\lib\python\debugpy\adapter/../
..\debugpy\launcher' '55601' '--' 'c:\Users\jevon\OneDrive\Documents\God Knows What\Code Palace\temp.py
'
```

Fig 18. Terminal message upon running the code

Note:
You should comment out everything related to the query once you ran the code. If you try to run the same code twice, the terminal would pop up an error message, stating that you have previously do a certain things, etc..

We have now succeed in creating a table. However, don't rush in your excitement just yet since you can't open the database rightaway. If you try to open the database (now), you would see something similar to **Fig 19.**

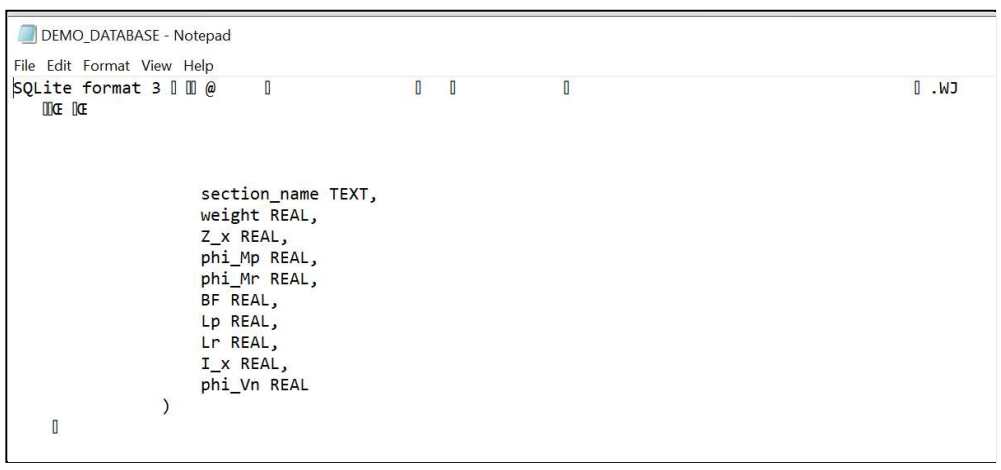


Fig 19. Openning the created database in notepad

In order to view the created items, we need some external help. There are many viable alternatives but I only use two of them:
1. SQLite VS Code extension (**Fig 20**).
2. SQLite Studio (3.3.3) (**Fig 21**).

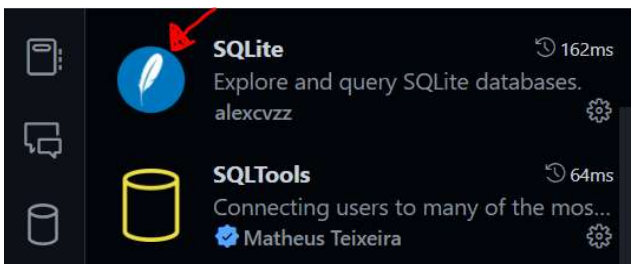


Fig 20. SQLite extension in vscode

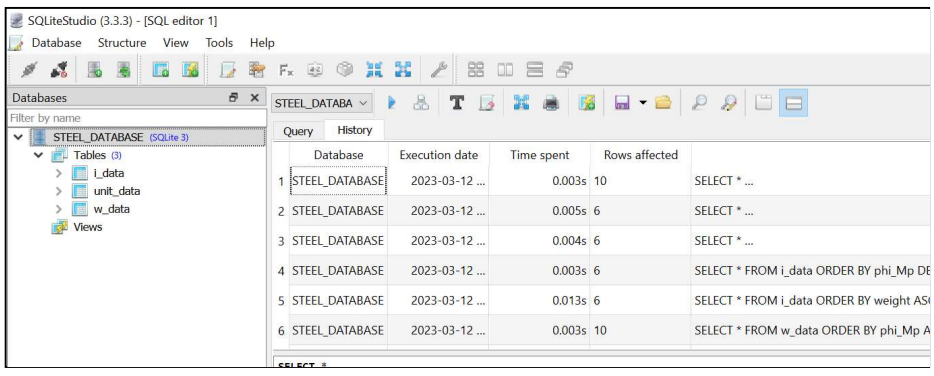


Fig 21. SQLite studio

Feel free to use any other alternatives (SQLite browser, beestudio, etc). If one has already installed the VS Code extension, you could see the created table by right clicking the database and select the open database option (See **Fig 22**). If one has already installed SQLite Studio, you could extract the file and search for the .exe in the extracted folder. Choose a language and you're ready to go. Open the database by clicking **Database** -> **add a database** (**Ctrl** + **O**). You would need to browse the database location (See **Fig 23**). The usage of SQLite studio is instinctive by nature, so you should not have any problem with that (If you do, feel free to open up youtube).

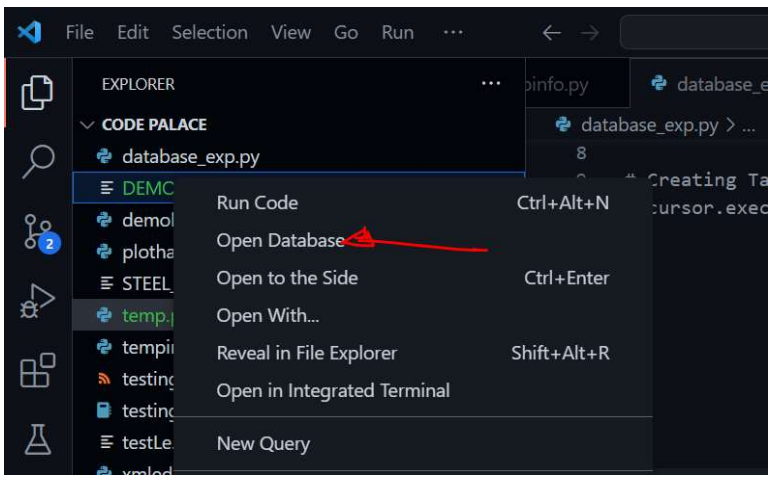


Fig 22. Openning a database in vscode

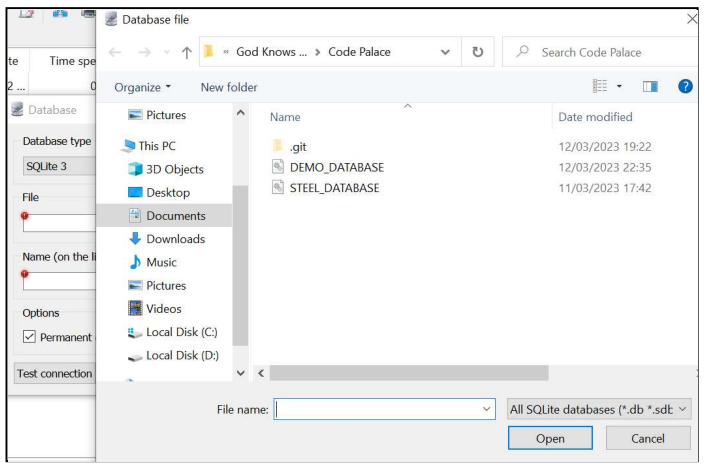


Fig 23. Openning a database in SQLite Studio

Upon openning the database, you would see **Fig 24** and **Fig 25.**

Fig 24. Created table in SQLite Explorer

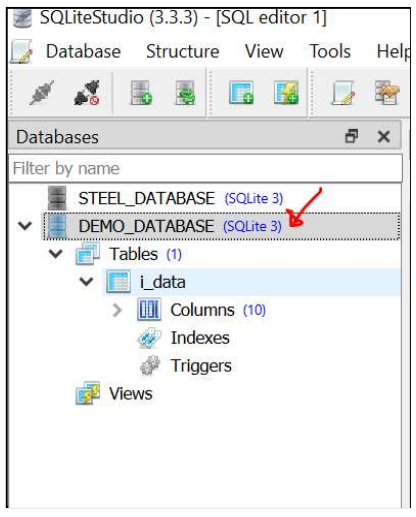


Fig 25. Created table structures and props in SQLite Studio

You could then check wheter this the i_data table that were created. On VS Code, you would likely see **Fig 26** and on SQlite Studio, you would see the displayed result in **Fig 27.**

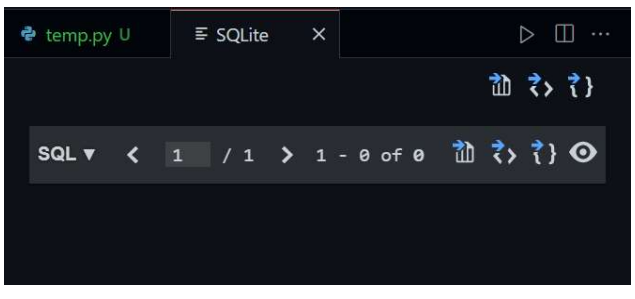


Fig 26. Empty table view

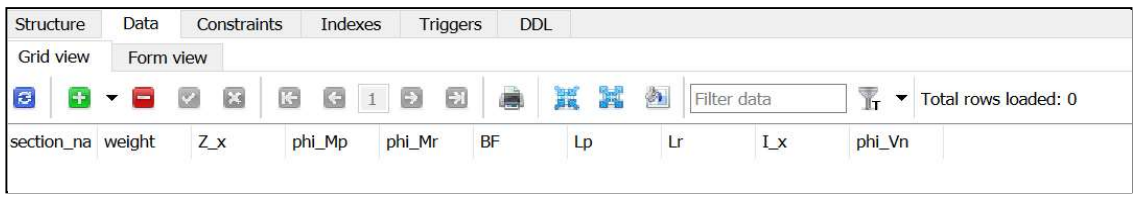


Fig 27. Empty table view

So far, we have sucessfully created a table. What we need to do next is to either create another table or start filling the table with some data. I would choose the former since I work faster by making a bunch of query (whether I ended up using them is another set of problem). Note that you could actually fill this table first rather then following my foot steps since the only rule that applies in programming is that it works. I would continue creating unit table using the syntax in **Fig 29.** I would also comment out the previously declared syntax in **Fig 17** (See **Fig 28**)

```
# Creating Table 1: I Profile
# cursor.execute("""
#                CREATE TABLE i_data(
#                    section_name TEXT,
#                    weight REAL,
#                    Z_x REAL,
#                    phi_Mp REAL,
#                    phi_Mr REAL,
#                    BF REAL,
#                    Lp REAL,
#                    Lr REAL,
#                    I_x REAL,
#                    phi_Vn REAL
#                )
#                """)
```

Fig 28. Commented query

```
21  #                    phi_Vn REAL
22  #                )
23  #                """)
24
25  # Creating Unit Table
26  cursor.execute("""
27                 CREATE TABLE unit_data(
28                     prop TEXT,
29                     unit TEXT
30                 )
31                 """)
```

Fig 29. Newly executed query

CTRL + F5

We would create the Unit Table using two parameters, the prop and unit. The Prop would represent the steel properties declared on the I profile table while the unit would represent the properties unit found in the steel table (**Fig 14).** We would then execute the whole program using **ctrl** + **f5** (by this point you should already know what to do and I wold not be posting more terminal messages since it would be a waste of space). We would then end up with two empty table (See **Fig 30** and **Fig 31**).

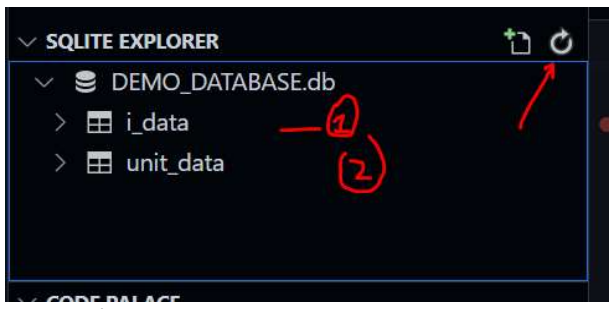


Fig 30. Created table view in VSCode

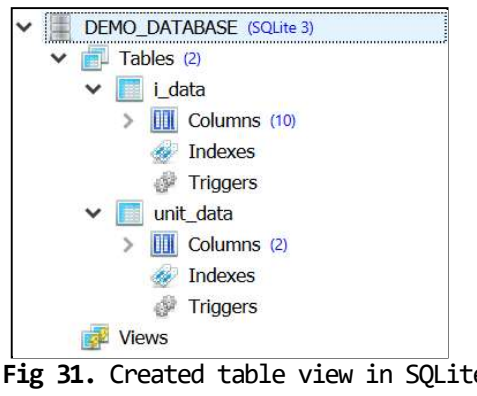


Fig 31. Created table view in SQLite Studio

We would also comment out the cursor after unit_data table is created. We would now begin to fill the the table. There are many ways to fill the table, you could loop them, execute them line per line or make a list then commiting them. I would choose the later since It's is the right time to use ***executemany()*** method instead of single ***execute()*** method. We would first declare a list. We would then put tupples that contain a single row worths of data (See **Fig 32** and **Fig 33**) -> Basically use put this structure:

[(content,content),(content,content),(content,content),(content,content),(content,content)]

```
# I profile data
i_profile_list = [
    ("I 100x50x4.5x6.8", 8.34, 40, 9, 5, 1.4, 0.6, 3.0, 172, 64),
    ("I 120x58x5.1x7.7", 11.1, 64, 14, 8, 2.1, 0.7, 3.3, 331, 88),
    ("I 140x66x5.7x8.6", 14.3, 96, 21, 12, 2.9, 0.8, 3.6, 579, 114),
    ("I 160x74x6.3x9.5", 17.9, 137, 30, 18, 3.9, 0.9, 3.9, 944, 146),
    ("I 180x82x6.9x10.4", 21.9, 188, 41, 25, 5.0, 0.9, 4.2, 1460, 179),
    ("I 200x90x7.5x11.3", 26.2, 251, 54, 33, 6.2, 1.0, 4.5, 2162, 216)
]
```

Fig 32. i_data table's content.

```
# Unit data
unit_list = [
    ("weight", "'kgf/'m"),
    ("Z_x", "'cm^3"),
    ("phi_Mp", "'kN*'m"),
    ("phi_Mr", "'kN*'m"),
    ("BF", "'kN"),
    ("Lp", "'m"),
    ("Lr", "'m"),
    ("I_x", "'cm^4"),
    ("phi_Vn", "'kN")
]
```

Fig 33. unit_list table's content.

We would then insert this listed tuple to the table using execute many method. Note that the number of ***?*** presented on the docstring should be the same as the number of columns on the newly created list. Note that query can be copied and pasted. Comment out all the syntax after entering **ctrl** + **f5** once.

```
# query to insert listed tuples to i_data table
# execute the code one at a time
# Inserting list to table
# Note: You can copy paste the code if you want to
cursor.executemany("""
                   INSERT INTO i_data VALUES (?,?,?,?,?,?,?,?,?,?)
                   """, i_profile_list)

cursor.executemany("""
                   INSERT INTO unit_data VALUES (?,?)
                   """, unit_list)
```

After all of that has been done, you could then view the populated tables (See **Fig 34** and **Fig 35**).

SQL ▾ 1 / 1 1 - 6 of 6

section_name	weight	Z_x	phi_Mp	phi_Mr	BF	Lp	Lr
I 100x50x4.5x6.8	8.34	40.0	9.0	5.0	1.4	0.6	3.0
I 120x58x5.1x7.7	11.1	64.0	14.0	8.0	2.1	0.7	3.3
I 140x66x5.7x8.6	14.3	96.0	21.0	12.0	2.9	0.8	3.6
I 160x74x6.3x9.5	17.9	137.0	30.0	18.0	3.9	0.9	3.9
I 180x82x6.9x10.4	21.9	188.0	41.0	25.0	5.0	0.9	4.2
I 200x90x7.5x11.3	26.2	251.0	54.0	33.0	6.2	1.0	4.5

Fig 34. i_data table's content view in VSCode

Structure | Data | Constraints | Indexes | Triggers | DDL

Grid view | Form view

Filter data — Total rows loaded: 6

	section_name	weight	Z_x	phi_Mp	phi_Mr	BF	Lp	Lr	I_x	phi_Vn
1	I 100x50x4.5x6.8	8.34	40	9	5	1.4	0.6	3	172	64
2	I 120x58x5.1x7.7	11.1	64	14	8	2.1	0.7	3.3	331	88
3	I 140x66x5.7x8.6	14.3	96	21	12	2.9	0.8	3.6	579	114
4	I 160x74x6.3x9.5	17.9	137	30	18	3.9	0.9	3.9	944	146
5	I 180x82x6.9x10.4	21.9	188	41	25	5	0.9	4.2	1460	179
6	I 200x90x7.5x11.3	26.2	251	54	33	6.2	1	4.5	2162	216

Fig 35. i_data table's content view in SQLite Studio

!Warning:

Note that we would not be assininning any primary keys nor other attributes to any parameter. We would only created a raw file and do the rest of the work in SMath Studio. You could find out more about SQLite on their website (I actually don't recommend beginners to actually jump there, try w3 school's tutorial instead).

https://www.sqlite.org/docs.html — Official SQLite documentaion

https://www.w3schools.blog/sqlite-tutorial — w3school's tutorial

Here's the complete code:

```
import sqlite3

# Making Connection
conn = sqlite3.connect("DEMO_DATABASE.db")

# Creating cursor
cursor = conn.cursor()

# Creating Table 1: I Profile
# cursor.execute("""
#                CREATE TABLE i_data(
#                    section_name TEXT,
#                    weight REAL,
#                    Z_x REAL,
#                    phi_Mp REAL,
#                    phi_Mr REAL,
#                    BF REAL,
#                    Lp REAL,
#                    Lr REAL,
#                    I_x REAL,
#                    phi_Vn REAL
#                )
#                """)

# Creating Unit Table
# cursor.execute("""
#                CREATE TABLE unit_data(
#                    prop TEXT,
#                    unit TEXT
#                )
#                """)

# I profile data
i_profile_list = [
    ("I 100x50x4.5x6.8", 8.34, 40, 9, 5, 1.4, 0.6, 3.0, 172, 64),
    ("I 120x58x5.1x7.7", 11.1, 64, 14, 8, 2.1, 0.7, 3.3, 331, 88),
    ("I 140x66x5.7x8.6", 14.3, 96, 21, 12, 2.9, 0.8, 3.6, 579, 114),
    ("I 160x74x6.3x9.5", 17.9, 137, 30, 18, 3.9, 0.9, 3.9, 944, 146),
    ("I 180x82x6.9x10.4", 21.9, 188, 41, 25, 5.0, 0.9, 4.2, 1460, 179),
    ("I 200x90x7.5x11.3", 26.2, 251, 54, 33, 6.2, 1.0, 4.5, 2162, 216)
]

# Unit data
unit_list = [
    ("weight", "'kgf/'m"),
    ("Z_x", "'cm^3"),
    ("phi_Mp", "'kN*'m"),
    ("phi_Mr", "'kN*'m"),
    ("BF", "'kN"),
    ("Lp", "'m"),
    ("Lr", "'m"),
    ("I_x", "'cm^4"),
    ("phi_Vn", "'kN")
]

# Inserting list to table
# cursor.executemany("""
#                    INSERT INTO i_data VALUES (?,?,?,?,?,?,?,?,?,?)
#                    """, i_profile_list)

# cursor.executemany("""
#                    INSERT INTO unit_data VALUES (?,?)
#                    """, unit_list)

# Commit Database
conn.commit()

# Closing Connection
conn.close()
```

```
Note:
You can copy and paste
this in your prefered IDE
```

3. Entering Database Query in SMath Studio

You could actually enter a sqlite query in SMath Studio with syntax that didn't involve type='' (SMath doesn't allow for '=' in its string method). But before writing anything down, you should first install the sqlite plugins (See **Fig 36** and **Fig 37**).

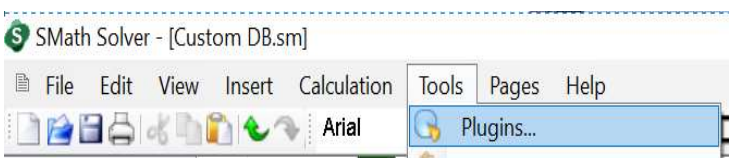


Fig 36. Plugins Tab location

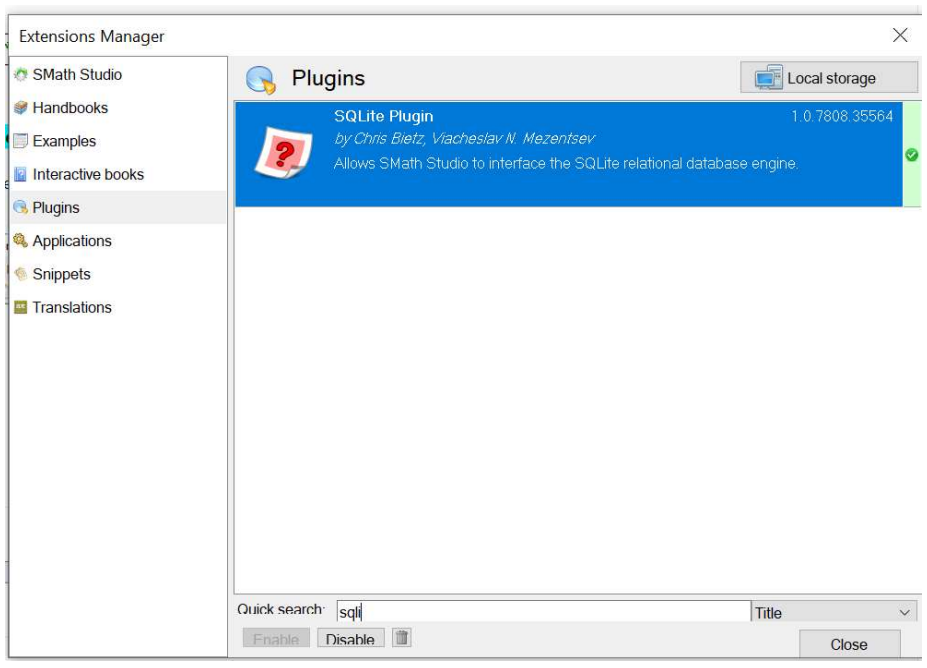


Fig 37. SQLite plugins installation

There are many SMath Studio versions, you could choose any version you like as long as it's above 0.97. You could find your SMath studio version by typing appVersion(int).

```
appVersion(3)="1.0.8238"
```

We would begin by specifying the database location. You could use a pathpicker, but I would stick to a manually obtainned path for now. You would first declare a variable containning the complete path of a database.

```
Database Path
db_dir:="C:\Users\jevon\OneDrive\Documents\God Knows What\Code Palace\STEEL_DATABASE.db"
```

The database used in this documentation contains a tabulated version of **Fig 15** and **Fig 16** (Both i_profile and w_profile has been inputted). The way that we would do things is described by **Fig 38.**

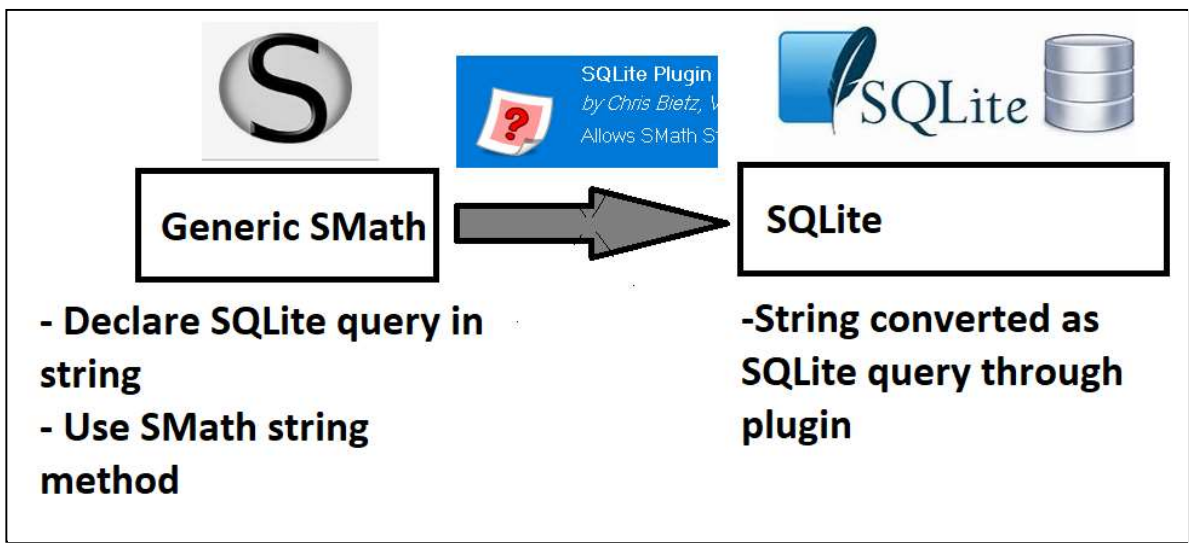
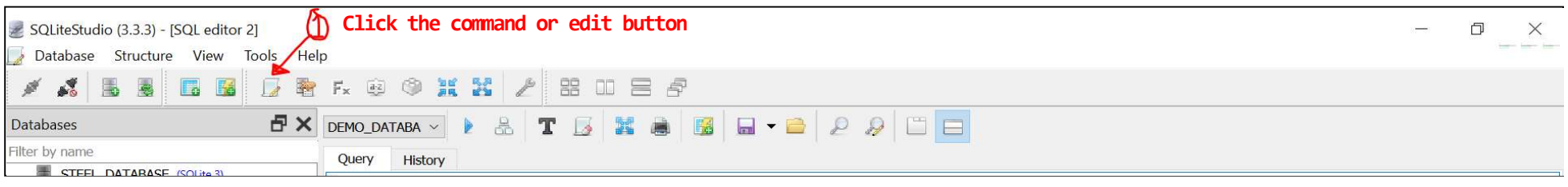


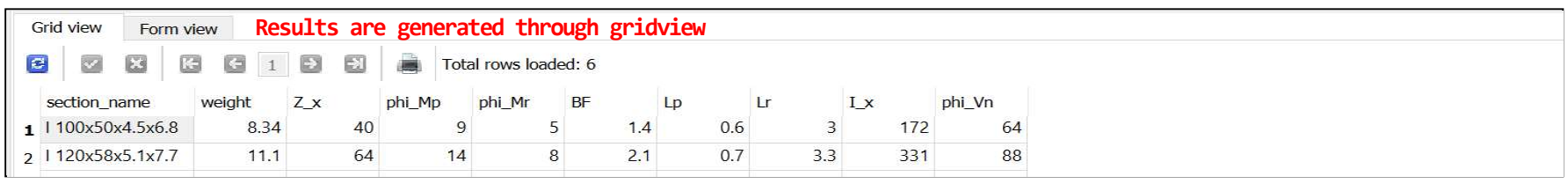
Fig 38. How SQLite plugins work and how we would use those work flow

We would first do several things so you could get the general idea of what we're trying to do. We would first select all the data from i_table. we would see a side by side comparison using either vscode/sqlite studio syntax with SMath SQLite's plugin. To do this, we would enter the following syntax (in either vscode sqlite query or sqlite studio query):

```
SELECT * FROM i_data;
```

Here's what you would do in sqlite studio:





We would do the same thing using SMATH SQLite plugins, we would first declare a string. But instead of declaring a regular string, I would make method that concatenates the **SELECT * FROM** with a (***table name***). This method would be more flexible then typing it out in either vscode's or sqlite studio's query terminal. SMath studio concat works by entering several parameters seperated by comma.
(By default, SMath Studio would use **;** and **,** to represent **,** and **.** You could stick to that convention or change both the comma and argument seperator in **tools** -> **option**.

Here's what concat does:

```
concat("Jordan Jevon is ", num2str(22), " years old") = "Jordan Jevon is 22 years old"
```

We would use a similar concept to extract the i_table data, we would first concat **SELECT * FROM** with a table argument.

```
GET_ALL(table) := concat("SELECT * FROM ", table)
```

We would then pass two things in **SQLiteQuery(*1*,*2*)** method.
1. Database Directory
2. Query (Through SMath Studio string)
It is recommended that you put the line infront of every query usage since some convertion would be converted to num (which would cause an error, you only need to use line for ones that requires both constant and unit evaluation).

```
I_All := SQLiteQuery(db_dir, GET_ALL("i_data"))
```

```
W_All := SQLiteQuery(db_dir, GET_ALL("w_data"))
```

By Default, it would return the requested result in the form of a 2d array (or matrix as it is described in SMath Studio).

i_data content in SMath's matrix form

$$I\_All = \begin{bmatrix} \text{"I 100x50x4.5x6.8"} & 8.34 & 40 & 9 & 5 & 1.4 & 0.6 & 3 & 172 & 64 \\ \text{"I 120x58x5.1x7.7"} & 11.1 & 64 & 14 & 8 & 2.1 & 0.7 & 3.3 & 331 & 88 \\ \text{"I 140x66x5.7x8.6"} & 14.3 & 96 & 21 & 12 & 2.9 & 0.8 & 3.6 & 579 & 114 \\ \text{"I 160x74x6.3x9.5"} & 17.9 & 137 & 30 & 18 & 3.9 & 0.9 & 3.9 & 944 & 146 \\ \text{"I 180x82x6.9x10.4"} & 21.9 & 188 & 41 & 25 & 5 & 0.9 & 4.2 & 1460 & 179 \\ \text{"I 200x90x7.5x11.3"} & 26.2 & 251 & 54 & 33 & 6.2 & 1 & 4.5 & 2162 & 216 \end{bmatrix}$$

w_data content in SMath's matrix form

$$W\_All = \begin{bmatrix} \text{"W 100x50x5x7"} & 9.3 & 42 & 9 & 5 & 1.4 & 0.6 & 3.1 & 178 & 72 \\ \text{"W 125x60x6x8"} & 13.1 & 74 & 16 & 10 & 2.3 & 0.7 & 3.5 & 394 & 108 \\ \text{"W 150x75x5x7"} & 14 & 98 & 21 & 13 & 3.5 & 0.9 & 3.2 & 642 & 108 \\ \text{"W 148x100x6x9"} & 21 & 150 & 32 & 20 & 3 & 1.2 & 5.4 & 981 & 128 \\ \text{"W 175x90x5x8"} & 18 & 152 & 33 & 20 & 4.6 & 1.1 & 3.8 & 1172 & 126 \\ \text{"W 198x99x4.5x7"} & 18 & 170 & 37 & 23 & 5.7 & 1.1 & 3.6 & 1498 & 128 \\ \text{"W 200x100x5.5x8"} & 21 & 200 & 43 & 27 & 6.1 & 1.1 & 3.9 & 1761 & 158 \\ \text{"W 194x150x6x9"} & 29.9 & 296 & 64 & 40 & 5.1 & 1.9 & 6.5 & 2585 & 168 \\ \text{"W 248x124x5x8"} & 25.1 & 305 & 66 & 41 & 8.6 & 1.4 & 4.3 & 3378 & 179 \\ \text{"W 250x125x6x9"} & 29 & 352 & 76 & 47 & 9.2 & 1.4 & 4.6 & 3893 & 216 \end{bmatrix}$$

You could put it in a table by first downloading the table plugins, then declared the headings. (I have a separate tutorial on SMath's table, hence I would not put it here since we would only discuss SQLite. You may skip ahead since the aim here is to get you know what a table really is + you don't actually need it to achieve the end goal).

```
data_heading := [ "Section Name" "weight" "Z_x" "ϕMp" "ϕMr" "BF" "Lp" "Lr" "I_x" "ϕVn" ]
```

Section Name	weight	Z_x	ϕMp	ϕMr	BF	Lp	Lr	I_x	ϕVn
I 100x50x4.5x6.8	8.34	40	9	5	1.4	0.6	3	172	64
I 120x58x5.1x7.7	11.1	64	14	8	2.1	0.7	3.3	331	88
I 140x66x5.7x8.6	14.3	96	21	12	2.9	0.8	3.6	579	114
I 160x74x6.3x9.5	17.9	137	30	18	3.9	0.9	3.9	944	146
I 180x82x6.9x10.4	21.9	188	41	25	5	0.9	4.2	1460	179
I 200x90x7.5x11.3	26.2	251	54	33	6.2	1	4.5	2162	216

Section Name	weight	Z_x	ϕMp	ϕMr	BF	Lp	Lr	I_x	ϕVn
W 100x50x5x7	9.3	42	9	5	1.4	0.6	3.1	178	72
W 125x60x6x8	13.1	74	16	10	2.3	0.7	3.5	394	108
W 150x75x5x7	14	98	21	13	3.5	0.9	3.2	642	108
W 148x100x6x9	21	150	32	20	3	1.2	5.4	981	128
W 175x90x5x8	18	152	33	20	4.6	1.1	3.8	1172	126
W 198x99x4.5x7	18	170	37	23	5.7	1.1	3.6	1498	128
W 200x100x5.5x8	21	200	43	27	6.1	1.1	3.9	1761	158
W 194x150x6x9	29.9	296	64	40	5.1	1.9	6.5	2585	168
W 248x124x5x8	25.1	305	66	41	8.6	1.4	4.3	3378	179
W 250x125x6x9	29	352	76	47	9.2	1.4	4.6	3893	216

This is really useful if you are trying to let's say, state the profile or state some informations at your appendix. We should then entered one more thing is that you should either control the syntax by displaying the lastError message, or making an exception handling case when declaring a query, both would not be covered in this tutorial (At this point, I expect you to do your own do dillegence).

Last Error, empty string indicates that there are no error in any execution

```
lastError = ""
```

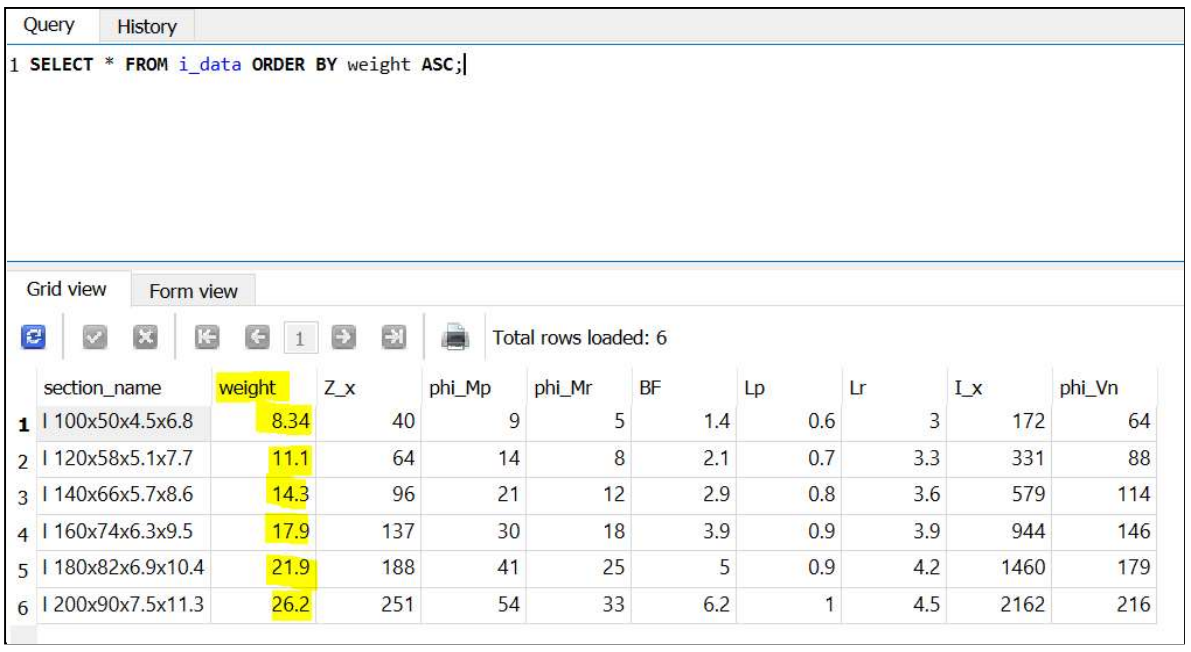
To give you more idea on how this works, you could try more sql syntax, if you have familiarize yourself enough with this workflow, you could skip ahead to section 4 as that is the main purpose of this tutorial. If you still need some time, feel free to retype and redo what I would do. We would continue to order things up. Let's say that we need to sort several things by a particular parameter (A rather nice case would be the consideration of selecting profile based on weight or Mp value). We would then SELECT and ORDER things up

```
This method would sort the table from the smallest to the largest number based on a particular parameter
GET_SORT_SMALL(params, table) := concat("SELECT * FROM ", table, " ORDER BY ", params, " ASC;")

This method would sort the largest to smallest number based on a particular parameter
GET_SORT_LARGE(params, table) := concat("SELECT * FROM ", table, " ORDER BY ", params, " DESC;")

I would sort the weight form the smallest to largest from i_data table
weight_StoL := SQLiteQuery(db_dir, GET_SORT_SMALL("weight", "i_data"))
```

Here's the result of the i_data table sorted based on the weight value. See weight_StoL comparison with the query inserted to sqlite studio

```
i_data table based on the weight sorted (ascended) weight value
```

$$weight\_StoL = \begin{bmatrix} \text{"I 100x50x4.5x6.8"} & 8.34 & 40 & 9 & 5 & 1.4 & 0.6 & 3 & 172 & 64 \\ \text{"I 120x58x5.1x7.7"} & 11.1 & 64 & 14 & 8 & 2.1 & 0.7 & 3.3 & 331 & 88 \\ \text{"I 140x66x5.7x8.6"} & 14.3 & 96 & 21 & 12 & 2.9 & 0.8 & 3.6 & 579 & 114 \\ \text{"I 160x74x6.3x9.5"} & 17.9 & 137 & 30 & 18 & 3.9 & 0.9 & 3.9 & 944 & 146 \\ \text{"I 180x82x6.9x10.4"} & 21.9 & 188 & 41 & 25 & 5 & 0.9 & 4.2 & 1460 & 179 \\ \text{"I 200x90x7.5x11.3"} & 26.2 & 251 & 54 & 33 & 6.2 & 1 & 4.5 & 2162 & 216 \end{bmatrix}$$



	section_name	weight	Z_x	phi_Mp	phi_Mr	BF	Lp	Lr	I_x	phi_Vn
1	I 100x50x4.5x6.8	8.34	40	9	5	1.4	0.6	3	172	64
2	I 120x58x5.1x7.7	11.1	64	14	8	2.1	0.7	3.3	331	88
3	I 140x66x5.7x8.6	14.3	96	21	12	2.9	0.8	3.6	579	114
4	I 160x74x6.3x9.5	17.9	137	30	18	3.9	0.9	3.9	944	146
5	I 180x82x6.9x10.4	21.9	188	41	25	5	0.9	4.2	1460	179
6	I 200x90x7.5x11.3	26.2	251	54	33	6.2	1	4.5	2162	216

```
I would sort the ϕMp value on the i_data table from the largest to the smallest
phiMP_LtoS := SQLiteQuery(db_dir, GET_SORT_LARGE("phi_Mp", "i_data"))
```

$$phiMP\_LtoS = \begin{bmatrix} \text{"I 200x90x7.5x11.3"} & 26.2 & 251 & 54 & 33 & 6.2 & 1 & 4.5 & 2162 & 216 \\ \text{"I 180x82x6.9x10.4"} & 21.9 & 188 & 41 & 25 & 5 & 0.9 & 4.2 & 1460 & 179 \\ \text{"I 160x74x6.3x9.5"} & 17.9 & 137 & 30 & 18 & 3.9 & 0.9 & 3.9 & 944 & 146 \\ \text{"I 140x66x5.7x8.6"} & 14.3 & 96 & 21 & 12 & 2.9 & 0.8 & 3.6 & 579 & 114 \\ \text{"I 120x58x5.1x7.7"} & 11.1 & 64 & 14 & 8 & 2.1 & 0.7 & 3.3 & 331 & 88 \\ \text{"I 100x50x4.5x6.8"} & 8.34 & 40 & 9 & 5 & 1.4 & 0.6 & 3 & 172 & 64 \end{bmatrix}$$

By now, you should have understood the basics of what we're trying to do with databases. We would take this a step futher by calling the inserted unit to SMath Studio. If you still don't understand what we're doing so far, feel free to watch some youtube videos on databases as it would improve your understanding about SQLite query. You would find yourself making good use of the learnt query later on.

Later on, we would be making our own app that could edit, view, or change a particular data. I would make this app using python alongside PySimpleGUI (the name of the library).

For now, we would continue to use an established database in SMath Studio before making anything else.

4. Using Database in SMath Studio

We would first make two types of method, we would be two types of method. We would load the numbers or the data and the unit. Since there are two tables, I would write two different method of the former. You could, technically speaking, make a single method to replace the table's name. I don't recommend this method since you would find yourself adjusting the table categories+parameters from time to time. You could follow along by typing this method:

```
ShapeQuery_1(a, b) := concat("SELECT ", b, " FROM i_data WHERE section_name is '", a, "';")
ShapeQuery_2(a, b) := concat("SELECT ", b, " FROM w_data WHERE section_name is '", a, "';")
UnitQuerty(PROP) := concat("SELECT unit FROM unit_data WHERE prop is '", PROP, "';")
```

You would then pass this method to the query. We would use a ***line()*** method here since we'd like to stop numeric evaluation to operate the unit alongst the properties.

$$I\_GetProp(SHAPE, PROP) := \left| SQLiteQuery(db\_dir, ShapeQuery\_1(SHAPE, PROP)) \right.$$

$$W\_GetProp(SHAPE, PROP) := \left| SQLiteQuery(db\_dir, ShapeQuery\_2(SHAPE, PROP)) \right.$$

$$GetUnit(PROP) := \left| str2num(SQLiteQuery(db\_dir, UnitQuerty(PROP))) \right.$$

You could now use the method. This is the original way of doing things found in SMath's forum.

$$I\_GetProp("I\ 100x50x4.5x6.8", "weight") = 8.34$$

$$W\_GetProp("W\ 100x50x5x7", "weight") = 9.3$$

$$GetUnit("weight") = 1\ \frac{kgf}{m}$$

We would then create a method that combines the properties alongside the unit. Here's the code:

$$WF(SHAPE, PROPERTY) := \left| W\_GetProp(SHAPE, PROPERTY) \cdot GetUnit(PROPERTY) \right.$$

$$I(SHAPE, PROPERTY) := \left| I\_GetProp(SHAPE, PROPERTY) \cdot GetUnit(PROPERTY) \right.$$

We could now call the method to access a certain parameter

$$I("I\ 100x50x4.5x6.8", "phi\_Mp") = 9\ kN\ m$$

$$WF("W\ 100x50x5x7", "weight") = 9.3\ \frac{kgf}{m}$$

This is the base of our method. But a problem arises; I can't remember all those clunky names, so we'd have to make a query that retrieves the section name. This calls for another method:

$$GET\_NAME(table) := concat("SELECT\ section\_name\ FROM\ ", table)$$

$$I\_Target := GET\_NAME("i\_data") = "SELECT\ section\_name\ FROM\ i\_data"$$

$$WF\_TARGET := GET\_NAME("w\_data") = "SELECT\ section\_name\ FROM\ w\_data"$$

We would then execute those query. We don't need any line() method since we'd be using and calling the result directly

$$I\_Name := SQLiteQuery(db\_dir, I\_Target)$$

$$WF\_NAME := SQLiteQuery(db\_dir, WF\_TARGET)$$

This would return an array containning section name of each profile

$$I\_Name = \begin{bmatrix} "I\ 100x50x4.5x6.8" \\ "I\ 120x58x5.1x7.7" \\ "I\ 140x66x5.7x8.6" \\ "I\ 160x74x6.3x9.5" \\ "I\ 180x82x6.9x10.4" \\ "I\ 200x90x7.5x11.3" \end{bmatrix}$$

$$WF\_NAME = \begin{bmatrix} "W\ 100x50x5x7" \\ "W\ 125x60x6x8" \\ "W\ 150x75x5x7" \\ "W\ 148x100x6x9" \\ "W\ 175x90x5x8" \\ "W\ 198x99x4.5x7" \\ "W\ 200x100x5.5x8" \\ "W\ 194x150x6x9" \\ "W\ 248x124x5x8" \\ "W\ 250x125x6x9" \end{bmatrix}$$

This is quite convenient But it's such a hastle to recall a list whenever we need to recall the section name. It's better if we just choose a profile once and change it later on when different cases arise. To do that, we would be using a widget called ComboBox List. This also requires another plugin installation (See Fig 39). I would also not explain how this works since you could find many online examples.

Fig 39. ComboBox List Region Plugins

We would then fill the components by returning it's index or it's 1st element. So now, you or another user should only need to choose the desired shape and entered the parameters. Here's what the end result would look like:

Choose the desired shape

W 100x50x5x7

WF_OPTION

$$WF(WF\_OPTION, "weight") = 9.3\ \frac{kgf}{m}$$

$$WF(WF\_OPTION, "phi\_Mp") = 9\ kN\ m$$

Choose the desired shape

I 100x50x4.5x6.8

I_OPTION

$$I(I\_OPTION, "weight") = 8.34\ \frac{kgf}{m}$$

$$I(I\_OPTION, "phi\_Mp") = 9\ kN\ m$$

$WF(WF\_OPTION, \text{"Lp"}) = 0.6\ \text{m}$ $I(I\_OPTION, \text{"Lp"}) = 0.6\ \text{m}$

If you are going to use a particular database regularly, I would suggest putting the highlited code in a snippet. Here's how you do it:

1. Open your SMath Studio installation path, it would be installed (by default) in **C:\Program Files (x86)\SMath Studio**. You would then see **Fig 40.**

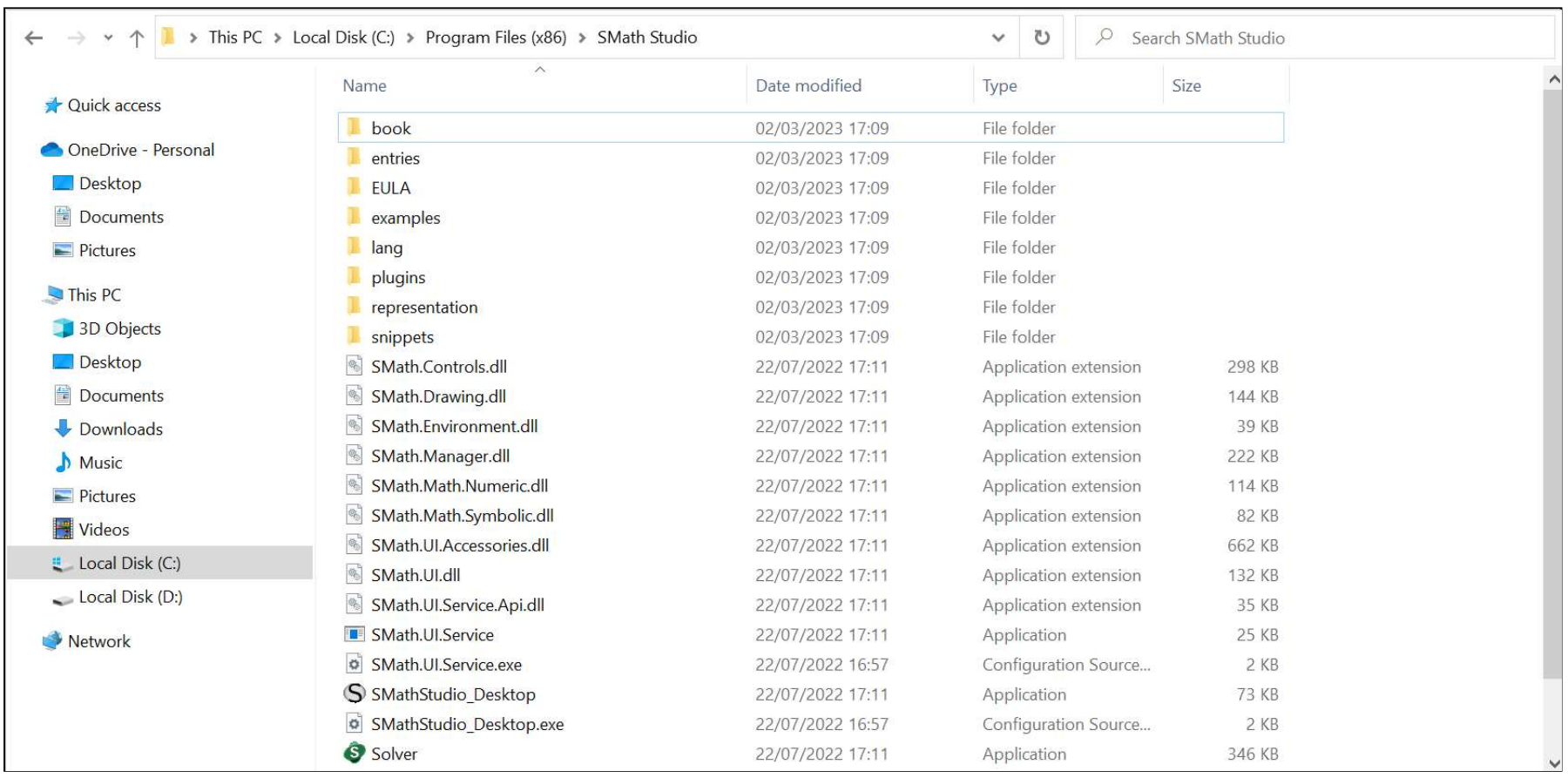


Fig 40. SMath Studio's installation directory

2. Make a new folder called database (See **Fig 41**). This folder would contain all the databases (I suggest you putting your databases somewhere as oppose to separate locations).

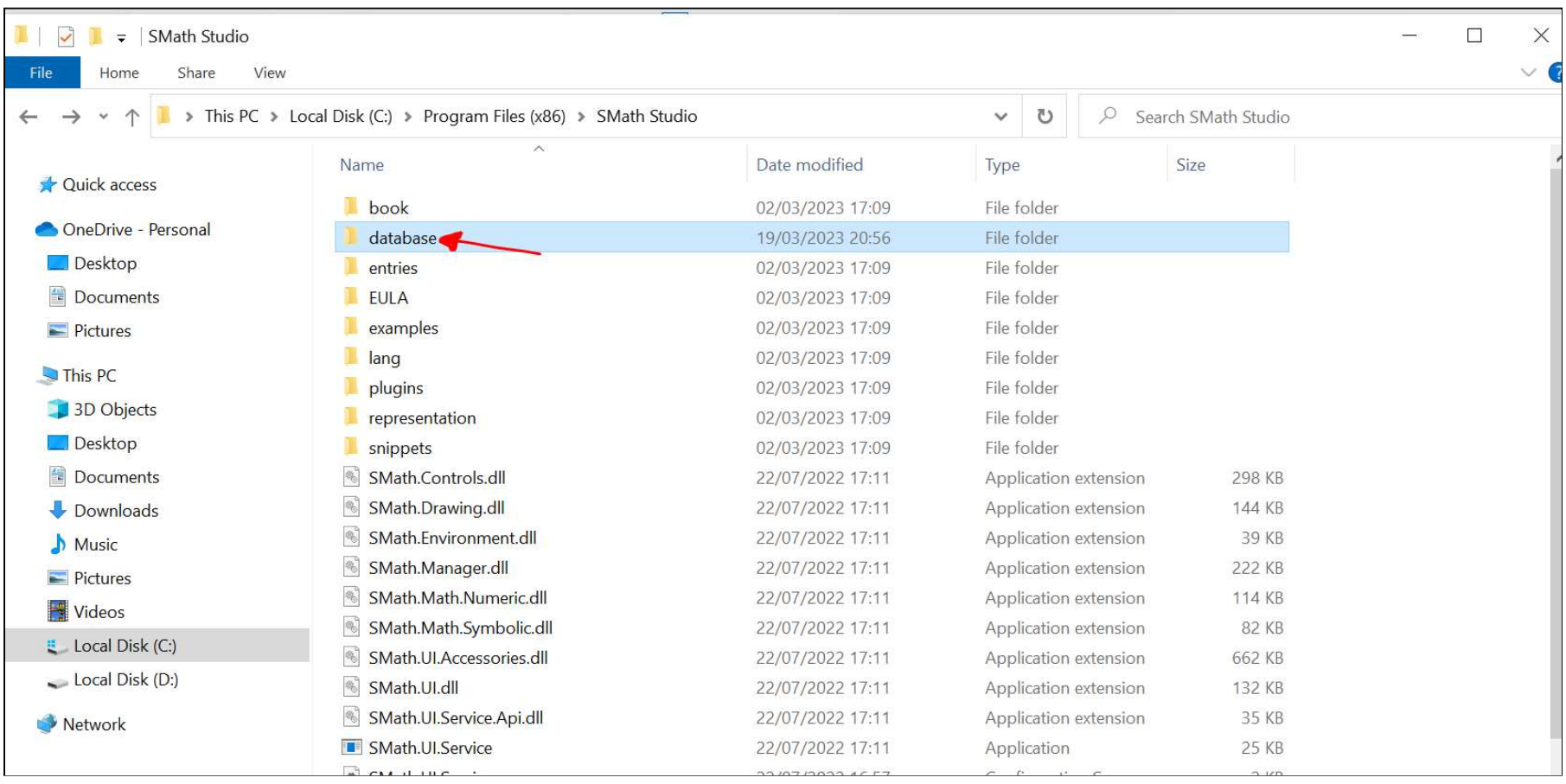


Fig 41. Creating new folder to hold database files

3. Put your ***.db*** file to the datbase folder (just cut and paste the database)(See **Fig 42**). We would now make this folder as the default location of any database that we'd make in the future.

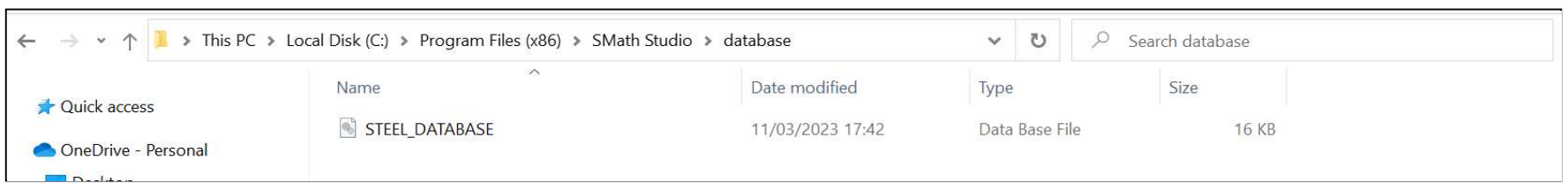


Fig 42. Putting the created and filled database to the created folder

4. Copy and Paste the highlited code in cyan to a new smath worksheet. (See **Fig 43**). You can change back the color of the text background to white or just let them be. The thing here is that we would load this code everytime we'd use the database (So please be wary of standard naming conventions).

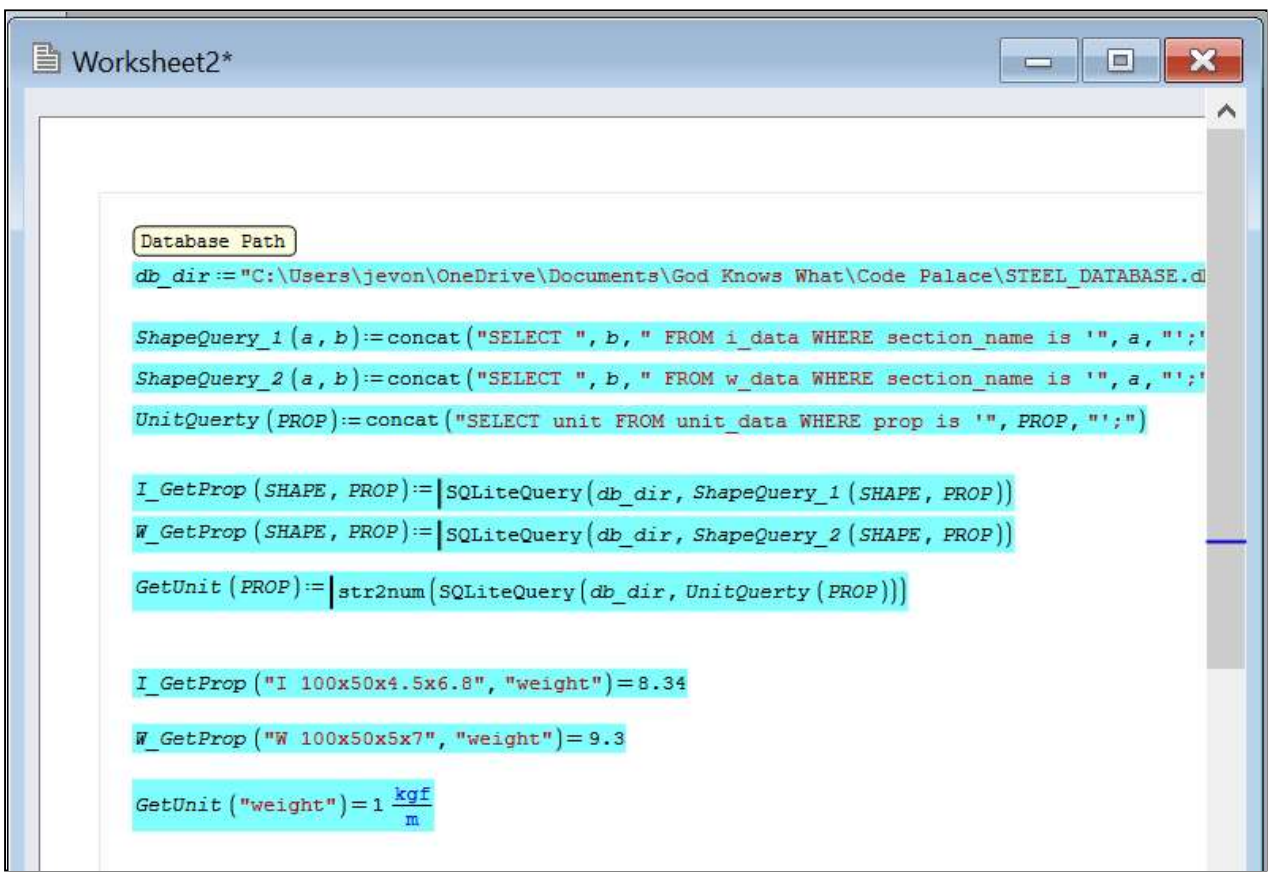


Fig 43. Copy + Paste the highlited code to a new SMath Studio Worksheet

5. We'd change the database path in **Fig 43.** to the ones that we'd just made in **Fig 42.** (See **Fig 44**)

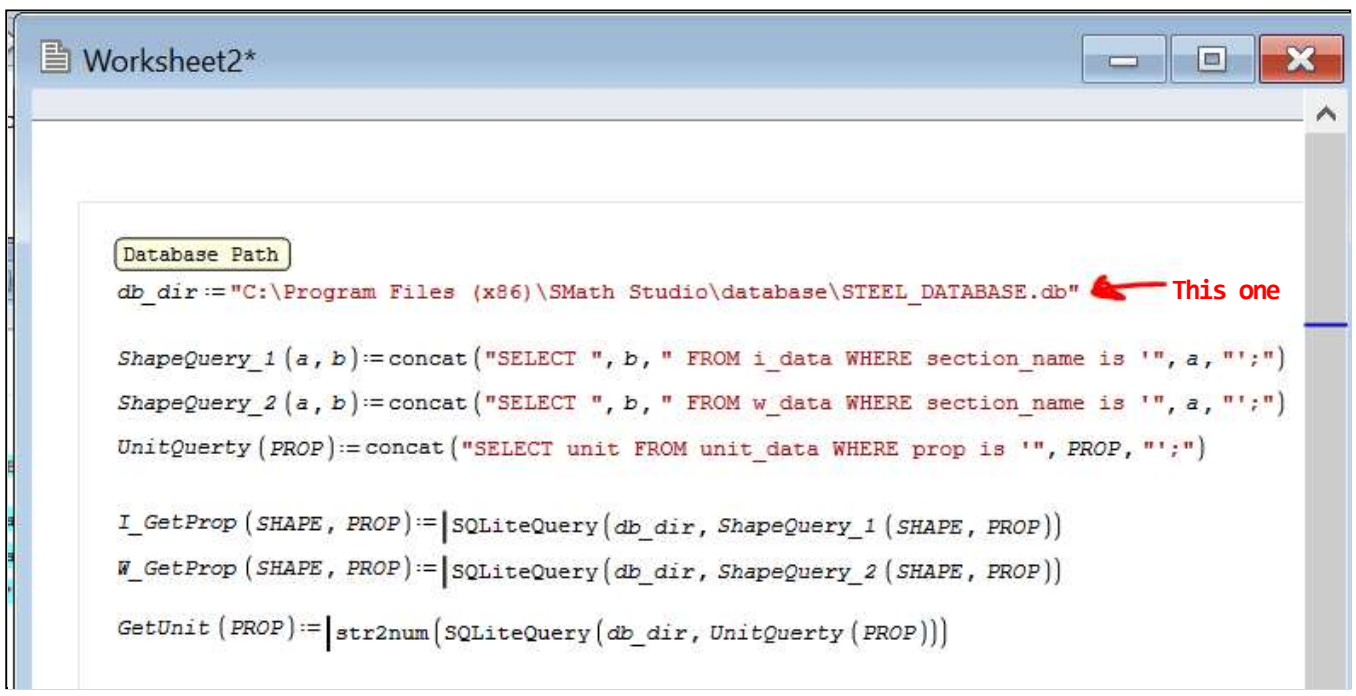


Fig 44. Change the database directory location

5. We would then go to **File** -> **Properties**. You would see a pop up in **Fig 45.** This File properties allows users to input some information which is then parsed as SMath's .xml metadata. For those of you who doesn't code, metadata is a descriptive information of a file.

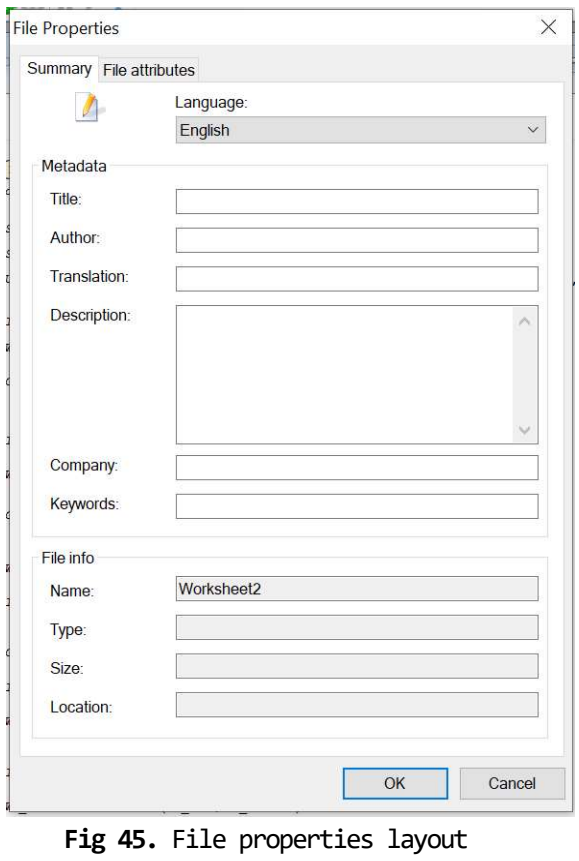


Fig 45. File properties layout

6. We would then fill out this information. The language doesn't really matter since most type of encoding is covered on utf-16. Here's what mine looks (**Fig 46**):

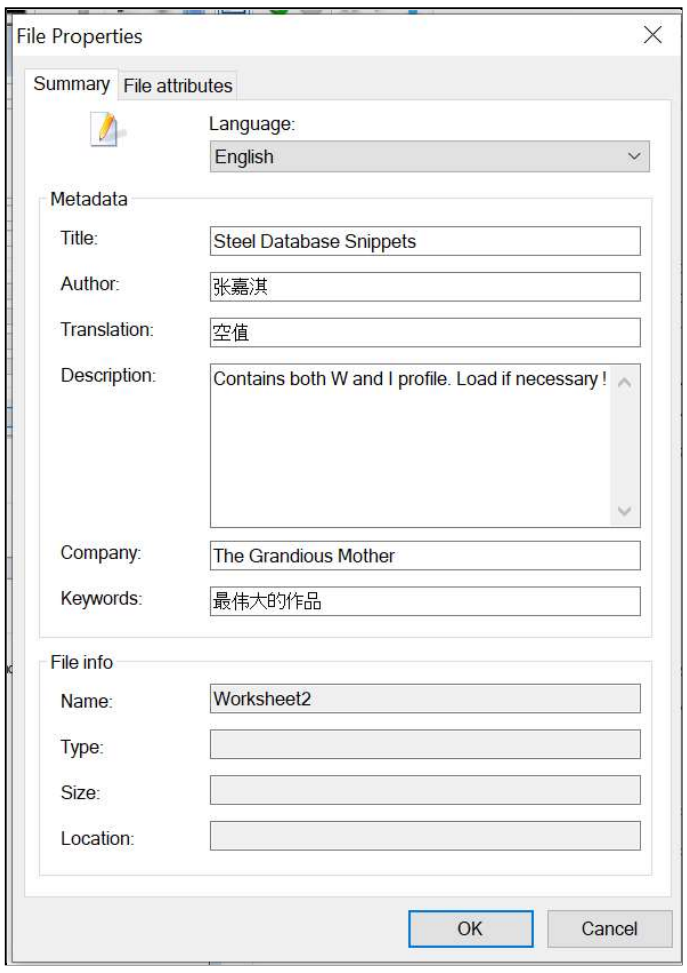


Fig 46. Filled Metadata

7. We would then save this worksheet on the Snippets folder, which is located on the same folder alongst the database. (See **Fig 47**)

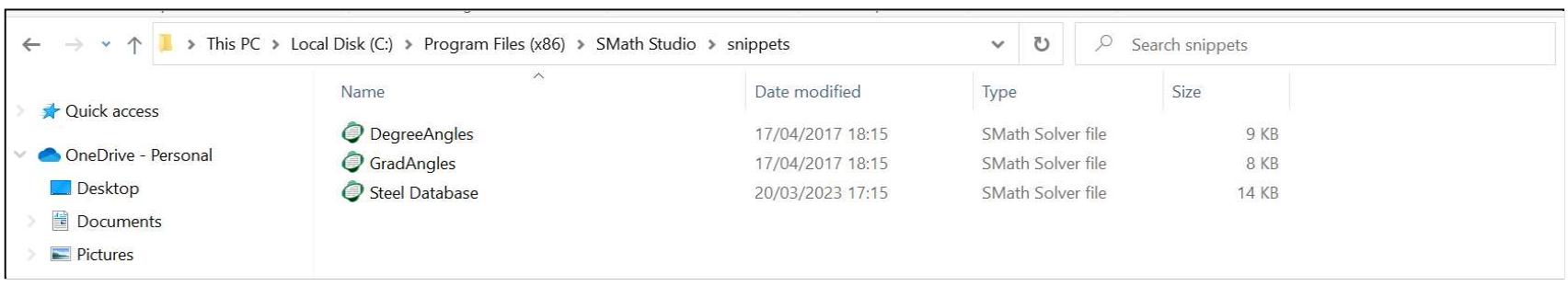


Fig 47. Saved SMath Worksheet in snippets folder

> ***Note:***
> I Based on some earlier review, some of you can't save the file on the installation folder. You can save it somewhere first then just cut and paste the file. Another solution would be disabling administrator permits (which is really dangerous if you don't know what you're doing, so yeah, just stick to some good old cut + paste for the time being).

8. We could then insert this snippet by going to ***tools*** -> ***Snippet Manager***. You'd see the created database snippets alongside other snippets. To load this up, you just need to insert the database. It would by default wrap the whole code in an area region (See **Fig 50**).

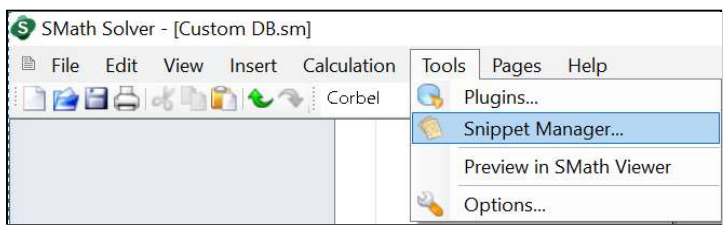


Fig 48. Snippet Manager Location

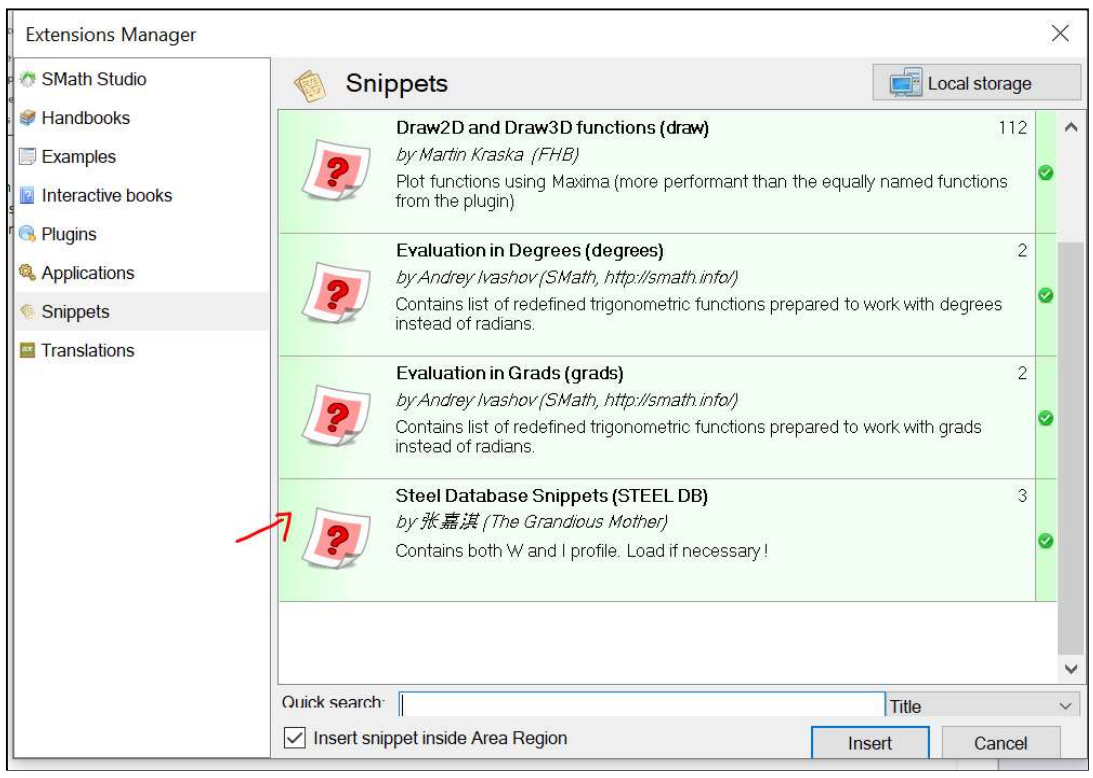


Fig 49. Lists of Snippets

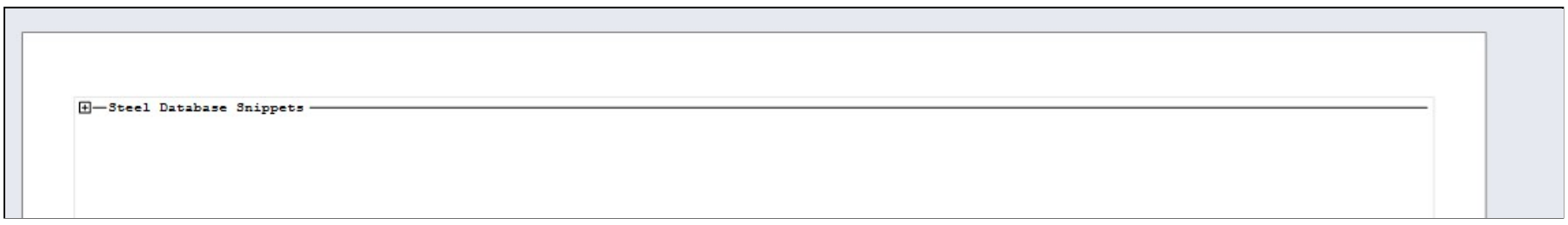


Fig 50. Inserted Snippets

9. Once you start getting confirmtable with this workflow, you would find yourself nesting more and more stuff, It's better to put an area that covers all snipets. This way, you could easily maintain your database and snippets (See the example below:)

JJ Library

Nest the snippets here

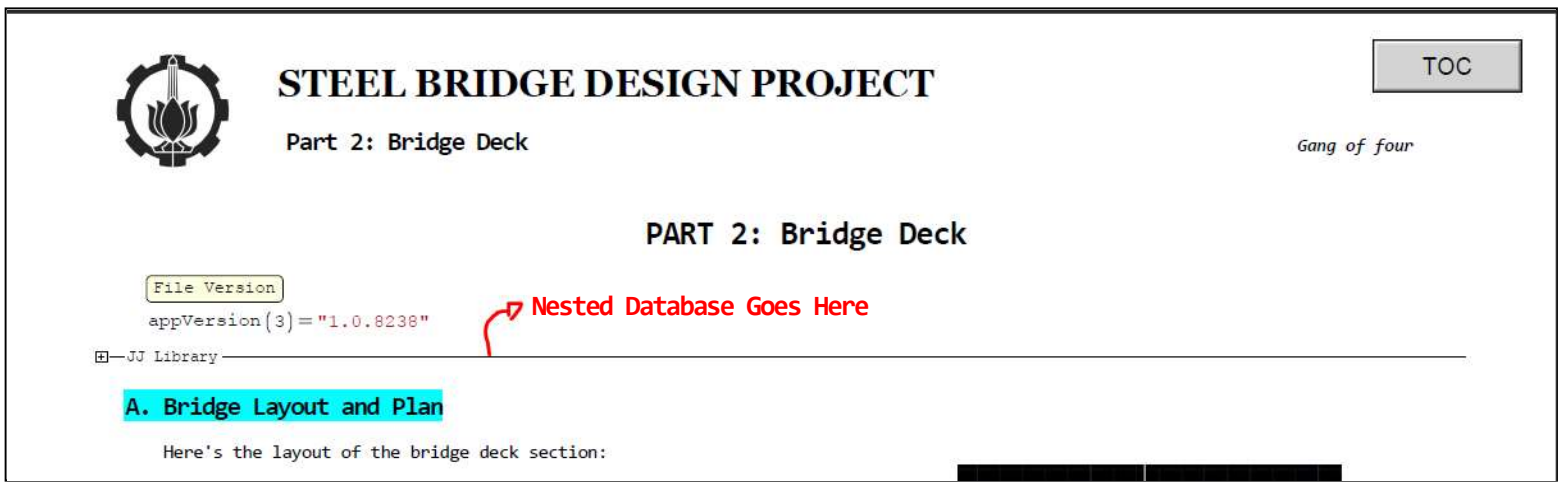
Steel Database

Concrete Database

Rebar Database

---

10. This is how I would use the database (taken from the same bridge engineering class report):



STEEL BRIDGE DESIGN PROJECT

Part 2: Bridge Deck

TOC

Gang of four

PART 2: Bridge Deck

File Version

appVersion(3) = "1.0.8238"

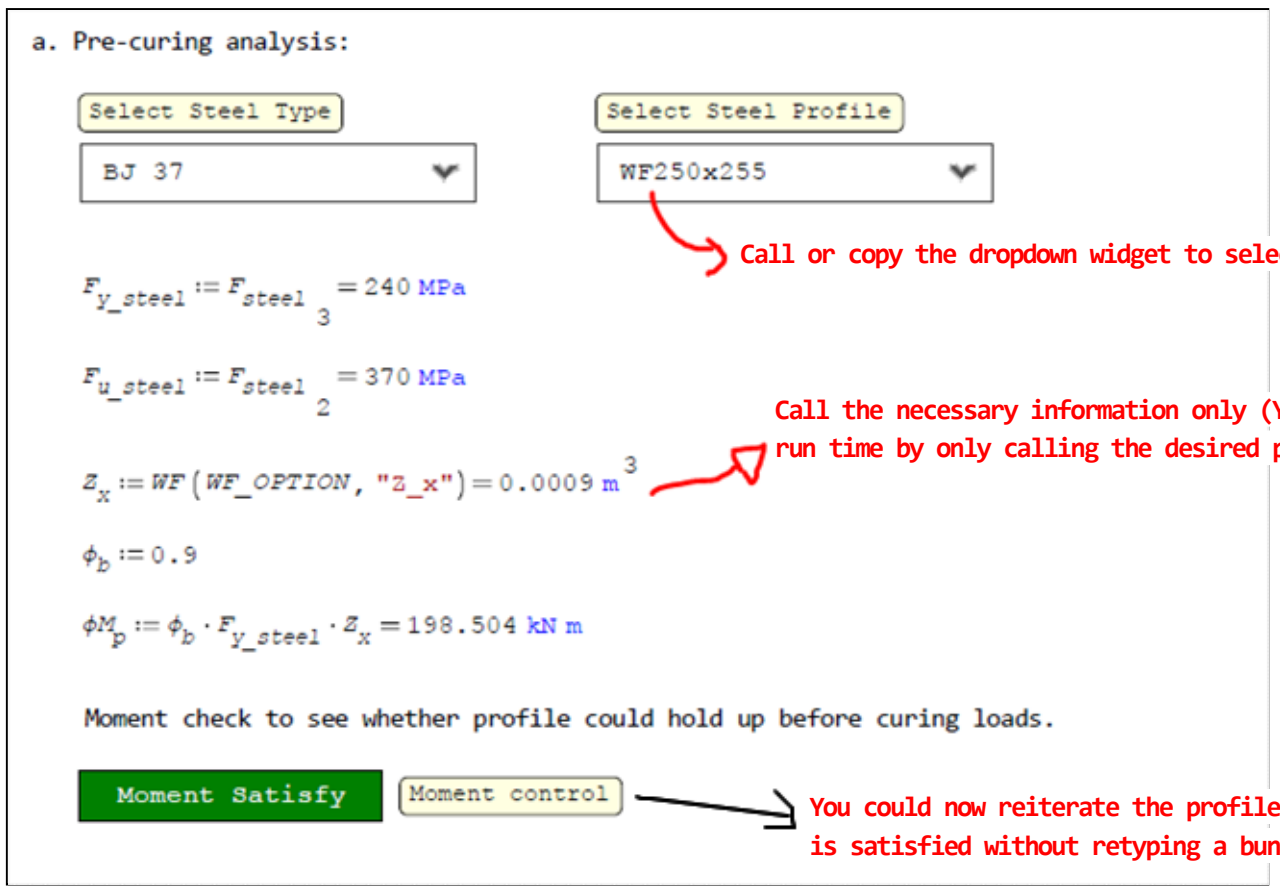
Nested Database Goes Here

JJ Library

A. Bridge Layout and Plan

Here's the layout of the bridge deck section:

[redacted]



a. Pre-curing analysis:

Select Steel Type: BJ 37

Select Steel Profile: WF250x255

Call or copy the dropdown widget to select a profile

$F_{y\_steel} := F_{steel}{}_{3} = 240\ MPa$

$F_{u\_steel} := F_{steel}{}_{2} = 370\ MPa$

Call the necessary information only (You would save a lot of run time by only calling the desired parameter)

$Z_x := WF(WF\_OPTION, "Z\_x") = 0.0009\ m^3$

$\phi_b := 0.9$

$\phi M_p := \phi_b \cdot F_{y\_steel} \cdot Z_x = 198.504\ kN\ m$

Moment check to see whether profile could hold up before curing loads.

Moment Satisfy

Moment control

You could now reiterate the profile until the condition is satisfied without retyping a bunch of time