**C++ Parsing in Python using Clang**

**Ref:** <https://eli.thegreenplace.net/2011/07/03/parsing-c-in-python-with-clang#id7>

**Libclang**

This is a C API developed by Clang team as a tool for analyzing the C/C++/ObjC code.

It allows the user to examine the parsed code at the level of an Abstract Syntax Tree (AST).

**Python bindings to Libclang**

* Script should have ***clang.cindex*** module. Once clang is installed you need to copy it’s path to PATH environment variable.

Eg: in Linux or Mac, “*export PATH=/<path\_to\_clang>:$PATH*”

* Clang.cindex should be able to find the *libclang* library. For Linux/MacOS, it can be either *libclang.so* or *libclang.dylib* depending on how it’s installed. For windows, it is *libclang.dll*

The below line will be added to the python script so that ***clang.cindex*** module can find the libclang library:

*Eg: clang.cindex.Config.set\_library\_file('C:/Program Files(x86)/LLVM/bin/libclang.dll')*

**How parsing works?**

We can see how a sample c++ code is parsed and what all details can be fetched.

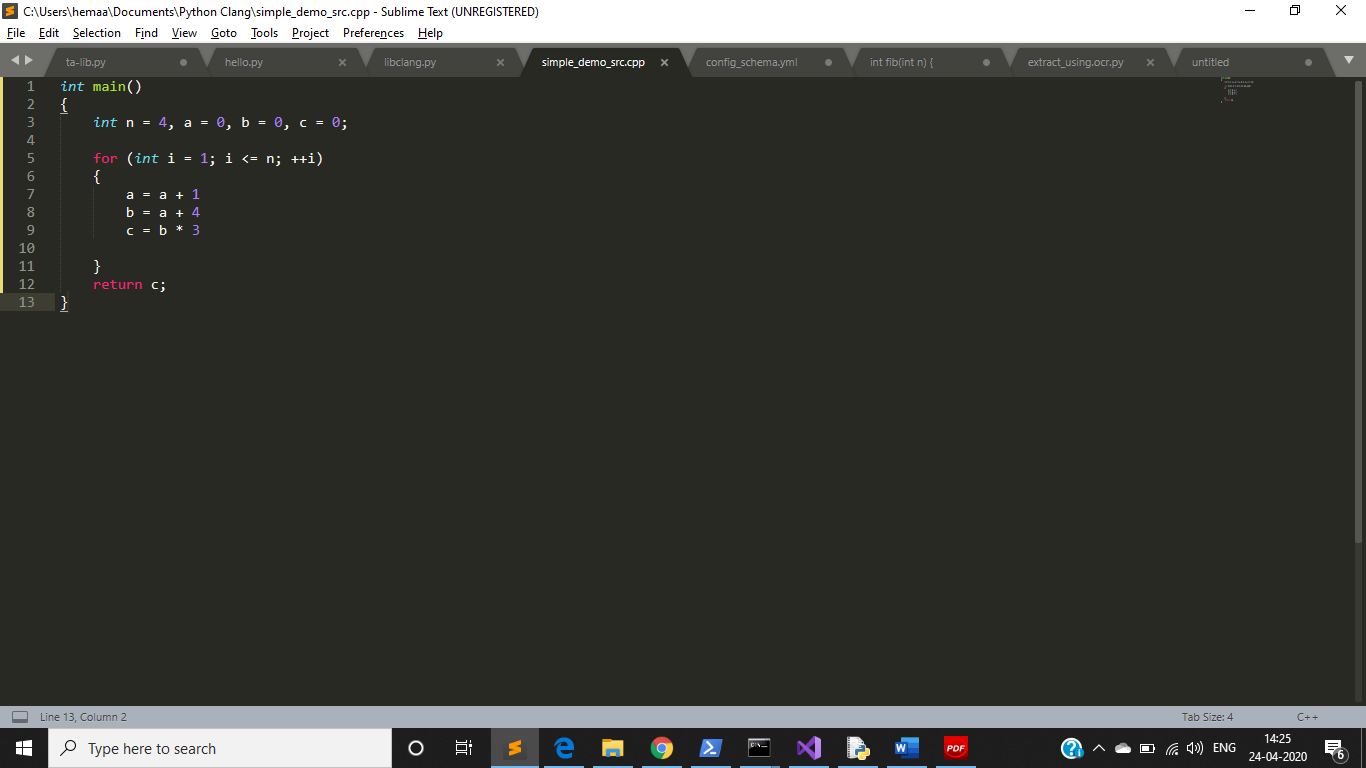


Fig 1.a Simple cpp code

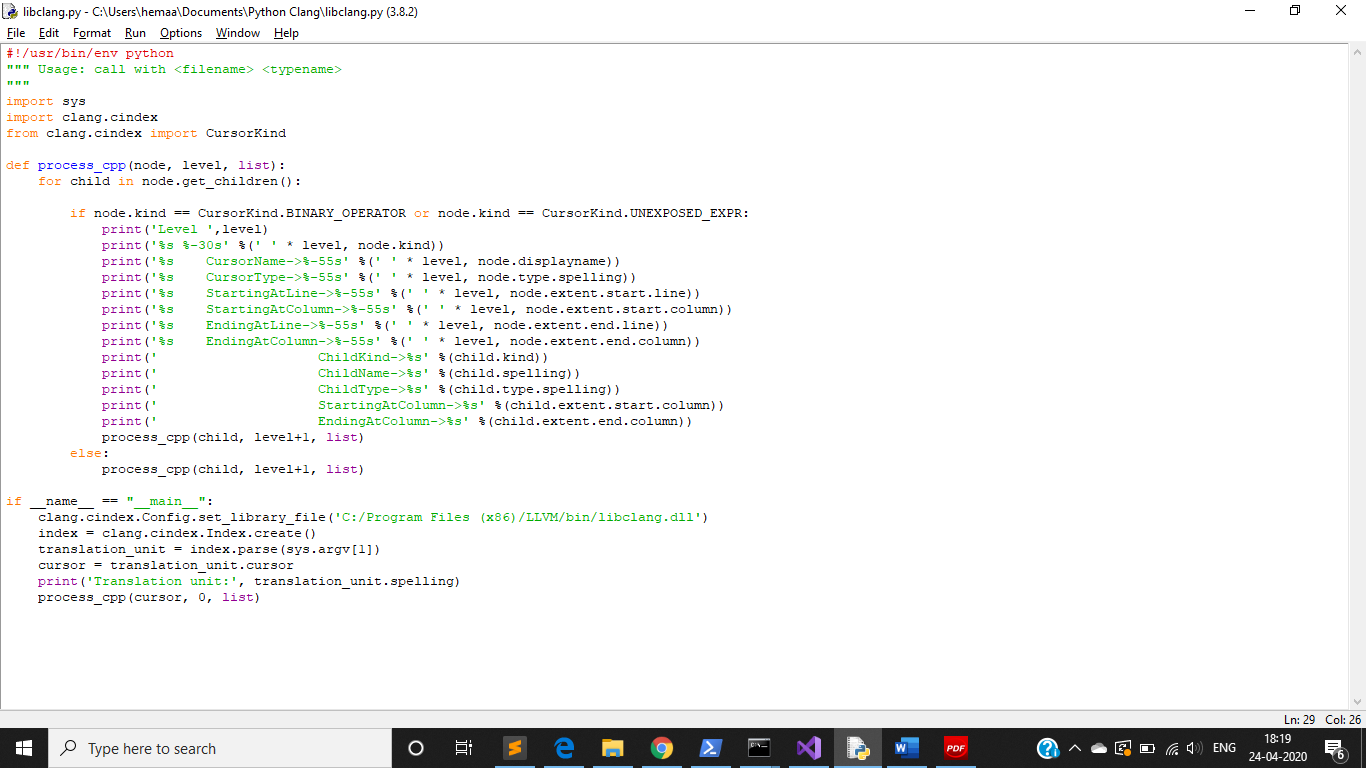
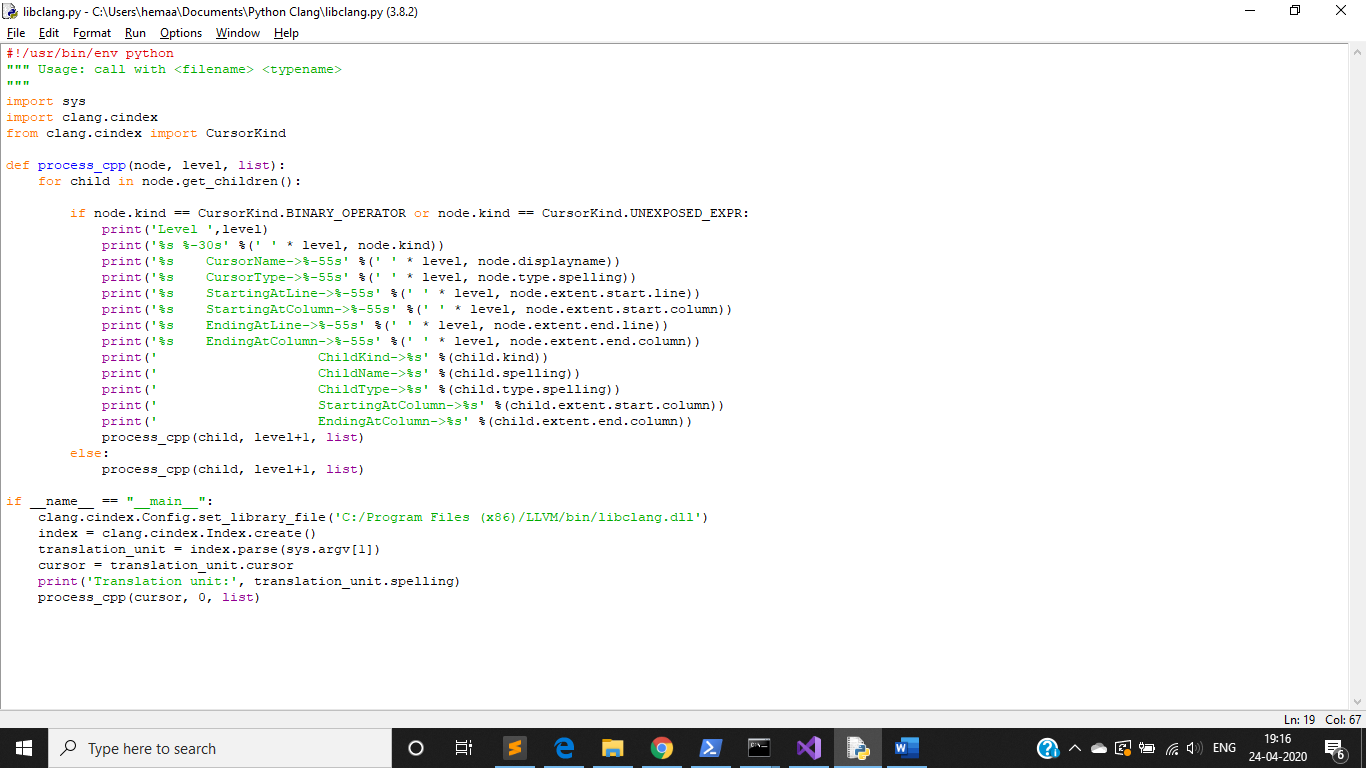


Fig 1.b Python code parsing the cpp code in Fig 1.a

Fig 1.a is the sample cpp code that we have used in our python code.

And Fig 1.b is the Python code. *[Please note that there has been some modification on the print statements when compared to the code that I submitted for first milestone. This modification has been done to get a better view of the tree nodes]*

As you can see inside the *\_\_main\_\_* function, first we have set the library file path for libclang library*. [Note: The path will vary for Linux/MacOS and this is mentioned at the beginning of this doc]*



We run the script from the terminal in the format “python3 <pyfile.py> <cppfile>

***index.parse*** takes cpp file as its argument and returns a single translation unit from that file. To explore the AST of the parsed translation unit, we can fetch the top-most cursor using ***translation\_unit.cursor***

This ***cursor*** will be mentioned as node inside the function ***process\_cpp()***. We will also be seeing the level of trees.

***Cursor*** has many attributes and some of them are:

* kind - an enumeration specifying the kind of AST node this cursor points at
* spelling - the source-code name of the node
* location - the source-code location from which the node was parsed
* get\_children – this is used to get the child node of a given cursor

Detailed information on the properties/clang library bindings are available at <https://github.com/llvm-mirror/clang/blob/master/bindings/python/clang/cindex.py>

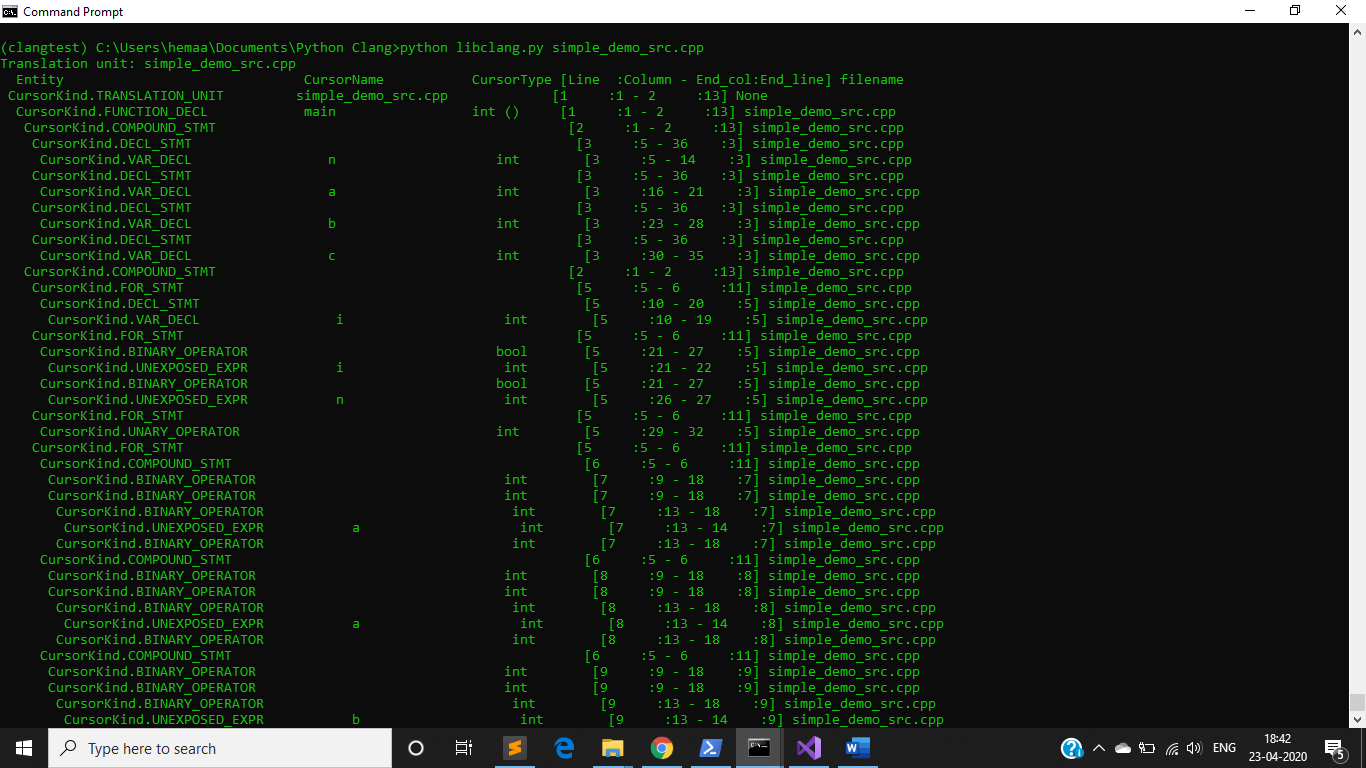


Fig 2.a Output of first milestone

As you can see Fig 2.a was the initial output, as I mentioned earlier, I did some modification in the python script to get a better view. While doing this I was able to display the parent and child relationship as shown in Fig 2.b *[As I was able to figure this out, I guess I will be able to analyze the dependency. But it would take one or two days.]*

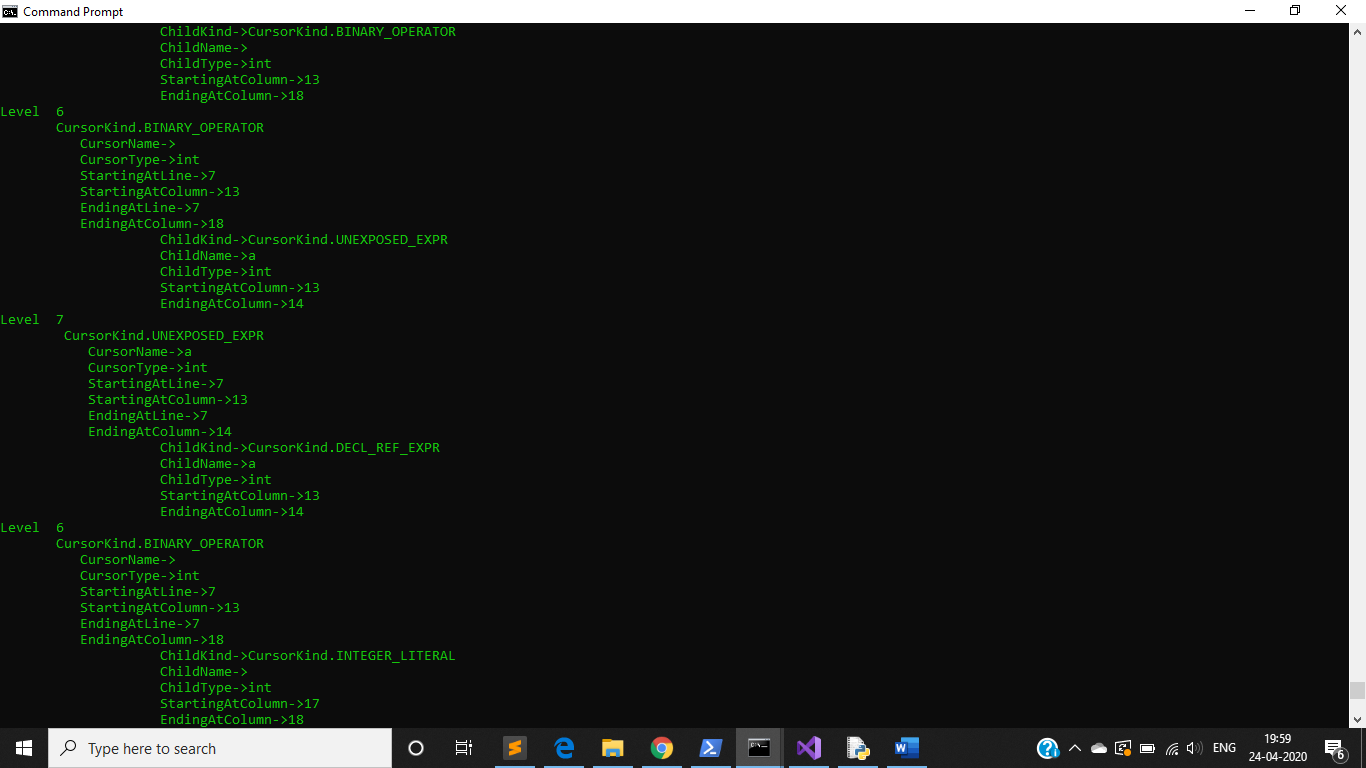
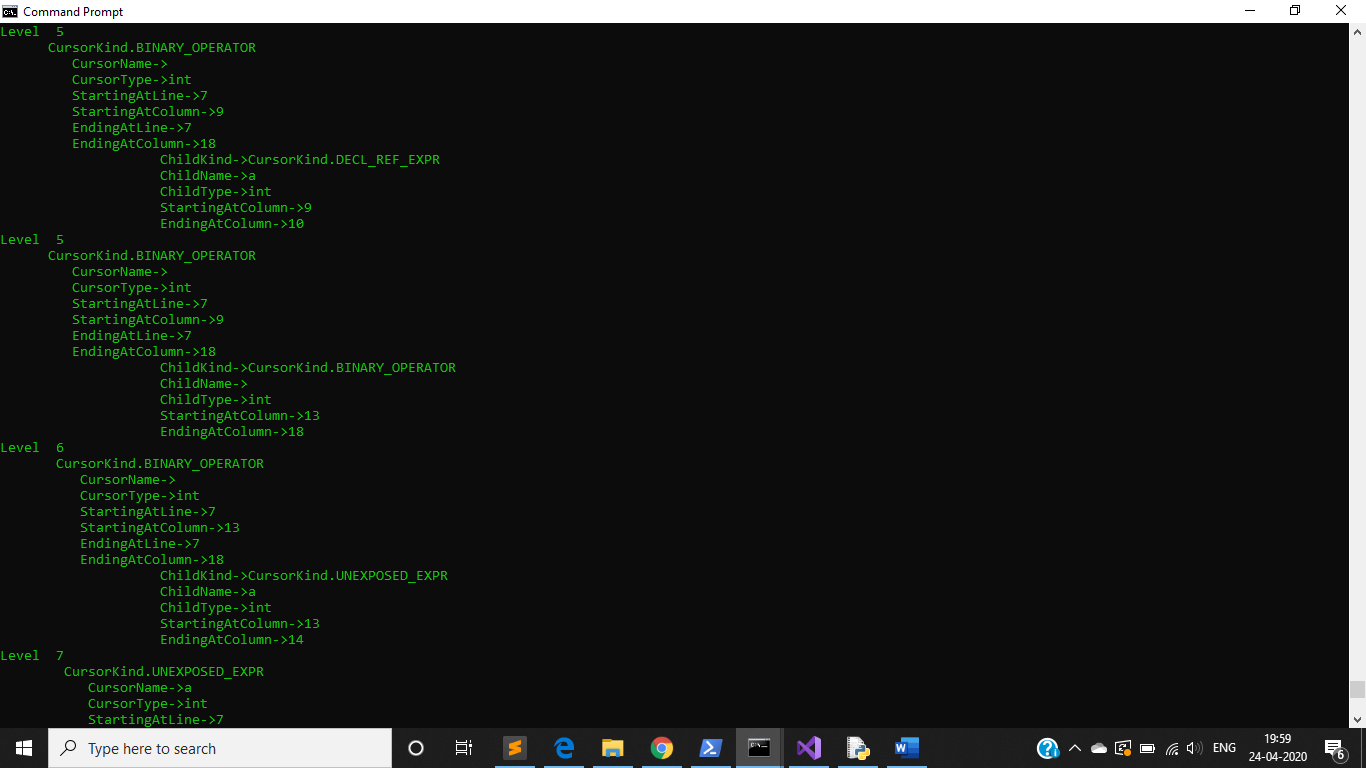


Fig 2.b Modified version of output

I referred the above mentioned github repository to understand the ***CursorKind*** and other properties. Here I have taken a part of output that corresponds to the line a = a + 1.

You can see Fig 2.c to get an idea of how the parent and child is represented. Each ***CursorKind*** is having a ***ChildKind***.

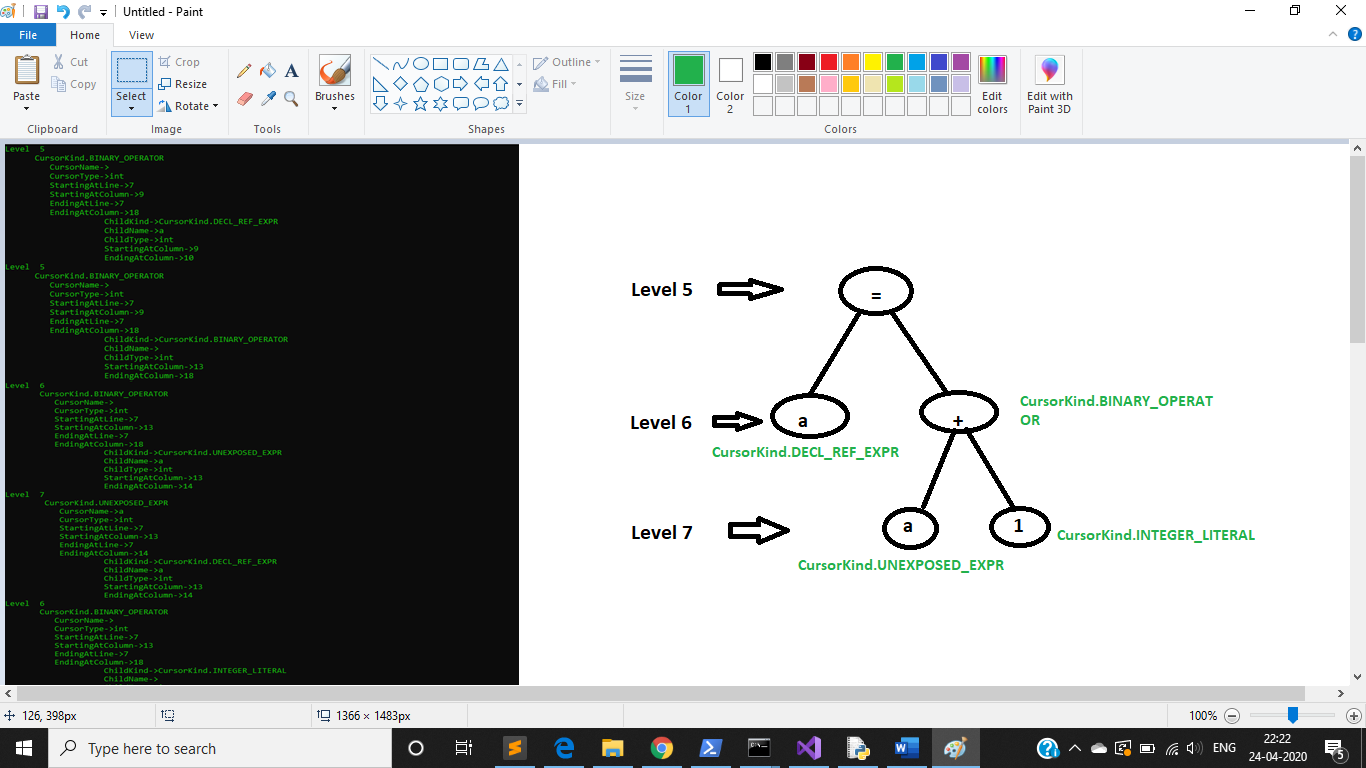
`

Fig 2.c Comparison of output with the expression a = a + 1

If you see Fig 1.b, I have written an if condition to print the node details only when

***node.kind == CursorKind.BINARY\_OPERATOR or node.kind == CursorKind.UNEXPOSED\_EXPR:*** .

We can remove this condition if we want all the node details.

Code will be modified based on dependency analysis findings.

Till this point, I have referred the below sites:

1. <https://eli.thegreenplace.net/2011/07/03/parsing-c-in-python-with-clang#id7>
2. <https://github.com/llvm-mirror/clang/blob/master/bindings/python/clang/cindex.py>
3. <http://szelei.me/code-generator/>
4. <https://pypi.org/project/clang/>
5. <https://releases.llvm.org/download.html#10.0.0>

**cppdep**

This is said to be a Dependency analyzer for C/C++ projects. But there are certain limitations as per the below reference.

Ref : <https://pypi.org/project/cppdep/>

Lot of issues have been logged for cppdep here <https://github.com/rakhimov/cppdep/issues> and the last update has been done on 3rd July, 2019.

I even tried to install cppdep and execute a sample but it got errored out for yaml file and there was no proper source to understand what could be the issue. May be the development is under progress.

**Further Additions:**

Till now, you saw how the cpp code is parsed and how does the structure of code looks like in Fig 2.b & 2.c. That was the basic level of parsing. Now to find the dependencies among variables within a for-loop, I have taken a sample code as shown below in Fig 3.a

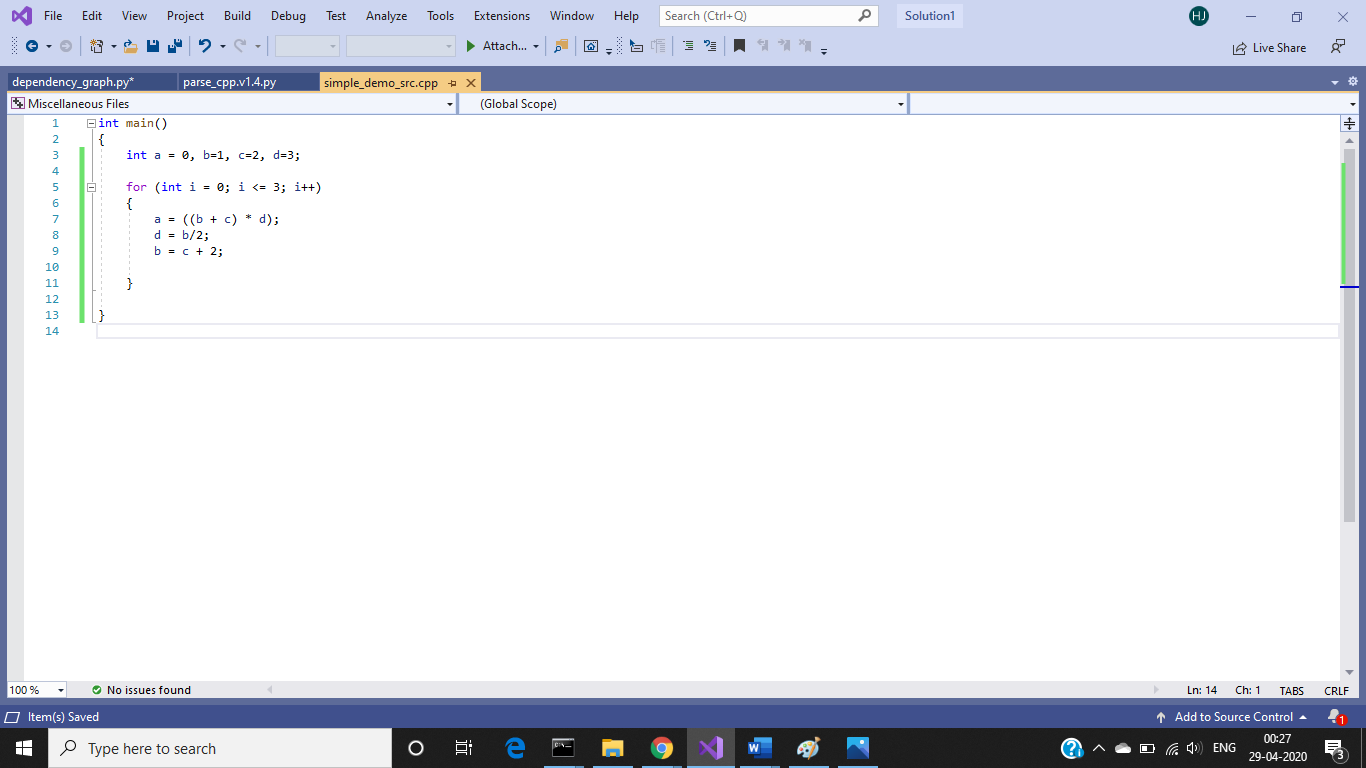


Fig 3.a Sample cpp code

As I have already mentioned on how parsing works in the earlier section, here I will discuss on the ***CursorKind*** associated with the above code.

The cpp code has:

* A main() function
* Inside main(), few variables declared
* A for-loop
* Inside for-loop, few binary operations on the declared variables
* Code clearly shows:
  + ‘a’ depends on ‘b’, ‘c’, ‘d’
  + ‘d’ depends on ‘b’
  + ‘b’ depends on ‘c’

Our agenda is to analyse the above dependency and represent it as a graph. In python, we can represent graph as an adjacency list or we can make use of libraries to draw a graph. I will show the output in both ways.

**What Python code does?**

* **Parsing the cpp code**
* **Collects the details of declared variables into a dictionary as they will represent the node for the graph**
* **While parsing the for loop,** 
  + **code will fetch the number of iterations. [It’s not a built-in functionality]**
  + **will check the binary operations, identify the neighbour nodes(dependency) and makes an adjacency list that contains the node and its list of edges like this: *[defaultdict(<class 'list'>, {'a': ['b', 'c', 'd'], 'd': ['b'], 'b': ['c']})***
* **Using the adjacency list, a graph is shown.** [You can refer<https://www.youtube.com/watch?v=bs2er-CleeI> to get an idea of adjacency list]

**Let’s look at how the code works?**

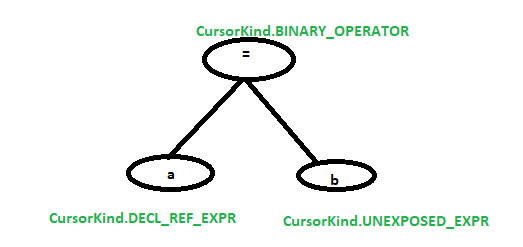
Parsing starts from the root and then followed by its child nodes.

Inside function parse\_cpp(), we will be calling another function process\_compound\_statement(), when the current node kind equals to ***CursorKind.COMPOUND\_STMT*** [In cpp code this will be the curly braces ‘{}’]

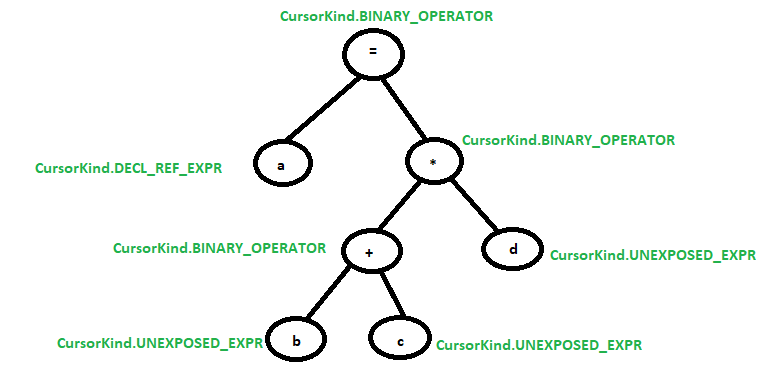
Inside function process\_compound\_statement(), the following takes place:

* Checks if the node kind is CursorKind.DECL\_STMT[In cpp, this belongs to the variable declaration part]
  + If yes, a function process\_decl() is called. This function will insert all the declared variables as a key into a dictionary.
* Checks if the node kind is CursorKind.BINARY\_OPERATOR or CursorKind.COMPOUND\_ASSIGNMENT\_OPERATOR [This is the binary operations such as +, - and other arithmetic, += compound]
  + If yes, a function process\_binary\_operation() is called. From this point, our code will start collecting the dependencies into an adjacency list based on certain conditions.
* For “a = b “, when the node kind matches as below, ‘b’ will be added to ‘a’ ‘s dependency list

CursorKind.UNEXPOSED\_EXPR denotes a variable here. For more info on the cursor kinds refer <https://clang.llvm.org/doxygen/group__CINDEX.html#ggaaccc432245b4cd9f2d470913f9ef0013aa4cd2c9319493a279d14815173e528a6>



* For “a = (b + c) \* d “, when the node kind matches as below, ‘b’, ‘c’,

‘d’ will be added to ‘a’ ‘s dependency. [Please note that when the right of ‘=’ is again a binary operation, then we have written another function process\_right\_nodes\_for(node, left\_expr) where left\_expr will be ‘a’ and the function will do the same condition checks and identifies the dependencies] 

* Similar way it works for other operations. Either binary or compound type. [With this sample, you can easily understand the conditions written within python code]
* Checks if the node kind is CursorKind.FOR\_STMT. This is where we will try to get the number of iterations of the loop.
  + If yes, process\_for\_loop(node, string) is called where an empty string is passed as an argument to store the for loop declaration stmt.
    - We use a get\_tokens() method to collect the for-loop statement and append those token’s spelling into the string
    - Split the string using ‘;’ separator and add to a list. We will have 3 items in the list like this [‘i=0’, ‘i<=3’, ‘i++’]. Now this list is passed to find\_for\_loop\_iteration()
    - On this list, we do some regular expression matching and gets the start, stop and step for the for-loop statement. And then calculates the number of times loop executed = (stop – start)/step [This number can be used to show dependency size]

Once the processing is done, control of the program comes back to the \_\_main\_\_ function. I have written a print statement as below for displaying the adjacency list representation:

for node, edges in dependency\_order.items():

print(node + " -> ", edges)

**How to draw a Graph?**

To draw a graph in python, we can use networkx and matplotlib packages. To install those, run the following:

* + pip install network
  + pip install matplotlib

(On MacOS, use pip3)

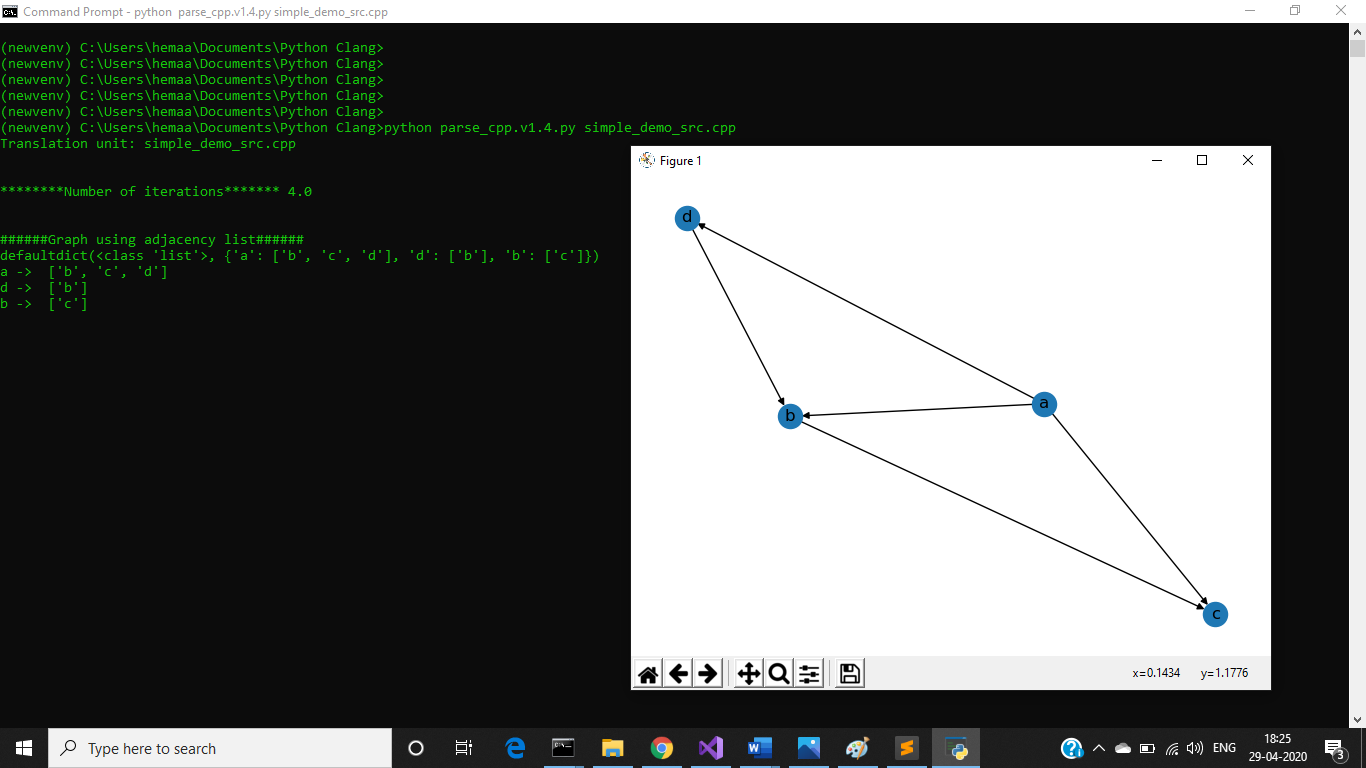
This package should be enough.

In our python code, a function draw\_graph() is written.

Finally, for the sample cpp code shown in Fig 3.a, output will be as below:

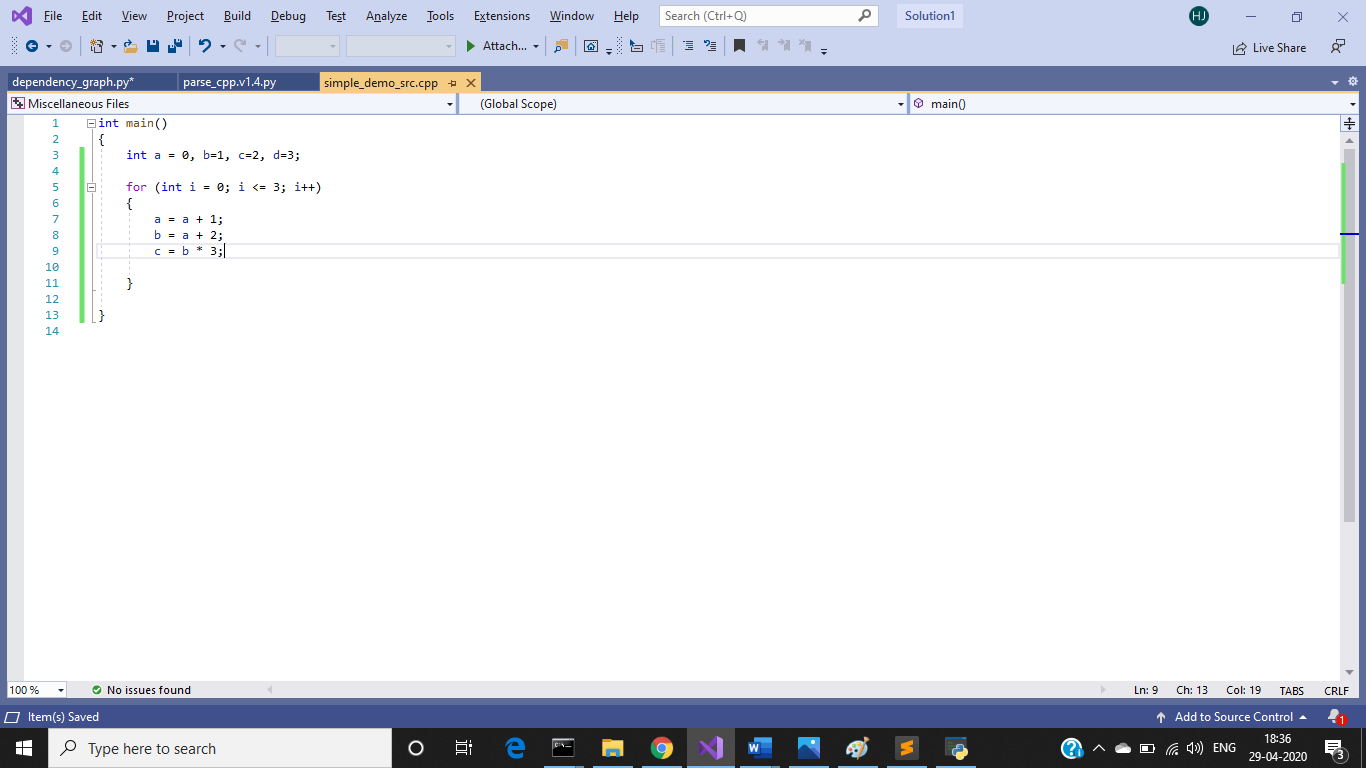
We display the number of iterations , an adjacency list and a graph.

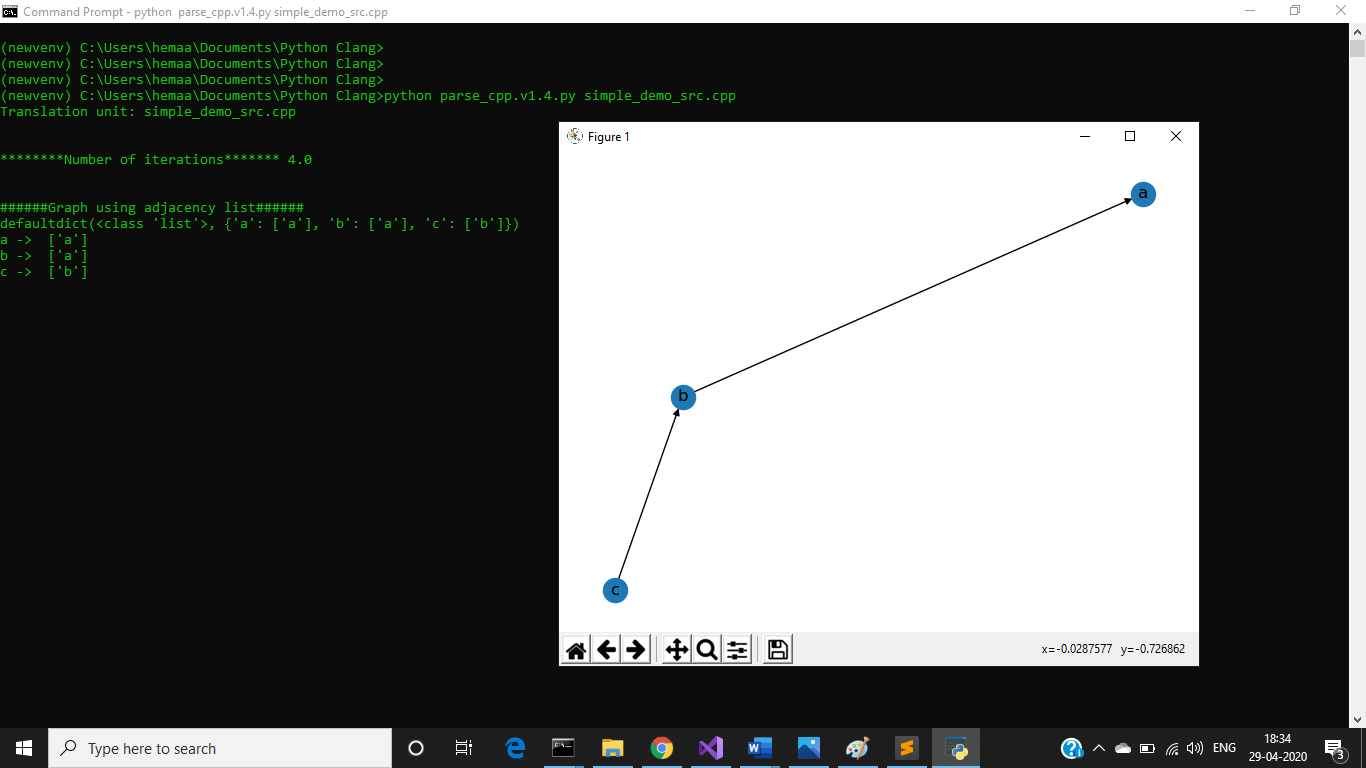
The starting and ending is in the order of how the nodes are visited. If you see below, it starts from node ‘a’ and ends at node ‘b’ -> ‘c’.



***Few additional test case examples:***

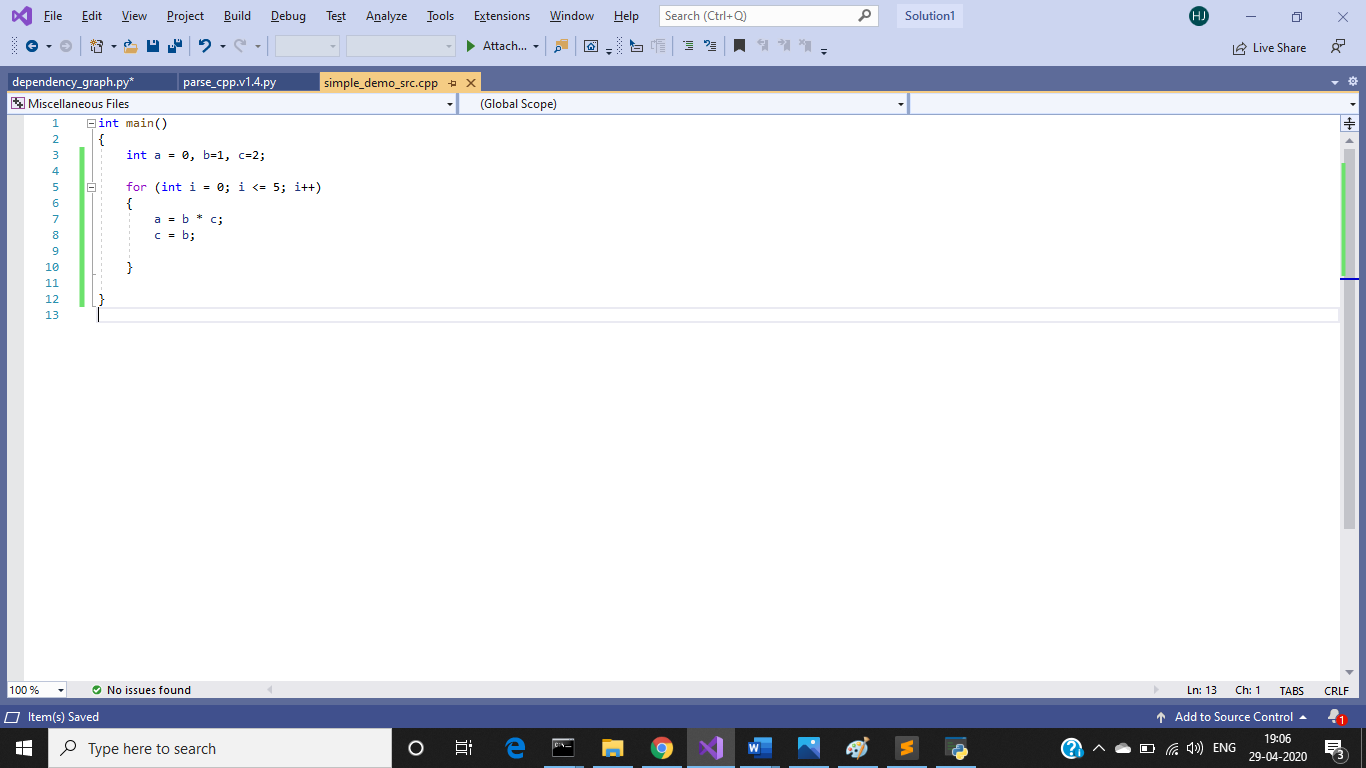
**1.**

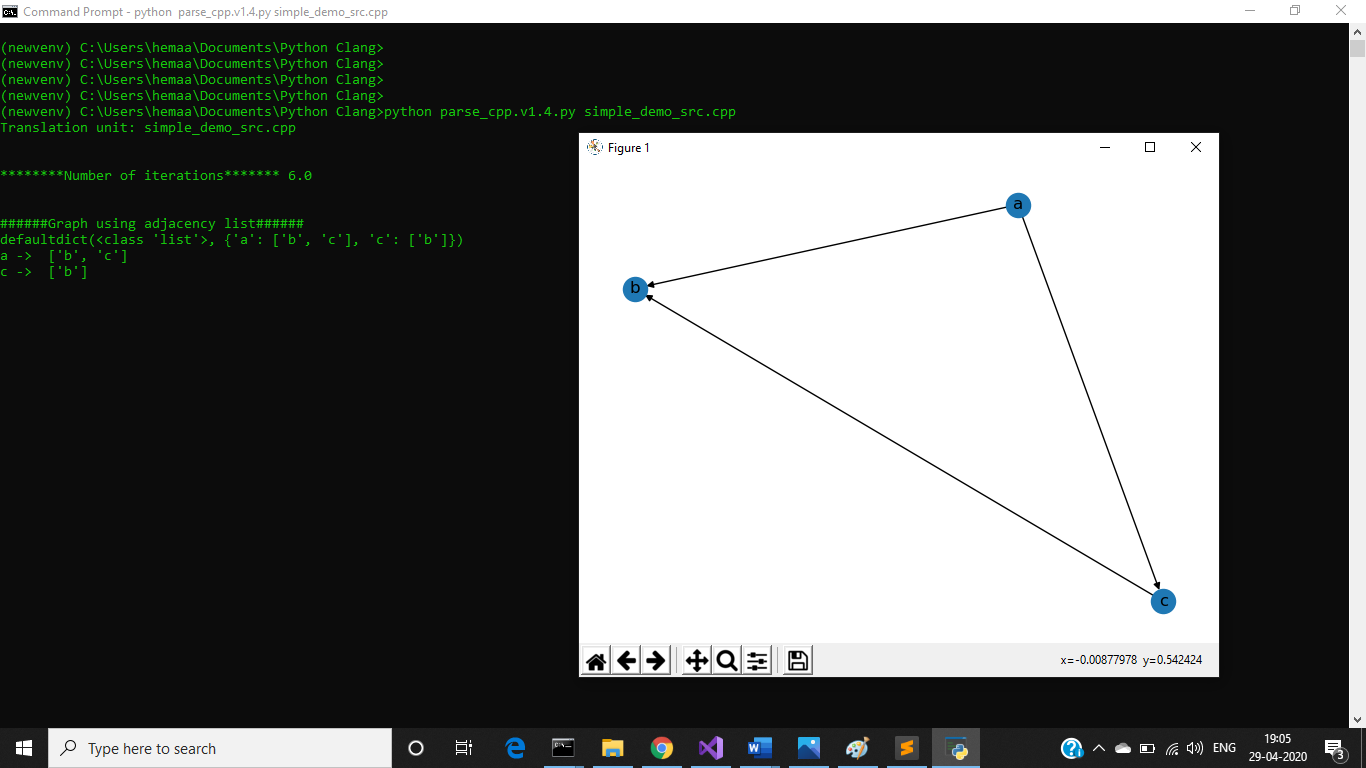




[If you see ‘a’ depends on ‘a’. Via adjacency list we are able to show that but networkx doesn’t have the feature to show a self-loop in the graph. There are some ways to show self-loops according to <https://stackoverflow.com/questions/22312334/how-to-show-cycles-in-networkx-graph-drawing>. But that doesn’t seem to work in my system. If you get a chance you can try that out in your system and see]

2.





References for graph:

1. <https://networkx.github.io/documentation/networkx-1.10/tutorial/tutorial.html>
2. <https://matplotlib.org/api/pyplot_api.html>

**Notes:**

This is how we can parse a cpp and find out the dependency. We can elaborate on this based upon the requirements.

As I mentioned earlier, there is very less documentation on libclang, may be at some point, new features would come up that makes the parsing easier and there could be a direct solution for analyzing the dependencies.