# C TO PYTHON PROGRAMMING LANGUAGE TRANSLATOR

[1]Onkar Apte, [2]Shubham Agre, [3]Rajendraprasad Adepu, [4]Mandar Ganjapurkar

[1,2,3]Undergraduate student,
[4]Assistant Professor,
[1,2,3,4]Department of Computer Engineering,
[1,2,3,4]K.C. College of Engineering and Management studies & Research, Thane, India.

***Abstract:*** C and Python are one of the two popular and robust programming languages still used today. These two languages have their own unique advantages, C is compiled whereas Python is interpreted. C follows imperative programming model and have limited built-in functions whereas Python is object-based language and have large library of built-in functions. C has harder syntax compared to Python as well. In this paper, we develop a C to Python programming language translator that converts C code into Python using Python as an intermediate language. This enables the direct, automatic translation instead of having to write Python program from scratch.

***Index Terms* – Compiler, Interpreter, Programming Language, Translator.**

## 1. INTRODUCTION

C and Python are one of the most commonly used programming languages in the world. They have their own features that developers like. Main reasons are Python is easy to learn, has simple syntax, and has large libraries whereas C is simple, and being compiled, has faster execution speeds. As we all know python isn't anywhere near C speed (with current compilers and interpreters) but worse than that is that what C is good at (bit-fiddling, integer math, tricks with blocks of memory) Python is very slow at, and what Python is good at you can't express in C directly. But due to vast libraries and easy to read code due to simple syntax of Python, Python is a better and a good choice in contemporary worlds. Therefore, a direct translation would be very useful.

A transition from one language to another is not easy. There are very few inter-programming language translators available and for C to Python there are basically none available. If programmers want to translate their C code and use the features that Python provides, they will have to write the whole code in python from scratch which will drain time as well as cost. Therefore, a mechanism that translates programs from C to Python automatically would be beneficial.

In this paper, we develop a simple C to Python translator that takes C code file as input and translates it and gives Python code file as output. The main objective of this work is that both input and output codes would give same exact output with same values stored in defined variables. We are keeping our translator open source, on GitHub, such that it discloses conversation steps and other people will gain more insight on working on our translator and can work on this in the future to improve the on the flaws that we have.

Our translator covers the basic, simple programming, comments, variable declarations, floating point, Boolean datatypes, functions, control loops, input (scanf) and output (printf) in C and translates it accordingly into Python. The translator reads the code as text file, iterates it once and identify each line's line type – more on that ahead, and then performs translation and gives output a text file which contains Python code which gives exact same output and has all variable values same as C-run program at the end of execution.

Attempts have been made to translate C to Python. None of the attempts that we are aware of have been able to solve this complex problem completely, but a few useful techniques and methods by which this problem can be solved can be deduced from these attempts. In this project we will try to create a scaled-down C to Python translator.

The structure of the paper ahead will be as follows-, section 2 gives the insight on syntactical differences in C and Python and explains language processors of C which is a compiler and for Python which is an interpreter. Our system design is illustrated in the next section, section 3 and section 4 demonstrates the actual implementation of the translator and shows the working examples, section 5 discusses the future scope of this project and what further advancements can be done and the last section i.e., section 6 is the conclusion.

## 2. SYNTACTICAL DIFFERENCES BETWEEN C AND PYTHON AND LANGUAGE PROCESSING

C is a general-purpose, procedural computer programming language. By design, C provides constructs that map efficiently to typical machine instructions. It has found the lasting use in applications previously coded in assembly language. Such applications include whole OS (operating systems) and various application software as well.

Python is an interpreted-high-level general-purpose programming language. It is mainly designed such that it emphasizes on code readability with its notable use of significant indentation i.e., tab spacing. We can't say that Python is written in some programming language since Python as a language is just a set of rules (like syntax rules, descriptions of standard functionality). So, we might say that it is written in English however, mentioned rules can be implemented in some programming language.

There are many other implementations of Python, CPython (Python written in C), Jython (Python written in Java), IronPython (Python written in C++ and .NET) and a funny but real and interesting- PyPy (Python written in Python).

Here, next we show the syntactical differences in C and Python that is considered in this work. We also talk in brief about language processing relevant to C and Python.

## 2.1 C and Python Basic Syntax

Here we compare C and Python in terms of syntax to get to know of what our translator is expected to perform. Table 2.1 compares the syntactical differences in terms of comments, variable declaration, input-output, functions, control statement.

Table 2.1: C vs. Python Syntax

| | ***C*** | ***Python*** |
|---|---|---|
| **Single line comment** | // This is a comment | #This is a comment |
| **Multi line comment** | /* This is<br>        Multi- line comment */ | ‘’’ This is<br>        Multi- line comment ‘’’ |
| **If-else statement** | if(condition)<br>{<br>            statements;<br>}<br>else<br>{<br>            statements;<br>} | if(condition):<br>        statements<br>else:<br>        statements |
| **Nested if-else statement** | if(condition)<br>{<br>        statements;<br>}<br>else if(condition)<br>{<br>        statements;<br>        if(condition)<br>        {<br>                statements;<br>        }<br>}<br>else<br>{<br>        statements;<br>} | if(condition):<br>        statements<br>elif(condition):<br>        statements<br>        if(condition):<br>                statements<br>else:<br>        statements |
| **While statement** | while(condition){<br>        //body of loop<br>        statements;<br>} | while(condition):<br>        #body of loop<br>        statements |
| **For statement** | for(initialization; condition; iteration)<br>{<br>        //body of loop<br>        statements;<br>} | for n in range(start, end, step):<br>        #body of loop<br>        statements |
| **Function declaration statement** | datatype FunctionName(Arg1, Arg2)<br>{<br>        //body of function<br>        statements;<br>} | def FunctionName(Arg1, Arg2):<br>        #body of function<br>        statements |
| **Variable declaration statement** | int a,b=0,c=10,d;<br>float x,y,z; | No need to declare variables beforehand, you can directly initialize. |
| **Print statement** | printf(“area of circle is %d”,a); | print(f’area of circle is {a}’) |
| **Scan statement** | scanf(“%d”,&a); | a = int(input()) |
| **Logical statement** | a && b<br>a \|\| b<br>!(a) | a and b<br>a or b<br>not a |

## 2.2 Language processors: Compiler (C) and Interpreter (Python)

Language processing in C is achieved by Compiler whereas achieved by an interpreter for Python.

### 2.2.1 Compiler

A compiler is a program that can read a program in one language (source language) and translate it into an equivalent program in other language (target language). The compiler has two main parts. One is analysis and other is synthesis. In analysis part, compiler divides the source program code into pieces, then applies the grammatical structure to them and generates an intermediate representation of the source code. If the syntax of the source code has syntactical errors, then it generates an informative message to the user. It also collects information about the source code and stores them in a table called symbol table. The synthesis part uses that table and intermediate representation to generate the target program.

### 2.2.2 Interpreter

An interpreter is a language processor which directly executes the source program on user inputs. The task of an interpreter is almost same as compiler, but the interpreter works in a different way. The interpreter takes a single line of code as input at a time and executes that line. It will terminate the execution as soon as it finds an error. Memory requirement is less because no object code (intermediate code from source language) is created.

The machine language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. An interpreter, however, can usually give better error diagnostic than compiler, because it executes the source program statement by statement.

## 2.3 C and Python Language Processor

### 2.3.1 C

In C, the source code is sent to preprocessor first. The preprocessor is responsible to convert preprocessor directives into their respective values. It generates an expanded source code. This expanded source code is sent to compiler which compiles the code and converts it into assembly code. The assembly code is sent to assembler which assembles the code and converts it into object code. A file, '*filename.obj*' is generated. This object code is sent to linker, which links it to the library such as header files and then it is converted into executable code. A new executable file, '*filename.exe'* is generated. Finally, the executable code is sent to loader which loads it into memory and then it is executed.

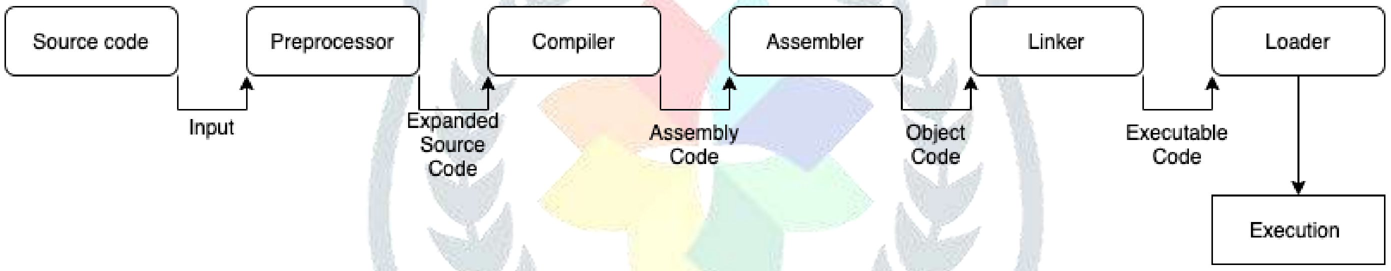


Fig 1: C Code Language Processing

### 2.3.2 Python

Python takes four steps. Those are lexing, parsing, compiling, and interpreting. In lexing step, the line of code is broken into tokens. The parsing phase uses those tokens and generates an Abstract Syntax Tree (AST). This Abstract Syntax Tree (AST) is a structure showing the relationships of the tokens. The third step- compiling, takes the AST and transforms it into one or more code objects. Finally in the last step- interpreting, the interpreter takes each code object and executes the code it represents. In this way, Python interpreter works by executing the code line-by-line at a time.

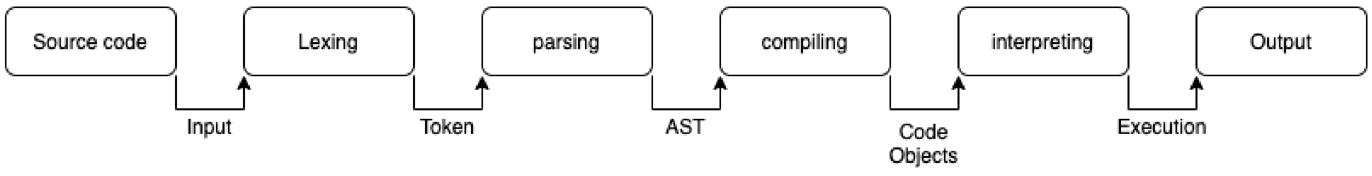


Fig 2: Python Code Language Processing

## 3. System Design of the C to Python Programming Language Translator

In computing, a translator is a program which takes a file written in source language and transforms it into another language i.e., target language without losing the functional or logical structure. Both the programs, i.e., the source code and target code provide same output. They are logically and structurally equivalent. Following ahead, we explain the design of our translator and its working.

### 3.1 System Architecture

The C to python programming language translator we created, reads the C program source code from a file, translates it into Python code and writes it into a file. The extreme simplified working of this is as follows-

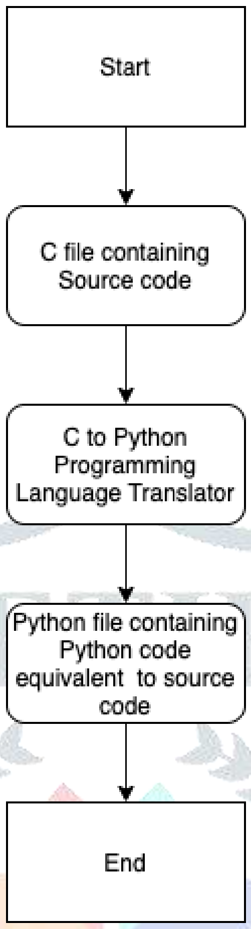

Start
C file containing Source code
C to Python Programming Language Translator
Python file containing Python code equivalent to source code
End

Fig 3: Working of Translator (Superficial view)

This is just a superficial view of our system to get just the overview of it. It is explained in detail in 3.2.

#### 3.1.1 Language used to create the C to Python Programming Language Translator

The C to Python programming Language Translator itself is written in the Python Language. The whole code of this translator is written in the Python Language with version Python 3.8.5

#### 3.1.2 Selecting Python as a language of choice to write the Translator

The main reason for using python is the facilities that Python provides and simplicity of the code writing. Python has many in-built ready-to-use functions useful for string manipulation, e.g. split(), made the task of creating translator easy. The translator can be written in any other language without any issues, with same logic. But the only thing will be different is actual code writing, which would be tedious in say C or Java compared to Python but won't be impossible. So, because of that we selected Python as a language of choice to write the Translator.

### 3.2 Translation Process

The translation basically happens in three phases. Firstly, the code in C is turned into ideal syntax. In the second phase, the translator parses the file line-by-line and determines its line-type, which we have arbitrarily selected. In the third phase, each line is translated by referring to line-type array created in second step and then fully translated Python code file is given as an output.

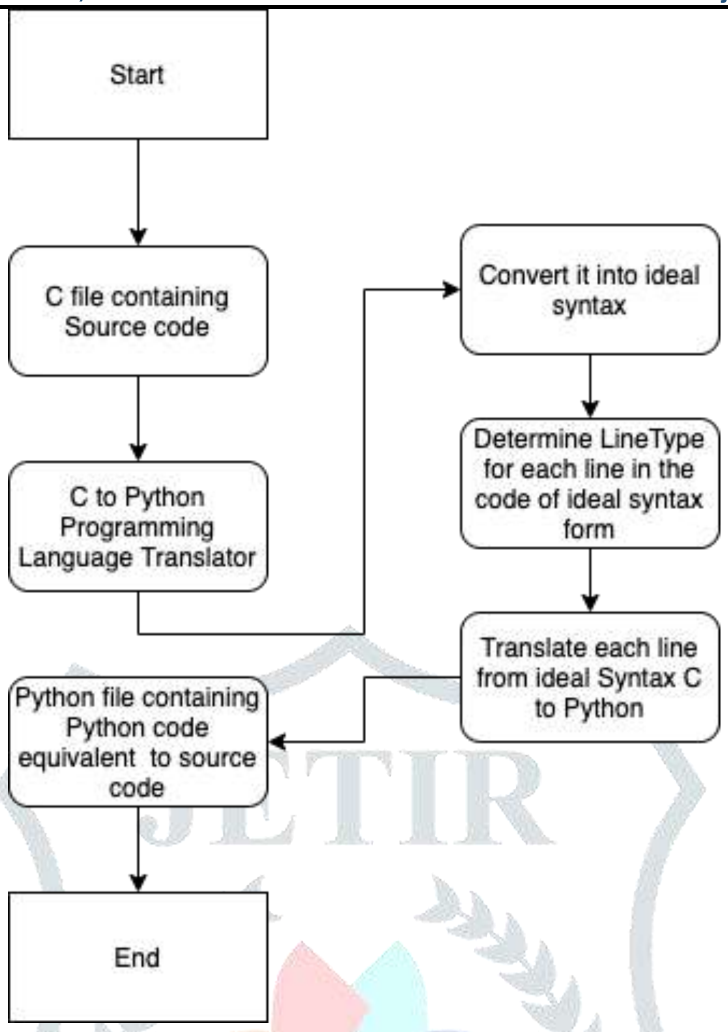

Start
C file containing Source code
C to Python Programming Language Translator
Convert it into ideal syntax
Determine LineType for each line in the code of ideal syntax form
Translate each line from ideal Syntax C to Python
Python file containing Python code equivalent to source code
End

Fig 4: Working of Translator (standard view)

The ideal syntax is just arbitrary rules we set to write C program in. Every statement in C can be written in many ways, with extra white spaces and brackets on previous line or in the same line or in the next line, so we first standardize the C code by converting the input program with randomly placed spaces, brackets and empty lines and turn it into ideal syntax with standard rules.

We first decided our ideal syntax for C source code input file, and at the start of implementation we assumed that input file will be already in ideal syntax format and started working from there. So, we had to implement a way to translate the C source code file in ideal syntax to Python target code which would give exact same output at source file and also all the variable values would be same at the end of execution.

### 3.2.1 Need of ideal syntax

In C, every line or a block of code can be written in different ways, as said above. Following is an example-

| | | |
|---|---|---|
| for ( i = 0 ; i<= 5; i++ )<br>printf( “%d”, i); | for (int i = 0;i<=5;i++ )<br>{printf(“%d”,i);} | for(i=0;i<=5;i++)<br>{<br>print(“%d”,a);<br>} |

All these above examples would provide the same output, but it would be difficult to consider all of these cases at the time of translation, so we added a step before translating i.e., to convert the code into a standard, fixed format and we call it the ideal syntax. It is made of the arbitrary chosen rules we have set up.

### 3.2.2 The ideal syntax

Table 3.1: Ideal Syntax

| | Ideal Syntax | Remarks |
|---|---|---|
| **Single line comment** | // This is a comment | No remarks |
| **Multi line comment** | /* This is<br>Multi- line comment */ | No remarks |
| **If-else statement** | if(condition)<br>{<br>statements;<br>}<br>else<br>{<br>statements;<br>} | No extra spaces anywhere. No blank lines. No white spaces at start of any line. Curly brackets ({, } ) on new line always. Curly brackets ({, } ) are must, even if it contains single statement. |
| **Nested if-else statement** | if(condition)<br>{<br>statements;<br>}<br>else if(condition)<br>{<br>statements;<br>if(condition)<br>{<br>statements;<br>}<br>}<br>else<br>{<br>statements;<br>} | No extra spaces anywhere. No blank lines. No white spaces at start of any line. Curly brackets ({, } ) on new line always. Curly brackets ({, } ) are must, even if it contains single statement. |
| **While statement** | while(condition)<br>{<br>//body of loop<br>statements;<br>} | No extra spaces anywhere. No blank lines. No white spaces at start of any line. Curly brackets ({, } ) on new line always. Curly brackets ({, } ) are must, even if it contains single statement. |
| **For statement** | for(initialization;condition;iteration)<br>{<br>//body of loop<br>statements;<br>} | No extra spaces anywhere. No blank lines. No white spaces at start of any line. Curly brackets ({, } ) on new line always. No spaces between loop arguments and semi colons (;). Curly brackets ({, } ) are must, even if it contains single statement. |
| **Function declaration statement** | datatype FunctionName(Arg1,Arg2)<br>{<br>//body of function<br>statements;<br>} | No extra spaces anywhere. No blank lines. No white spaces at start of any line. Curly brackets ({, } ) on new line always. Single space between datatype and function name. No spaces in arguments. |
| **Variable declaration statement** | int a,b,c,d;<br>b=0<br>c=10<br>float x,y,z; | No initialization in the same line as declaration, do it in the next line if you wish. No spaces between variables and comma. |
| **Print statement** | printf("area of circle is %d",a); | No extra blank spaces anywhere. |
| **Scan statement** | scanf("%d",&a); | No extra blank spaces anywhere. |
| **Assignment operations** | a=b+C<br>a[n]=b[i]+3 | No extra blank spaces anywhere. |
| **Logical operations** | x!=0<br>value1&&value2 | No extra blank spaces anywhere. |

So, this is the format we decided to be as our ideal syntax. So, we firstly convert any input file into this ideal syntax format and then perform the translation on that. Next step is to identify the LineType of each line.

### 3.2.3 LineType

To perform the translation of each line, we first parse the C file in ideal syntax form and determine the type of each line. According to the type of the line we perform C to Python translation of it.

Basically, we create an array of length equal to the number of lines in the C ideal syntax file and represent each array element with a number which corresponds to its LineType. We have set the numbers as follows:

0. include statements
1. function declaration
2. variable declaration
3. input line (scanf)
4. output line (printf)
5. if statement
6. else if statement
7. else statement
8. for statement
9. while statement
10. arithmetic operations
11. do while statement
12. switch case
13. single line comment
14. multi-line comment

-1. For everything rest.

So, for example if your 10th line would be a print statement then LineType[9] will be equal to 4. (9th index because 1st line will correspond to 0th index and so on)

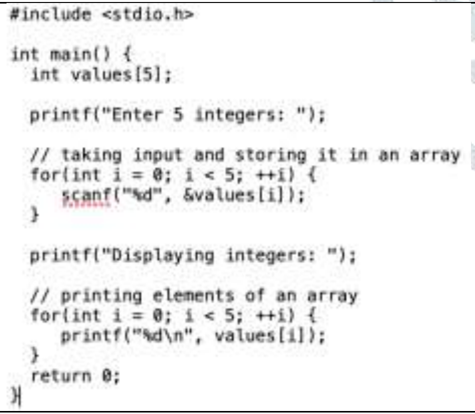
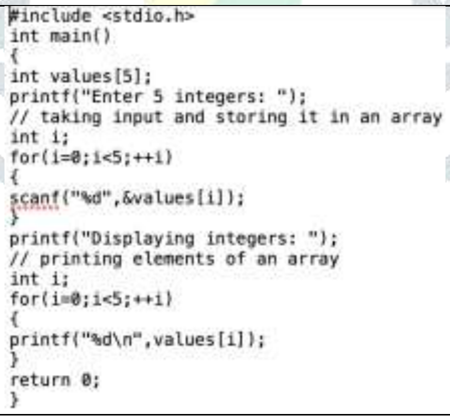
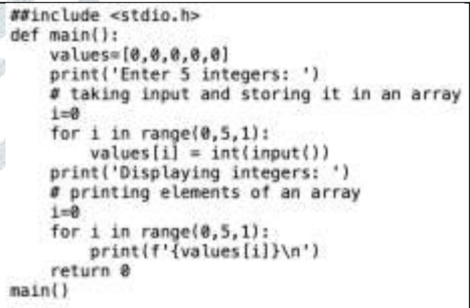
### 3.2.4 Translation Final Step

Till this point, we took a C source code file as input, converted into an intermediate format called as ideal syntax and then determined LineType for every line in the ideal syntax source code. The final step is translating each line.

We created translation functions for each type of line, and we call that function referencing the LineType array, give proper indentation and then the final output is ready i.e., Python code file which gives exact same output as Source C code file having same logic and structure.

### 3.2 Translation Example

Table 3.2: Working flow of translator – step by step output

| | | |
|---|---|---|
|  |  |  |
| Fig 5: Input file (C language Source code file) | Fig 6: Intermediate state (C code in ideal syntax format) | Fig 7: Final output (Python equivalent code to the input file) |

```
#include <stdio.h>

int main() {
  int values[5];

  printf("Enter 5 integers: ");

  // taking input and storing it in an array
  for(int i = 0; i < 5; ++i) {
     scanf("%d", &values[i]);
  }

  printf("Displaying integers: ");

  // printing elements of an array
  for(int i = 0; i < 5; ++i) {
     printf("%d\n", values[i]);
  }
  return 0;
}
```

```
#include <stdio.h>
int main()
{
int values[5];
printf("Enter 5 integers: ");
// taking input and storing it in an array
int i;
for(i=0;i<5;++i)
{
scanf("%d",&values[i]);
}
printf("Displaying integers: ");
// printing elements of an array
int i;
for(i=0;i<5;++i)
{
printf("%d\n",values[i]);
}
return 0;
}
```

```
##include <stdio.h>
def main():
    values=[0,0,0,0,0]
    print('Enter 5 integers: ')
    # taking input and storing it in an array
    i=0
    for i in range(0,5,1):
        values[i] = int(input())
    print('Displaying integers: ')
    # printing elements of an array
    i=0
    for i in range(0,5,1):
        print(f'{values[i]}\n')
    return 0
main()
```

This is how the translator works. As shown in Fig 4, as well as table above, Table 3.2, First C input source file is converted into intermediate ideal syntax format and then that ideal syntax formatted C file gets translated into equivalent Python code.

## 4. Running the Translator

C to Python Programming Language Translator is a Python executable file. It has been tested on Windows and Mac Operating Systems. To perform the translation, create a text file named *'input.txt'* in the same directory as executable Python file of translator. Paste your C code you need to translate into the text file we created and then execute the Python file. After execution '*output.txt*' file will appear in the same directory as of Python executable Translator file and C Source code containing text file.

Further in this section, we demonstrate the output of the translator. The section also discusses few challenges faced during translation to get correct resultant output.

### 4.1 Challenges

The translation for most of the stuff was just text manipulations but there were some tricky situations. Translating for loop was tricky, we had to consider multiple possibilities of having a constant or variables in its parameters, also increment-decrement had to be seen. Translating print statements were tricky as well since they contain many things, identifiers (e.g., %d) or slash characters (e.g., \n) and string itself, so we had to replace the identifiers with proper variables in python print statements as you can see in further examples.

When there is only one statement after a if, for, while statements, there is no need to put curly brackets. With or without them, C program would work fine, but in our ideal syntax, curly brackets were a must, so we had to identify when which loop ends and add curly brackets accordingly, it was hard in cases of nested statements, but we were able to do it successfully.

Implementation of arrays was somewhat tricky but more than it was implementation of multidimensional arrays. For a long time, we were unable to come up with a solution to overcome it but finally we solved it and used a recursive function to create a multidimensional array in Python while translation and now our translator can successfully translate even N-dimensional arrays and perform all array operations with no issues.

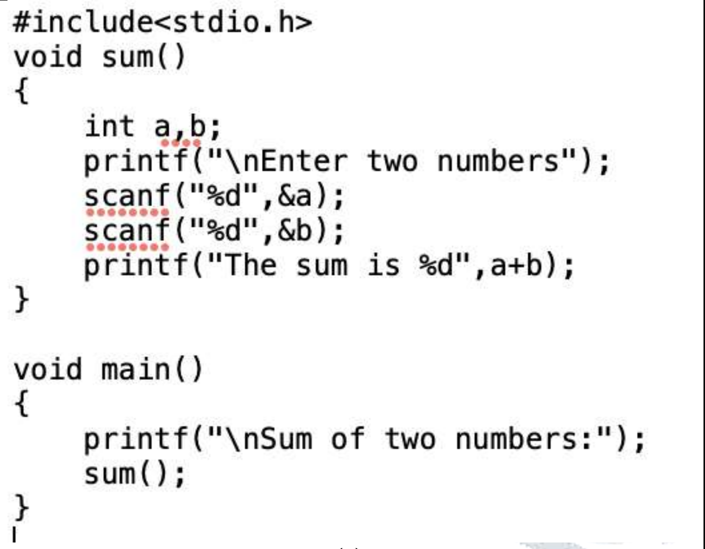
Pointers were tough to implement as well and partly that's due to how Python works at core. We were unable to implement it, more on that in section 5.

### 4.2 Translation Examples

The sections show different translation examples:

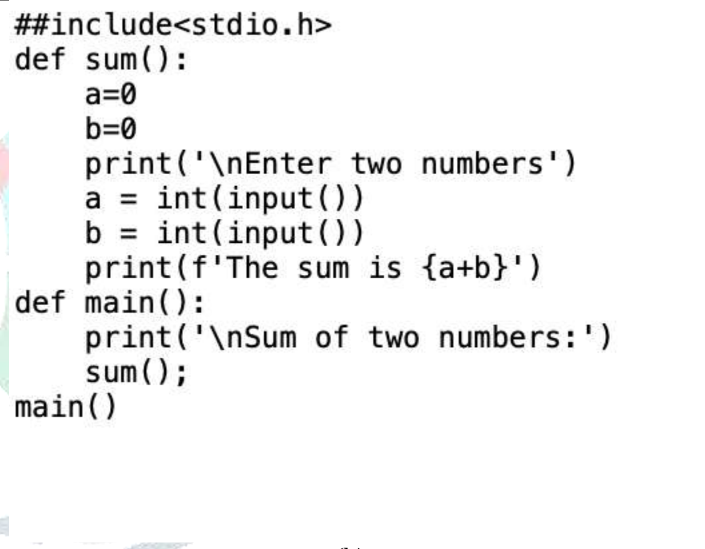
#### 4.2.1 Functions

Table 4.1: Functions example (a) C Input Code (b) Python Output Code



```c
#include<stdio.h>
void sum()
{
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d",&a);
    scanf("%d",&b);
    printf("The sum is %d",a+b);
}

void main()
{
    printf("\nSum of two numbers:");
    sum();
}
```

(a)

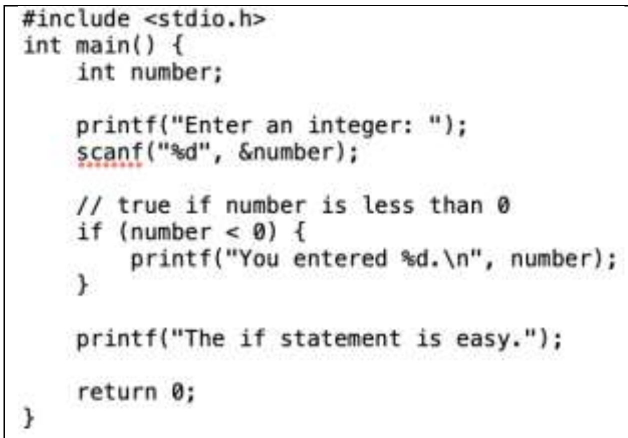


```python
##include<stdio.h>
def sum():
    a=0
    b=0
    print('\nEnter two numbers')
    a = int(input())
    b = int(input())
    print(f'The sum is {a+b}')
def main():
    print('\nSum of two numbers:')
    sum();
main()
```

(b)

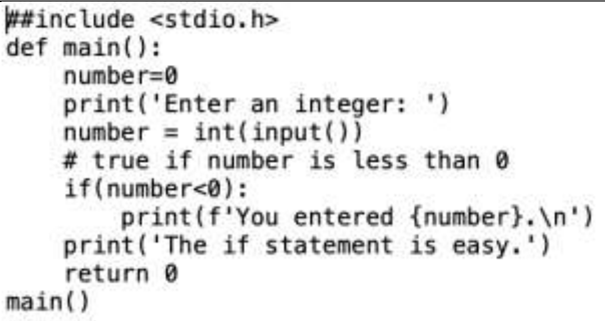
### 4.2.2 if statement

Table 4.2: if statement example (a) C Input Code (b) Python Output Code



```
#include <stdio.h>
int main() {
    int number;

    printf("Enter an integer: ");
    scanf("%d", &number);

    // true if number is less than 0
    if (number < 0) {
        printf("You entered %d.\n", number);
    }

    printf("The if statement is easy.");

    return 0;
}
```

(a)

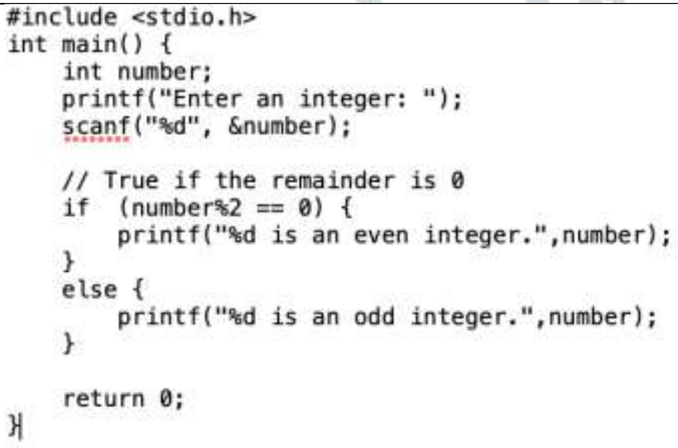


```
##include <stdio.h>
def main():
    number=0
    print('Enter an integer: ')
    number = int(input())
    # true if number is less than 0
    if(number<0):
        print(f'You entered {number}.\n')
    print('The if statement is easy.')
    return 0
main()
```

(b)

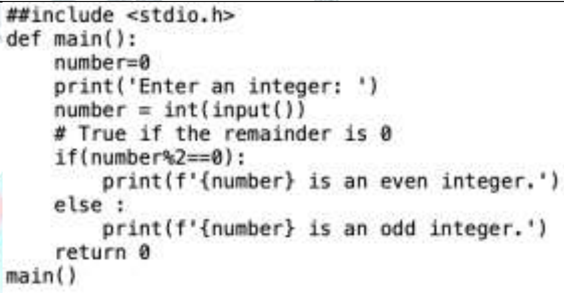
### 4.2.3 if – else statements

Table 4.3: if-else statements example (a) C Input Code (b) Python Output Code



```
#include <stdio.h>
int main() {
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);

    // True if the remainder is 0
    if  (number%2 == 0) {
        printf("%d is an even integer.",number);
    }
    else {
        printf("%d is an odd integer.",number);
    }

    return 0;
}
```

(a)

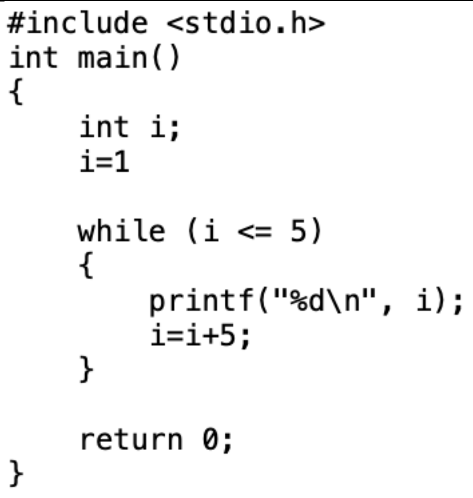


```
##include <stdio.h>
def main():
    number=0
    print('Enter an integer: ')
    number = int(input())
    # True if the remainder is 0
    if(number%2==0):
        print(f'{number} is an even integer.')
    else :
        print(f'{number} is an odd integer.')
    return 0
main()
```

(b)

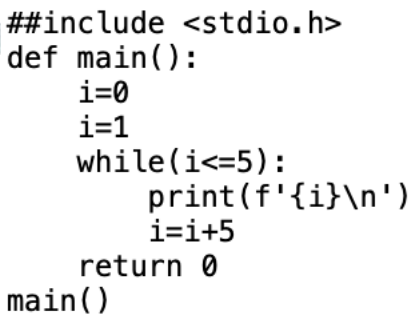
### 4.2.4 while statement

Table 4.4: While statement example (a) C Input Code (b) Python Output Code



```
#include <stdio.h>
int main()
{
    int i;
    i=1

    while (i <= 5)
    {
        printf("%d\n", i);
        i=i+5;
    }

    return 0;
}
```

(a)

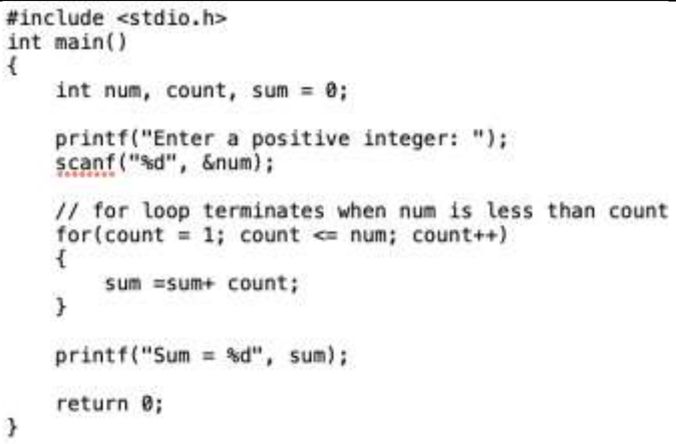


```
##include <stdio.h>
def main():
    i=0
    i=1
    while(i<=5):
        print(f'{i}\n')
        i=i+5
    return 0
main()
```

(b)

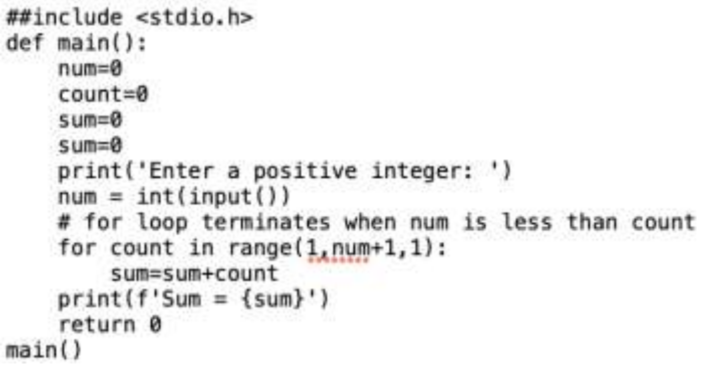
### 4.2.5 for statement

Table 4.5: for statement example (a) C Input Code (b) Python Output Code



```
#include <stdio.h>
int main()
{
    int num, count, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    // for loop terminates when num is less than count
    for(count = 1; count <= num; count++)
    {
        sum =sum+ count;
    }

    printf("Sum = %d", sum);

    return 0;
}
```

(a)

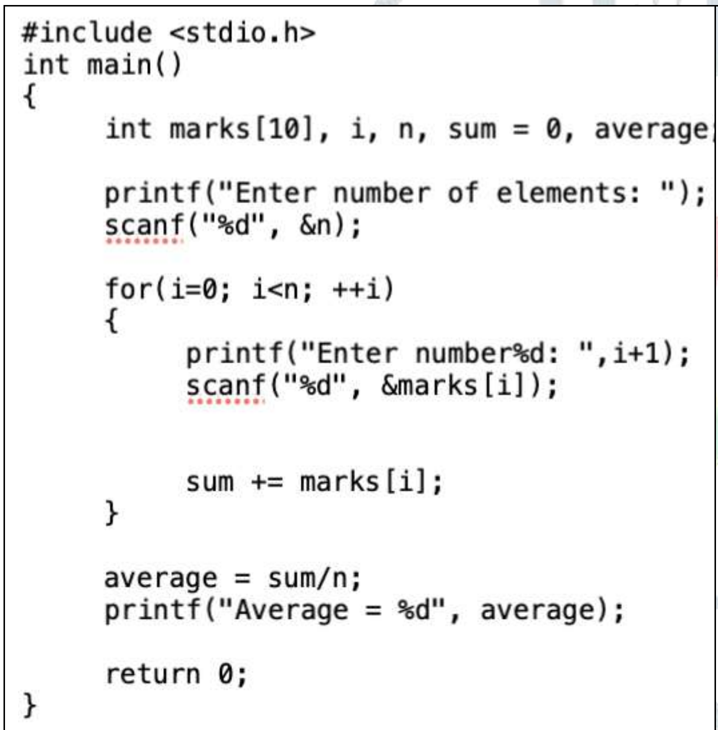


```
##include <stdio.h>
def main():
    num=0
    count=0
    sum=0
    sum=0
    print('Enter a positive integer: ')
    num = int(input())
    # for loop terminates when num is less than count
    for count in range(1,num+1,1):
        sum=sum+count
    print(f'Sum = {sum}')
    return 0
main()
```

(b)

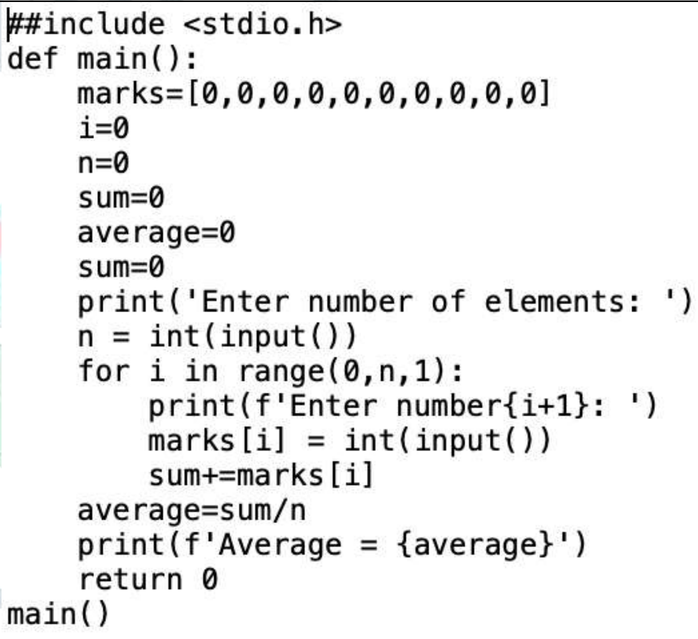
### 4.2.6 One-dimension array

Table 4.6: 1-D array example (a) C Input Code (b) Python Output Code



```
#include <stdio.h>
int main()
{
    int marks[10], i, n, sum = 0, average;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    for(i=0; i<n; ++i)
    {
        printf("Enter number%d: ",i+1);
        scanf("%d", &marks[i]);


        sum += marks[i];
    }

    average = sum/n;
    printf("Average = %d", average);

    return 0;
}
```

(a)

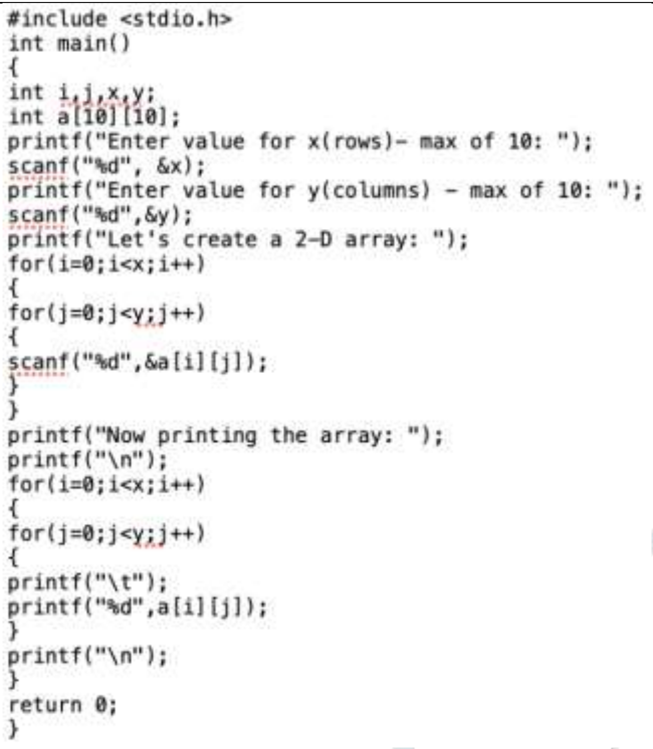


```
##include <stdio.h>
def main():
    marks=[0,0,0,0,0,0,0,0,0,0]
    i=0
    n=0
    sum=0
    average=0
    sum=0
    print('Enter number of elements: ')
    n = int(input())
    for i in range(0,n,1):
        print(f'Enter number{i+1}: ')
        marks[i] = int(input())
        sum+=marks[i]
    average=sum/n
    print(f'Average = {average}')
    return 0
main()
```

(b)

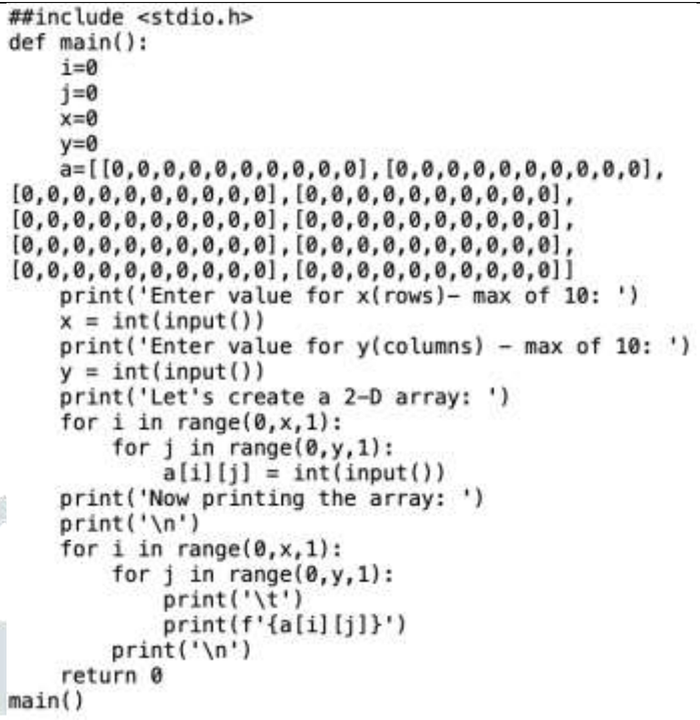
**4.2.7 Nested iterative loops with Multi-dimension array**

Table 4.7: Nested iterative loops with Multi-dimension array example (a) C Input Code (b) Python Output Code



```
#include <stdio.h>
int main()
{
int i,j,x,y;
int a[10][10];
printf("Enter value for x(rows)- max of 10: ");
scanf("%d", &x);
printf("Enter value for y(columns) - max of 10: ");
scanf("%d",&y);
printf("Let's create a 2-D array: ");
for(i=0;i<x;i++)
{
for(j=0;j<y;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("Now printing the array: ");
printf("\n");
for(i=0;i<x;i++)
{
for(j=0;j<y;j++)
{
printf("\t");
printf("%d",a[i][j]);
}
printf("\n");
}
return 0;
}
```

(a)

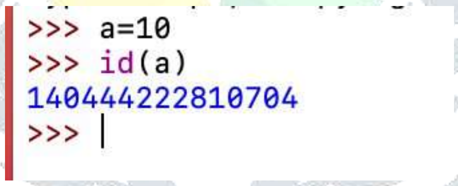


```
##include <stdio.h>
def main():
    i=0
    j=0
    x=0
    y=0
    a=[[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0]]
    print('Enter value for x(rows)- max of 10: ')
    x = int(input())
    print('Enter value for y(columns) - max of 10: ')
    y = int(input())
    print('Let's create a 2-D array: ')
    for i in range(0,x,1):
        for j in range(0,y,1):
            a[i][j] = int(input())
    print('Now printing the array: ')
    print('\n')
    for i in range(0,x,1):
        for j in range(0,y,1):
            print('\t')
            print(f'{a[i][j]}')
        print('\n')
    return 0
main()
```

(b)

## 5. Limitations

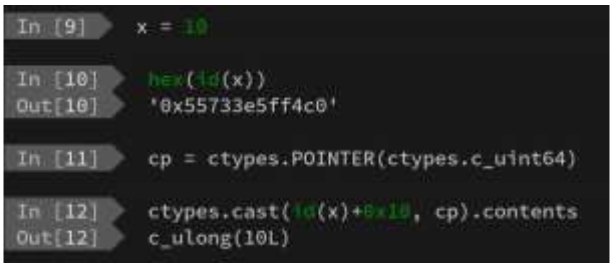
We thought of implementing pointer in Python. Since there is a function in Python called id(), which gives the memory location of any object/variable, we tried to work with that.



```
>>> a=10
>>> id(a)
140444222810704
>>> |
```

Fig 8: Example of working of id() function

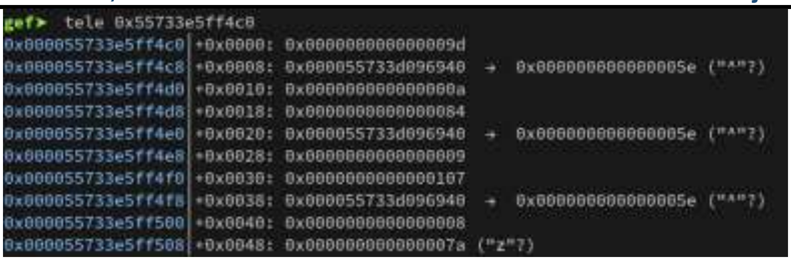
id() gives you an address of the python object. In the above example, a is an int object and its memory location is *140444222810704* i.e., at memory location *140444222810704*, value 10 is stored in the computer. So, at offset 0x10 we should be able to extract the int as shown below.



```
In [9]    x = 10

In [10]   hex(id(x))
Out[10]   '0x55733e5ff4c0'

In [11]   cp = ctypes.POINTER(ctypes.c_uint64)

In [12]   ctypes.cast(id(x)+0x10, cp).contents
Out[12]   c_ulong(10L)
```

Fig 9: id() in hex at offset 0x10

But this is just a hack. When we used it in the project it broke the whole translator and we had to debug a lot for other object types manually. Here's how it looks in the memory wrote that address



```
gef>  tele 0x55733e5ff4c8
0x000055733e5ff4c0│+0x0000: 0x000000000000009d
0x000055733e5ff4c8│+0x0008: 0x000055733d096940  →  0x000000000000005e ("^"?)
0x000055733e5ff4d0│+0x0010: 0x000000000000000a
0x000055733e5ff4d8│+0x0018: 0x0000000000000084
0x000055733e5ff4e0│+0x0020: 0x000055733d096940  →  0x000000000000005e ("^"?)
0x000055733e5ff4e8│+0x0028: 0x0000000000000009
0x000055733e5ff4f0│+0x0030: 0x0000000000000107
0x000055733e5ff4f8│+0x0038: 0x000055733d096940  →  0x000000000000005e ("^"?)
0x000055733e5ff500│+0x0040: 0x0000000000000008
0x000055733e5ff508│+0x0048: 0x000000000000007a ("z"?)
```

Fig 10: Memory and values stored in it

So, in the end, we were unable to find a perfect way to implement pointers in Python. We think it is somewhat impossible, but if we changed something in the Python's end, at the assembly language level, how Python uses memory, we may be able to implement it but that's not feasible and not very useful as well. So, we were not able to implement translation of pointers in C to Python. Implementation of structures is also possible, i.e., translating structures of C program to Python using ctypes. Due to time constraint on this project, we were unable to implement it, but it is possible and feasible as well.

## 6. Conclusion

This paper presents a C to Python Programming Language Translator that converts C code to Python code using a translator coded in Python. The translator can successfully translate function declaration, variable declaration, single line comments, multi-line comments, iterative loops, control statements, logical operations, arithmetic operations, multi-dimensional arrays.

After designing, implementing, and testing the translator, we found that it successfully converts the syntax of C programs to equivalent Python programs. For future work we would like to implement structure which we were unable to due to time constraint on the project and see if we can implement pointer functionality.

Our full code is open-source and available at GitHub [online], https://github.com/FDGod99/C-to-Python-Translator

## References

**[1]** Wikipedia [online], https://en.wikipedia.org/wiki/C_(programming_language)

**[2]** Wikipedia [online], https://en.wikipedia.org/wiki/Python_(programming_language)

**[3]** C documentation [online], https://devdocs.io/c-algorithms/

**[4]** Python documentation [online], https://docs.python.org/3/

**[5]** Python implementations [online], https://www.python.org/download/alternatives/

**[6]** Brian W. Kernighan, Dennis M. Richie, "The C Programming Language 2nd Edition", Pearson.

**[7]** Richard L. Halterman, "Learning to program with Python", Richard L. Halterman

**[8]** A foreign function library for Python [online], https://docs.python.org/3/library/ctypes.html

**[9]** P. J. Brown, "Writing interactive compilers and interpreters", Wiley

**[10]** George, D., Girase, P., Gupta, M., Gupta, P., & Sharma, A., (2010) "Programming Language Interconversion". International Journal of Computer Applications, vol. 1, no. 20

**[11]** James B. McAtamney, **"**C-to-java programming language translator (2013)"

**[12]** James Adam Cataldo, Long Bill Huynh, Stanley T. Jefferson, "Programming language translator and enabling translation of machine-centric commands for controlling instrument (2009)"

**[13]** Tom Simonsen, "Translating Python to C++ for palmtop software development (2005)"

**[14]** Eman J. Coco, Hadeel A. Osman, "JPT: A Simple Java-Python Translator (2018)"

**[15]** A. A. Terekhov, C. Verhoef, "The realities of language conversions (2000)"