

**1.0 Release Candidate 1**

A stack-based Programming language implemented in Python.

***Sayak Brahmachari***

Contents

[About struixLang 1](#_Toc484791478)

[Use Cases 2](#_Toc484791479)

[Usage 3](#_Toc484791480)

[Data Model 4](#_Toc484791481)

[List of Primitive Words in this Implementation 5](#_Toc484791482)

[struixLang 101 7](#_Toc484791483)

[Execution Process 7](#_Toc484791484)

[Syntax and Other Stuff 7](#_Toc484791485)

[Basics of struixLang 8](#_Toc484791486)

[Input and Output 8](#_Toc484791487)

[Variables and Constants 10](#_Toc484791488)

[Mathematical Operations 11](#_Toc484791489)

[Mathematical Operators: 12](#_Toc484791490)

[Stack Operations 13](#_Toc484791491)

[DUP – Duplicate ToS 13](#_Toc484791492)

[DROP – Remove ToS 13](#_Toc484791493)

[SWAP – Swap ToS and 2oS 13](#_Toc484791494)

[OVER – Copy 20S on top of ToS 14](#_Toc484791495)

[ROT – Move 3oS on top of ToS 14](#_Toc484791496)

[Comments 14](#_Toc484791497)

[List 15](#_Toc484791498)

[Nested Lists 15](#_Toc484791499)

[LEN or LENGTH – Find length of List 15](#_Toc484791500)

[ITEM – To access an element from the List 15](#_Toc484791501)

[Executable Lists 16](#_Toc484791502)

[Conditional Statements 17](#_Toc484791503)

[IFTRUE 17](#_Toc484791504)

[IFFALSE 17](#_Toc484791505)

[IFELSE 17](#_Toc484791506)

[Looping Techniques 18](#_Toc484791507)

[TIMES 18](#_Toc484791508)

[WHILE 18](#_Toc484791509)

[DOWHILE 18](#_Toc484791510)

# About struixLang

A stack-based case-insensitive programming language implemented in Python3.

**struixLang** implements a **stack**, which is a list of objects which the program operates on.

Also, a **dictionary** is present, containing **words** *(functions/subroutines)* which may be executed in a program.

Several primitive *(read: built-in)* words are pre-defined and mechanisms to define new *user-defined* words within struixLang itself are in place.

INPUT

OUTPUT

Stack = Struix

# Use Cases

The most potential use case for struixLang is as an **embedded domain specific scripting language**.

Being so compact, struixLang can be easily adapted to the specifics of the host language and of the intended task. Not to mention its inherent simplicity should ensure acceptable performance even on top of another interpreted language (like this implementation).

# Usage

To run the default shell for struixLang, run the repl.py file under Python 3.

The interpreter can also be imported from within other Python programs.

import struixTerp

However, the interpreter by itself does not form the language. To import the primitives, do:

import struixPrimitives

Then create a new instance of the interpreter:

terp = struixTerp.Terp()

Add the primitives to it:

struixPrimitives.AddWords(terp)

And give the interpreter a string of struixLang code to run:

terp.run("""

var a

a = 10

"Hello, World!"

a fetch

[ print ] 2 times

""")

However, some potentially dangerous operations can be enabled by passing True to struixPrimitives.AddWords():

struixPrimitives.AddWords(terp, True)

# Data Model

struixLang supports the following data types:

* Integers,
* Floats,
* Strings,
* Boolean,
* Lists, and
* Words.

However, the current implementation can utilize words which put or use values of any type supported in Python 3.

Also, note that as struixLang is a *Homoiconic Language*, it can treat code as data and data as code, hence **words** are included in the list above.

# List of Primitive Words in this Implementation

1. PRINT – Pops and displays the last item put in the stack.
2. INPUT – Accepts input from the user.
3. PSTACK – Displays all items of the stack in Last-In-First-Out (LIFO) order.
4. RAISE – Raises
5. EXIT – Stops the execution of the struixLang code.
6. Operators from CALCGEN:
7. + - Adds last 2 numbers from stack or concatenates
8. – - Subtracts last 2 numbers from stack.
9. \* - Multiplies last 2 numbers form stack.
10. \*\* - Raises 2nd last number to the power of last number.
11. / - Divides last 2 numbers.
12. // - Divides last 2 numbers and removes decimal part to produce an integer.
13. % - Gives the remainder of division of last 2 numbers.
14. @
15. <<
16. >>
17. |
18. ^
19. ~
20. < - Relational operator LESS THAN.
21. > - Relational operator GREATER THAN.
22. <= - Relational operator LESS THAN OR EQUAL TO.
23. >= - Relational operator GREATER THAN OR EQUAL TO.
24. == - Relational operator EQUAL TO.
25. != - Relational operator NOT EQUAL TO.
26. IN – Check if 2nd last item is present within last item (string, list, etc).
27. IS – Checks if the 2nd last item is the same item (same instance of same class) as last item.
28. OR – Logical or Boolean OR.
29. AND – Logical or Boolean AND.
30. DUP – Duplicates last item in stack.
31. DROP – Removes last item from stack.
32. SWAP – Swaps last item with 2nd last item.
33. OVER – Duplicates and pushes 2nd last item to stack.
34. ROT – Duplicates and pushes 3rd last item to stack.
35. VAR – Creates a variable with following name.
36. CONST – Creates constant with following name and value.
37. = – Stores a value to a variable. (Infix)
38. STORE – Stores a value to a variable. (Postfix)
39. FETCH – Retrieves value from variable.
40. # – Flags a line for no execution.
41. IMPORT – Imports struixLang code from a .sxLib library.
42. REQUESTUNSAFE – Asks user for permission to allow potentially unsafe code.
43. PYEVAL – Evaluates a Python expression.
44. PYEXEC – Executes a Python statement.
45. PYIMPORT – Imports a python module.
46. PYLITEVAL – Evaluates a Python expression safely.
47. DEF – Starts creation of a user-defined new word.
48. END – Ends a new user-defined word definition.
49. NEXT – Skips a word and appends it to the stack.
50. [ - Starts a list.
51. ] – Ends a list.
52. LENGTH – Finds the length of a list or a string.
53. ITEM – Returns an item from a list
54. NOT – Logical or Boolean NOT.
55. TRUE – Logical or Boolean value TRUE, HIGH, or 1.
56. FALSE – Logical or Boolean value FALSE, LOW, or 0.
57. RUN – Runs a list containing struixLang code.
58. TIMES – Runs a list a number of times.
59. IFTRUE – Runs a list on receiving TRUE.
60. IFFALSE – Runs a list on receiving FALSE.
61. IFELSE – Runs one of two lists on receiving either TRUE or FALSE.
62. WHILE – Runs a list while another list yields TRUE.
63. DOWHILE – Exit-Control Loop similar to WHILE.

# struixLang 101

## Execution Process

1. Any integer, float or string goes to the stack.
2. Any comments are ignored.
3. Anything else is interpreted.

## Syntax and Other Stuff

1. Parts of code are separated only by whitespace.

“Enter 2 Numbers:” print [ input ] 2 times + print is same as

“Enter 2 Numbers:” print

[ input ] 2 times

+

print

1. Integers are numbers with no decimal points.
2. Floats are numbers with decimal points
3. Strings start with either ‘ or “.
4. Comments start with either # followed by a space.
5. struixLang is case-insensitive (except for strings; struixLang can be made case-sensitive with minor modifications to the interpreter code).
6. Infinity (inf) is a float.

# Basics of struixLang

## Input and Output

Keeping the above rules in mind, specifically Rule 1, the following should put the integer 10 on the stack:

sxL> 10

To verify, type PSTACK and press enter:

sxL> pstack

10

Let’s try again:

sxL> 11.5

sxL> pstack

The output now will not be just 11 but will list both values:

11.5

10

PSTACK is the word (command/function) which is used for displaying the entire stack (working memory) in Last-In-First-Out (LIFO) order.

To display the last element put in the stack, use PRINT:

sxL> print

11.5

sxL> pstack

10

NOTE: PRINT *and most other words* remove the item which it prints *or operates on* from the stack.

Strings can be put in the stack is a similar way:

sxL> “Hello, World!” print

Hello, World!

sxL> ‘Hello Again!’ print

Hello Again!

The INPUT word comes to your rescue when you want to accept a value from the user:

sxL> “Enter a number:” print input

Enter a number:

\_\_

The last line is a prompt for the user to type something. Whatever the user enters is pushed (appended) to the stack.

The following number accepts a number, squares it and displays it.

sxL> “Enter a number:” print input 2 \*\* print

Enter a number:

10

100

## Variables and Constants

Sometimes a value needs to be stored for easy reference. Variables are used for that. A variable can be created with the VAR word:

sxL> var a

To store values in it, the = word is used:

sxL> a = 10

Values are retrieved using the FETCH word:

sxL> a fetch print

10

But for some instances a permanent value need be assigned a name, a value which is not supposed to change after it has been assigned. The CONST word is what’s needed:

sxL> const a “Constant String”

sxL> a pstack

“Constant String”

NOTE: Constants don’t need a FETCH word to get its value, it is accessed directly.

## Mathematical Operations

In almost all programming languages, and in formal mathematics, the operator is almost always placed in between the operands like 1 + 2. This is called infix notation.

struixLang is a stack-based language and before an operator can work, the operands need to be on the stack. Hence, to reduce the complexity of the interpreter, mathematical (and most other) operations are expressed in postfix notation.

1 + 2 🡪 1 2 +

(34 + 78) \* 8 🡪 34 78 + 8 \*

Let’s try them out:

sxL> “Enter a number:” print const n input n 2 + n \* print

Enter a number:

10

120

What happens:

1. A message is displayed.
2. The input (10) is set as constant.
3. n 2 + n \* = (n + 2) n \* [∵ n 2 + 🡪 n + 2]

= (n + 2) \* n [∵ m n \* 🡪 m \* n, m = (n + 2)]

1. = (10 + 2) \* 10 [Putting value of n]
2. The result (120) is displayed.

### Mathematical Operators:

NOTE about Integer and Real Division: 5 2 / will produce 2.5 while 5 2 // will produce 2.

1. Add:

sxL> a b + = a + b

1. Subtract:

sxL> a b - = a ‒ b

1. Product:

sxL> a b \* = a × b

1. Divide as Integer:

sxL> a b // = **˻**a ÷ b**˼**

1. Divide as Real Number:

sxL> a b / = a ÷ b

1. Power:

sxL> a b \*\* = a**b**

1. Remainder:

sxL> a b % = a **mod** b

## Stack Operations

One of the most important parts of a stack-based language, stack operations are necessary for almost everything more complex than adding 2 numbers and displaying them.

ToS – Top of Stack; last element put in stack.

2oS – 2nd Item on Stack; below ToS.

3oS – 3rd Item on Stack, and so on…

### DUP – Duplicate ToS

sxL> 10 pstack

10

sxL> dup pstack

10

10

### DROP – Remove ToS

sxL> 10 11 12 pstack

12

11

10

sxL> drop pstack

11

10

### SWAP – Swap ToS and 2oS

sxL> 10 11 pstack

11

10

sxL> swap pstack

10

11

### OVER – Copy 20S on top of ToS

sxL> 10 11 pstack

11

10

sxL> over pstack

10

11

10

### ROT – Move 3oS on top of ToS

sxL> 10 11 12 pstack

12

11

10

sxL> rot pstack

10

12

11

## Comments

sxL> 10 11 + print # This is a comment.

21

Note that a space is needed after the # for the comment to work.

## List

Lists are like miniature versions of the stack†; they can store anything the stack can store.

Lists start with the [ symbol and items in it are separated by spaces. Its end is marked by a ].

sxL> [ 1 2 3 ] pstack

[1, 2, 3]

### Nested Lists

Lists can store anything the stack can store, and the stack can store lists. When a list is stored in a list the structure is known as nested lists.

† The stack is actually a list.

sxL> [ 1 2 3 [ 'a' 'b' 'c' ] ] pstack

[1, 2, 3, ['a', 'b', 'c']]

### LEN or LENGTH – Find length of List

sxL> [ 1 2 3 ] length print

3

### ITEM – To access an element from the List

The index at whose element is to be extracted is to be placed at ToS and the list at 2oS.

Indexing of lists start from 0, like most other programming language. Thus, the first element is at index 0 and the last element at index (length of list – 1).

sxL> [ 9 8 7 6 ]

sxL> 1 length print

8

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| INDEX | 0 | 1 | 2 | 3 |
| ELEMENTS | 9 | 8 | 7 | 6 |

### Executable Lists

As mentioned in the [Data Model](#_Data_Model), struixLang is a homoiconic language and can treat code as data and data as code. Lists are made to store sequences of data and thus can also store sequences of code to be treated as a single unit.

sxL> [ 10 20 + print ]

sxL> pstack

[10, 20, <function NAME at MEMORY>, <function NAME at MEMORY>]

However, having lists with code without being able to execute them is useless. The RUN word is used to execute a list in the stack.

sxL> [ 10 20 + print ]

sxL> run

30

## Conditional Statements

Conditional statements are an integral part of any programming language. struixLang includes 3 conditional structures which executes a particular list depending on a logical condition:

* IFTRUE
* IFFALSE
* IFELSE

### IFTRUE

It executes the list at ToS if 2oS is TRUE.

sxL> 2 3 < [ "2 is less than 3" print ] iftrue

2 is less than 3

### IFFALSE

It executes the list at ToS if 2oS is FALSE.

sxL> 2 3 > [ "2 is not greater than 3" print ] iffalse

2 is not greater than 3

### IFELSE

It executes the list at 2oS if 3oS is TRUE, else executes ToS.

sxL> 2 3 < [ "2 is less than 3" print ] [ "2 is greater than 3"

...> print ] ifelse

2 is less than 3

sxL> 3 2 < [ "3 is less than 2" print ] [ "3 is greater than 2"

...> print ] ifelse

3 is greater than 2

## Looping Techniques

Quite often, a section of code needs to be executed repeatedly. struixLang includes 3 looping structures which helps execute a particular list multiple times:

* TIMES
* WHILE
* DOWHILE

### TIMES

It executes the list at 2oS the number of times as determined by the ToS.

sxL> [ "Hello, World!" print ] 2 times

Hello, World!

Hello, World!

### WHILE

It keeps executing the list at ToS while the list at 2oS is TRUE. Checking is done before execution.

sxL> var i i = 0

sxL> [ i fetch 3 < ] [ i fetch print i += 1 ] while

0

1

2

### DOWHILE

It keeps executing the list at ToS while the list at 2oS is TRUE. Checking is done after execution.

sxL> var i i = 0

sxL> [ i fetch 3 < ] [ "Hello" print i += 1 ] dowhile

Hello

Hello

Hello