Stacks & Expression Evaluation via Polish Notation

Stacks

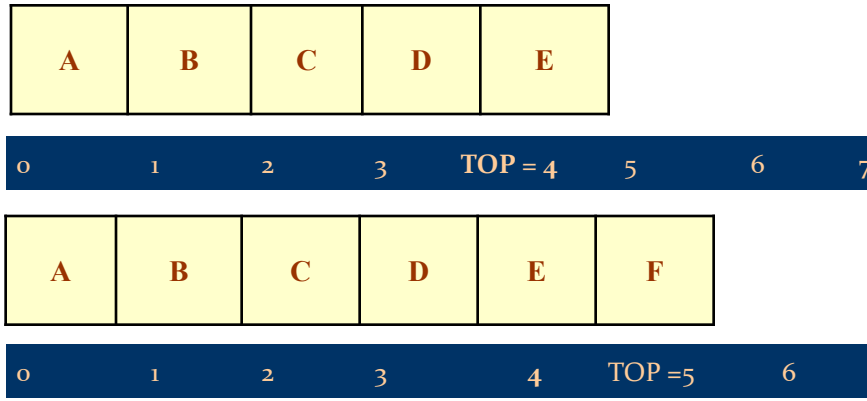
- Widely used linear data structure
- Consider a physical example a stack of plates where one plate is placed on top of the other.
- You can add and remove the plate only at/from one position, that is, the topmost position.
  - You can look at the top plate/ add a plate above it or remove the top plate exposing the plate below

Stacks

- A stack is a linear data structure with the property that the elements in a stack are added (push) and removed (pop) only from one end (top)
- A stack is called a LIFO (Last-In First-Out) data structure as the element that is inserted last is the first one to be taken out
  - Push – add an element to a stack
  - Pop -- remove the top element from stack, return or the data of top element
  - Peek – get the data of top element of stack,
  - empty() — true/false if stack has elements in it or not

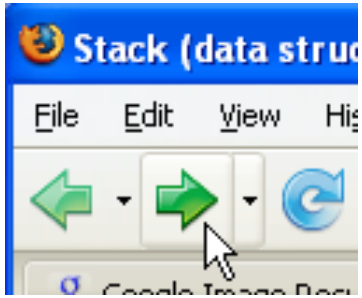
Stacks in Python Using Lists

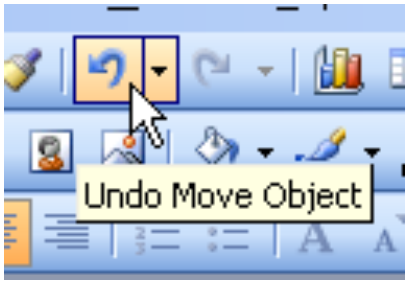
- create stack
  - S=[]
- Look at top element
  - S[-1]
- push x
  - S.append(x)
- pop and get popped value in y
  - y=S.pop()
- check if stack is empty
  - if len(S) == 0:

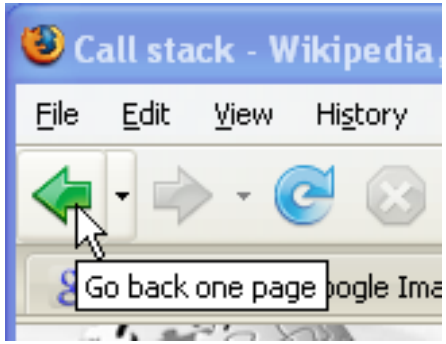


Uses of Stacks

- Undo, redo, back, forward



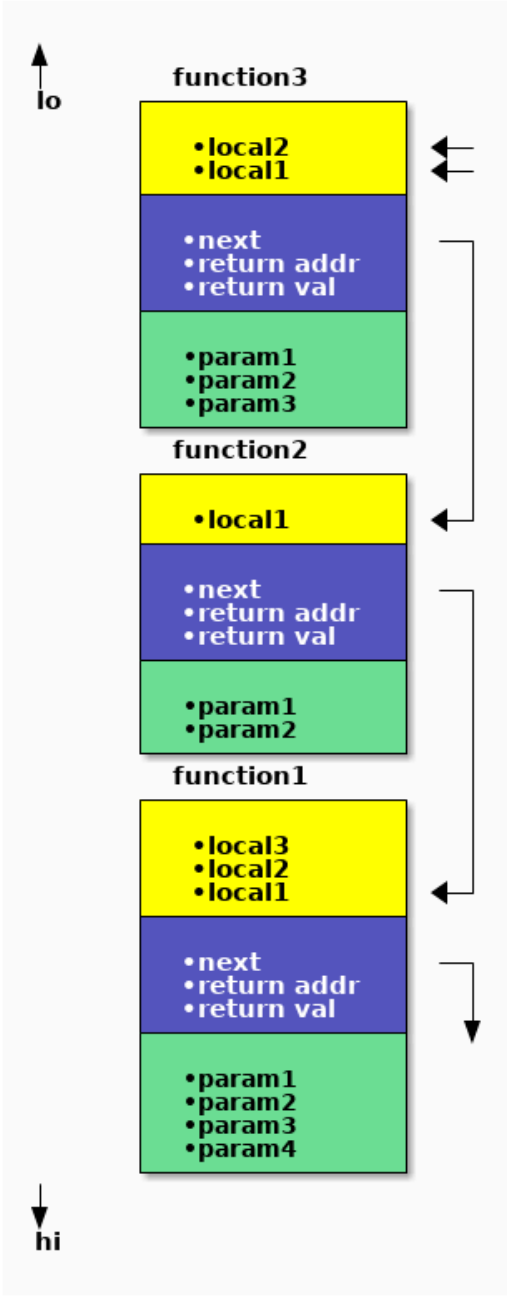




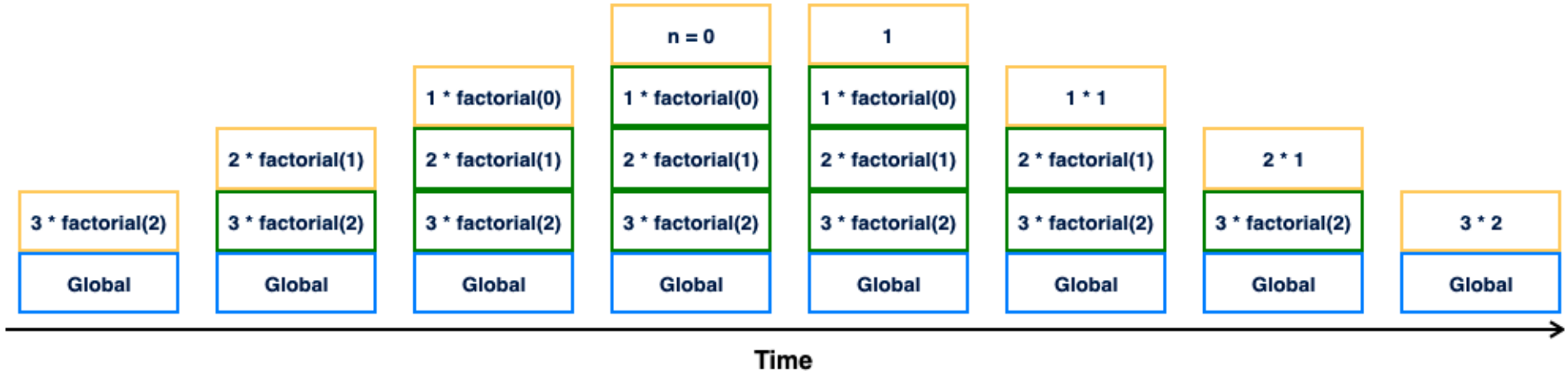
Program Execution Stack:

Stack Frame:

- Local variables
- Parameters
- Where to return



Function Call Execution Stack: Recursion



What is Output?

```
s=[]
for i in range(5):
    s.append( i )
for i in len(s):
    print( s.pop())
```

A `0 1 2 3 4` B `4 3 2 1 0`

Mathematical Calculations

- What does 3 + 2 * 4 equal?
  2 * 4 + 3? 3 * 2 + 4?
- The precedence of operators affects the order of operations.
- A mathematical expression cannot simply be evaluated left to right.
- A challenge when evaluating a program.
- *Lexical analysis* is the process of interpreting a program.

What about 1 - 2 - 4 ^ 5 * 3 * 6 / 7 ^ 2 ^ 3

Polish & Reverse Polish Notation

- For expressions we normally use infix notation as the operator is placed between the operands. For example, *A+B*.
- Although it is easy to write expressions using infix notation, computers find it difficult to parse as they need a lot of information to evaluate the expression.
- Information is needed about operator precedence, associativity rules, and brackets which overrides these rules.
- Jan Łukasiewicz was a Polish logician, mathematician, and philosopher. His aim was to develop a parenthesis-free notation. This became known as Polish prefix notation or just prefix notation. Prefix or Polish and its symmetrical Reverse Polish or Postfix notations are two different but equivalent notations of writing algebraic expressions.

Prefix Notation

- When using polish notation no rules of operator precedence or parentheses are used as long as each operator has a fixed number of operands.
- In a prefix notation, the operator is placed before the operands.
- For example, if A+B is an expression in infix notation, then the corresponding expression in prefix notation is given by +AB.
- The order of evaluation of a polish expression is always from left to right.
- While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator.
- Hence the expression (A + B) * C is written as:
  *+ABC in the prefix notation

Postfix Notation

- Postfix notation better known as Reverse Polish Notation or RPN.
- In postfix notation, the operator is placed after the operands. For example, if an expression is written as *A+B* in infix notation, the same expression can be written as *AB+* in postfix notation.
- The expression (A + B) * C is written as:

  AB+C* in the postfix notation.
- For example, above While evaluation, addition will be performed prior to multiplication.

TRY

- What does the following postfix expression evaluate to?

  6 3 2 + *

A. 18

B. 36

C. 24

D. 11

E. 30

TRY

- What does the following postfix expression evaluate to?

Ex A:6 3 2 + * => 6 5 * => 30

Ex B: 7 4 * 2 3 + * => 28 2 3 + * => 28 5 * => 140

Evaluation of Postfix Expressions

- Easy to do with a stack
- given a proper postfix expression:
  - get the next token
  - if it is an operand push it onto the stack
  - else if it is an operator
    - pop the stack for the right hand operand
    - pop the stack for the left hand operand
    - apply the operator to the two operands
    - push the result onto the stack
  - when the expression has been exhausted the result is the top (and only element) of the stack

POSTFIX EXPRESSION EVALUATION USING STACK

Ex A:6 => 6 3 => 6 3 2 + => 6 5 * => 30

Ex A:7=> 7 4 * => 28 2 3 + => 28 5 * => 140

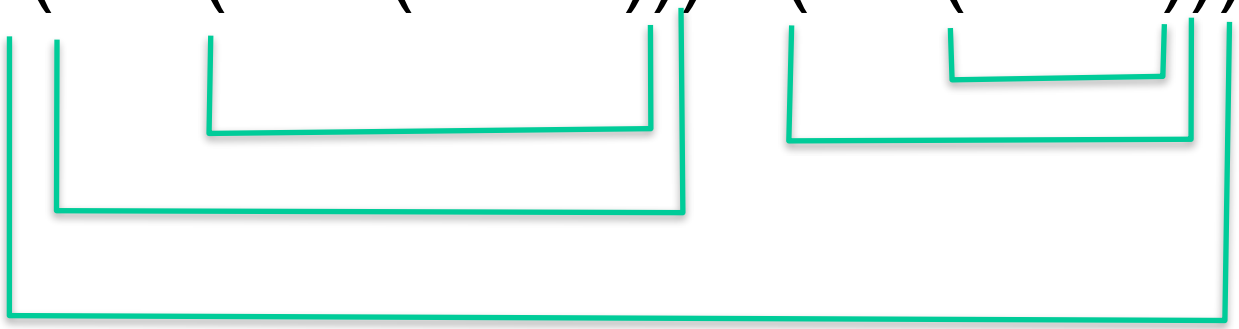
___ indicates stack contents at different stages (recall only operands are in stack)

- Requires operator precedence parsing algorithm
  - parse v. To determine the syntactic structure of a sentence or other utterance

INFIX TO POSTFIX CONVERSION

1 + 2 ^ 3 ^ 4 - 5 * (6 + 7)

((1+ (2 ^ (3 ^ 4))) - (5 * (6 + 7)))



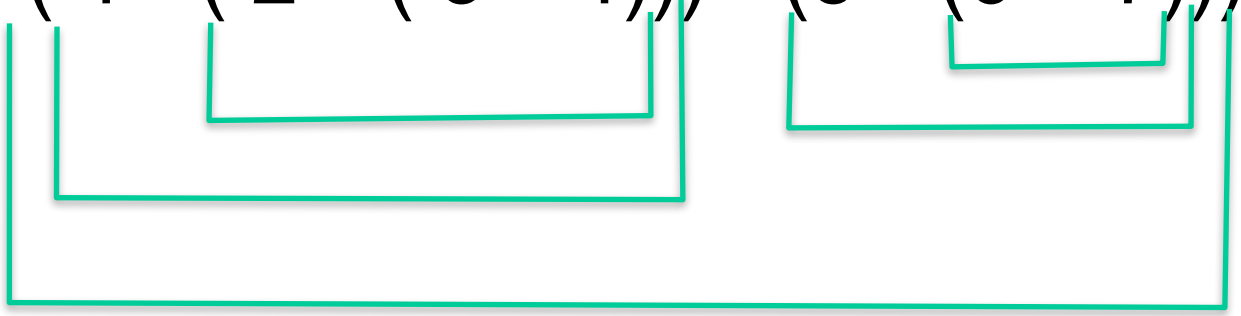
Postfix — replace) by corresponding op

1 2 3 4 ^^+ 5 6 7 +*-

Infix to Prefix Conversion

Ex: 1 + 2 ^ 3 ^ 4 - 5 * (6 + 7)

((1+ (2 ^ (3 ^ 4))) - (5 * (6 + 7)))



Prefix - replace (by corr. operator

((1 (2(3 4 (5 (6 7

- + 1 ^ 2 ^ 3 4 * 5 + 6 7

Infix to Postfix Conversion

- Requires operator precedence parsing algorithm
  - parse v. To determine the syntactic structure of a sentence or other utterance

Operands: append to output postfix expression

Close parenthesis: pop stack symbols until an open parenthesis appears. Delete the pair of parentheses.

Operators:

Pop all stack symbols until a symbol of lower precedence* appears. Then push the operator

End of input: Pop all remaining stack symbols and add to the expression

INFIX TO POSTFIX CONVERSION

1 + 2 - 3

12+3-

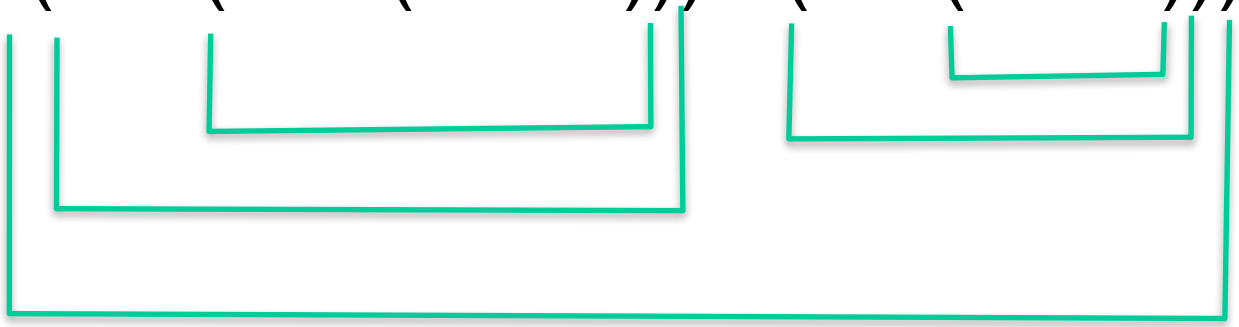
1+ 2 * 3

123*+

Ex:

1 + 2 ^ 3 ^ 4 - 5 * (6 + 7)

((1+ (2 ^ (3 ^ 4))) - (5 * (6 + 7)))



postfix expression 1 2 3

Stack +^ Input ^

Operator Precedence

Symbol	Off Stack Precedence	On Stack Precedence
+	1	1
-	1	1
*	2	2
/	2	2
^	10	9
(	20	0

Infix to Postfix Conversion

- Requires operator precedence parsing algorithm
  - parse v. To determine the syntactic structure of a sentence or other utterance

Operands: append to output postfix expression

Close parenthesis: pop stack symbols until an open parenthesis appears. Delete the pair of parentheses.

Operators:

Have an on stack and off stack precedence

Pop all stack symbols until a symbol of lower precedence appears. Then push the operator

End of input: Pop all remaining stack symbols and add to the expression

Evaluation of an Infix Expression

- Example: evaluate "9 - ((3 * 4) + 8) / 4".
- Step 1 infix "(9 - ((3 * 4) + 8) / 4)" => postfix "9 3 4 * 8 + 4 / -"
- Step 2 evaluate "9 3 4 * 8 + 4 / -"

infix	Stack	postfix
(	(	
9	(	9
-	(-	9
(	(-(	9
(	(-((	9
3	(-((	9 3
*	(-((*	9 3
4	(-((*	9 3 4
)	(-(	9 3 4 *
+	(-(+	9 3 4 *
8	(-(+	9 3 4 * 8
)	(-	9 3 4 * 8 +
/	(-/	9 3 4 * 8 +
4	(-/	9 3 4 * 8 + 4
)		9 3 4 * 8 + 4 / -

Character scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

Balanced Symbol Checking

- In processing programs and working with computer languages there are many instances when symbols must be balanced

  { } , [ ] , ( )

  A stack is useful for checking symbol balance. When a closing symbol is found it must match the most recent opening symbol of the same type.

- Applicable to checking html and xml tags!

Algorithm for Balanced Symbol Checking

- Make an empty stack
- read symbols until end of file
  - if the symbol is an opening symbol push it onto the stack
  - if it is a closing symbol do the following
    - if the stack is empty report an error
    - otherwise pop the stack. If the symbol popped does not match the closing symbol report an error
- At the end of the file if the stack is not empty report an error