Outline

1. Chapter 5: Stacks and Queues
   - Stacks

The Stack ADT

A Container Class for Last-In-First-Out Access

A stack is a **last in, first out** (**LIFO**) structure, i.e. a list-like container with access restricted to one end of the list (the top of the stack). One can

- push an item onto the stack
- pop an item off the stack (precondition: stack is not empty)
- Inspect the top position (precondition: stack is not empty)
- Obtain the current size of the stack.

The Stack ADT

Specification for a typical stack

```
class Stack:
   def __init__(self):
      """ post:  creates an empty LIFO stack"""

   def push(self,x):
      """post:  places x on top of the stack"""

   def pop(self):
      """pre:  self.size()>0
      post:  removes and returns the top element"""

   def top(self):
      """pre:  self.size()>0
      post:  returns the top element"""

   def size(self):
      """post:  returns the number of elements in the stack"""
```

Simple Stack Applications

Few Examples of Stack Applications

- graphical editors ("undo" operations)
- function calls ("nested" function calls)
- Evaluation of expressions
  example: $((x + y)/(2 * x) - 10 * z)$ - balance of grouping symbols

See the code of Stack.py

Stack Applications: Grouping Symbols

Balanced Grouping Symbols

Assume we are given an algebraic expression and are asked to check that the grouping symbols are ballanced.

Examples:

$((x + y)/(2 * x) - 10 * z)$

$[x * *3 - 2 * (2 * x * *5 - 19x * *3)]$

$\{2 - x * ([a - b] * *2 - 10 * g) + 7 * (2 - 5 * [a * *2 - b * *2])\} - 10 * x$

$\{x - y\}/\{x + y\}$

Stack Applications: Grouping Symbols

Reasoning

Questions:

- What grouping symbols can we meet?
- Do we care about all other symbols (non-grouping ones)?

Examples:

$((x + y)/(2 * x) - 10 * z)$

$[x * *3 - 2 * (2 * x * *5 - 19x * *3)]$

$\{2 - x * ([a - b] * *2 - 10 * g) + 7 * (2 - 5 * [a * *2 - b * *2])\} - 10 * x$

Stack Applications: Grouping Symbols

Balanced Grouping Symbols

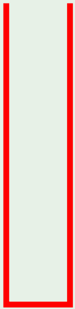
IDEA:
input: a string (or a sequence) of symbols
output: verdict (True/False)

1. get the next symbol from the input
2. if it is an opening grouping symbol, push it into the stack
3. if it is a closing grouping symbol, pop the grouping symbol from the stack, check for correspondence : {},(),[]
   if they correspond, proceed to step 1
   otherwise return False
4. (there are no more symbols in the input) if the stack is not empty return False, otherwise return True

Stack Applications: Grouping Symbols

Balanced Grouping Symbols

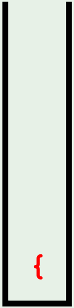
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

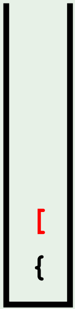
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

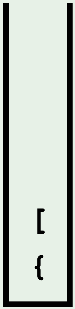
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

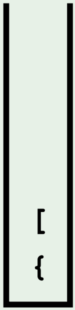
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

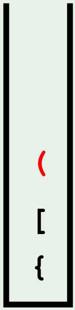
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

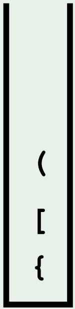
{[2 * (7 - 4) + 2] + 3} * 4

(
[
{

Stack Applications: Grouping Symbols

Balanced Grouping Symbols

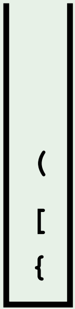
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

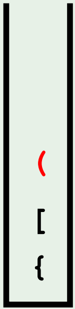
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

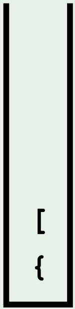
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

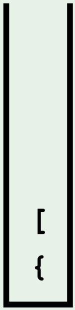
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

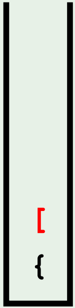
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

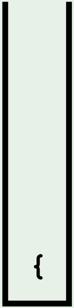
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

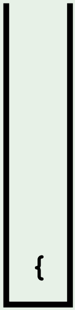
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

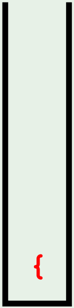
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

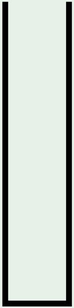
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

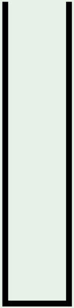
{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

{[2 * (7 - 4) + 2] + 3} * 4



Stack Applications: Grouping Symbols

Balanced Grouping Symbols

```
def parensBalance2(s):
   stack = Stack()
   for ch in s:
      if ch in "([{":  # push an opening marker
         stack.push(ch)
      elif ch in ")]}":  # match closing
         if stack.size() < 1:  # no pending open
            return False
         else:
            opener = stack.pop()
            if opener+ch not in ["()", "[]", "{}"]:
               return False # not a matching pair
   return stack.size() == 0 # everything matched?
```

An Application: Expression Manipulation

Notations For Operations

- infix notation: `(2 + 3) * 4`
  operators are between numbers
- prefix (Polish) notation: `* + 2 3 4`
  start from the right, walk to the left
- postfix (reverse Polish) notation: `2 3 + 4 *`
  start from the left, walk to the right

An Application: Expression Manipulation

prefix (Polish) notation

* + 2 3 4 =

An Application: Expression Manipulation

prefix (Polish) notation

```
  * + 2 3 4 =
= *   5   4
= 20
```

An Application: Expression Manipulation

postfix (reverse Polish) notation

```
2 3 + 4 * =
```

An Application: Expression Manipulation

postfix (reverse Polish) notation

```
  2 3 + 4 * =
=   5   4 * =
20
```

An Application: Expression Manipulation

Prefix and Postfix Notations

The *advantage* of the prefix and postfix notations: parentheses are not necessary to modify the order of operations.

An Application: Expression Manipulation

Notation For Operations

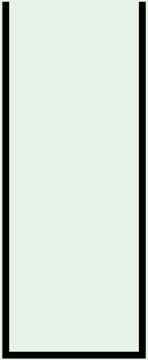
Postfix notation expressions can be evaluated easily using a stack:

- each time an operation is encountered,
- two numbers are popped off the stack,
- the operator is applied to those two numbers, and
- the result is pushed on the stack.

An Application: Expression Manipulation

Evaluating a Postfix Expression

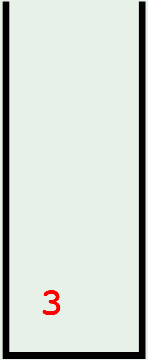
`3 4 5 + * 2 - 3 6 * +`



An Application: Expression Manipulation

Evaluating a Postfix Expression

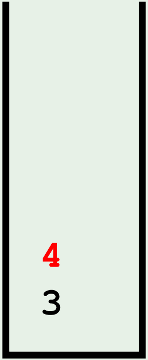
3 4 5 + * 2 - 3 6 * +



An Application: Expression Manipulation

Evaluating a Postfix Expression

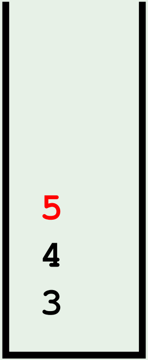
3 4 5 + * 2 - 3 6 * +



An Application: Expression Manipulation

Evaluating a Postfix Expression

3 4 5 + * 2 - 3 6 * +



An Application: Expression Manipulation

Evaluating a Postfix Expression

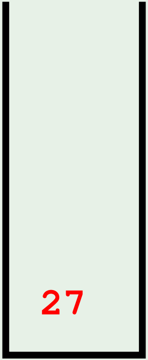
3 4 5 + * 2 - 3 6 * +

9
3

An Application: Expression Manipulation

Evaluating a Postfix Expression

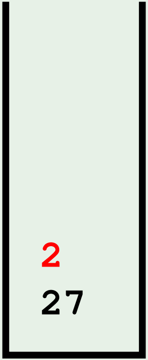
3 4 5 + * 2 - 3 6 * +



An Application: Expression Manipulation

Evaluating A Postfix Expression

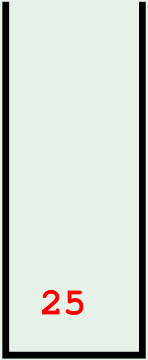
3 4 5 + * 2 - 3 6 * +



An Application: Expression Manipulation

Evaluating a Postfix Expression

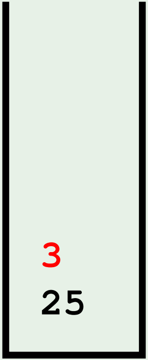
3 4 5 + * 2 - 3 6 * +



An Application: Expression Manipulation

Evaluating a Postfix Expression

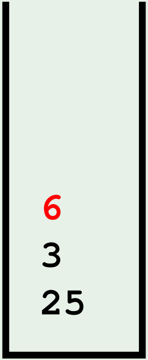
3 4 5 + * 2 - 3 6 * +



An Application: Expression Manipulation

Evaluating a Postfix Expression

3 4 5 + * 2 - 3 6 * +



An Application: Expression Manipulation

Evaluating a Postfix Expression

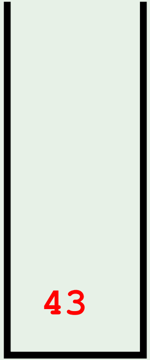
3 4 5 + * 2 - 3 6 * +

18
25

An Application: Expression Manipulation

Evaluating a Postfix Expression

3 4 5 + * 2 - 3 6 * +



An Application: Expression Manipulation

Evaluating A Postfix Expression

Note that the order in which the values are popped from the stack is important!

`4 5 - 2 *` stands for `(4-5)*2`.

Not `(5-4)*2`, not `2*(5-4)`

Your HW assignment will be to implement the evaluation of a valid post-fix expression.

The Call Stack

Function Calls Can Be Nested

- function `A` calls function `B`
- function `B` returns
- function `A` continues

The Call Stack

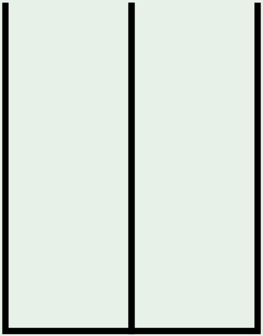
Activation Records

- Function `A` is running, and calls function `B`.
- The local variables of function `A`, their current values, and where function `B` should return to are put into an activation record.
- The activation record is pushed onto the call stack which has been allocated for the program that is running.
- When function `B` returns, this record is popped off the call stack and used to continue running the program.

The Call Stack

Example

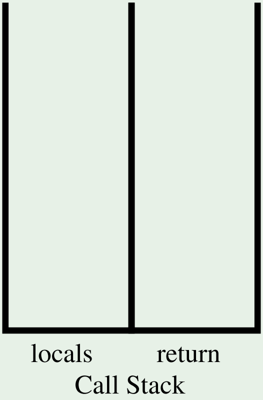
```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
 def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
 def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```



The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```

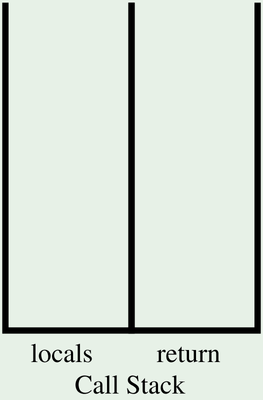


a = 3

The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
 def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
 def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```

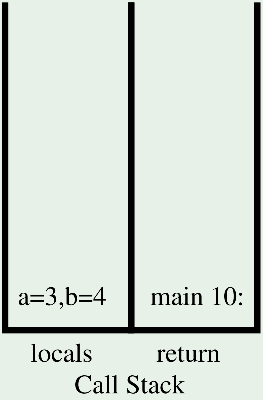


a = 3, b = 4

The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```

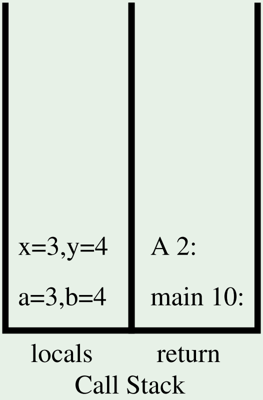


x = 3, y = 4

The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```

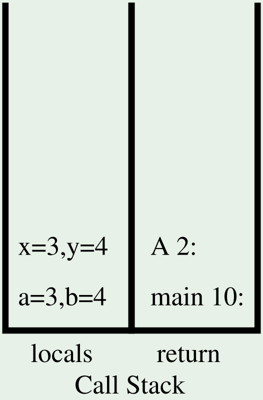


n = 3

The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
 def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
 def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```

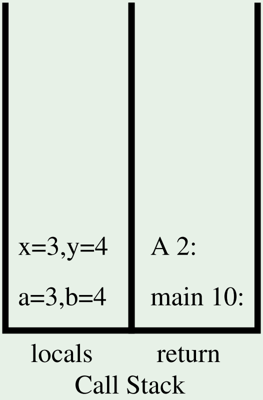


n = 3, n2 = 9

The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```

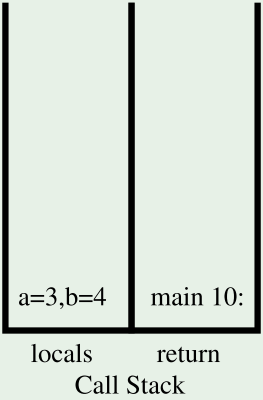


`n = 3, n2 = 9`

The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
 def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
 def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```

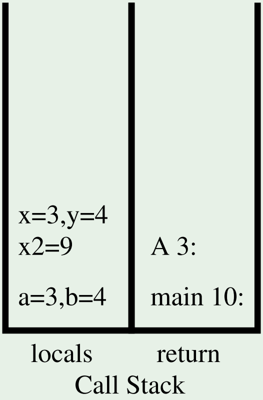


x = 3, y = 4, x2 = 9

The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```

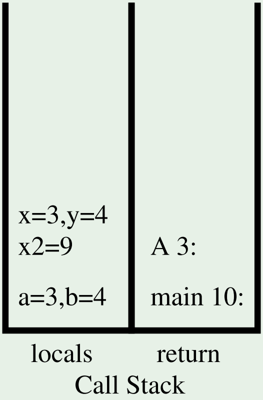


n = 4

The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```

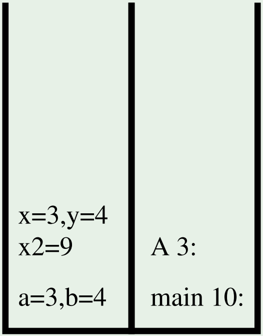


n = 4, n2 = 16

The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
 def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
 def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```



locals return

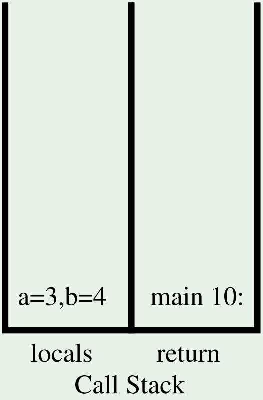
Call Stack

`n = 4, n2 = 16`

The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```

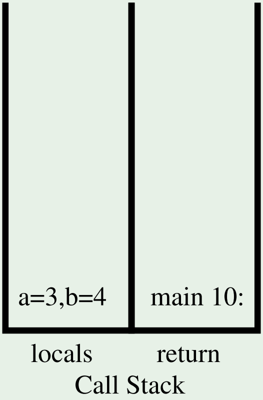


x=3,y=4,x2=9,y2=16

THE CALL STACK

EXAMPLE

```
def A(x, y):
    1:   x2 = B(x)
    2:   y2 = B(y)
    3:   z = x2 + y2
    4:   return z
 def B(n):  'squares n '
    5:   n2 = n * n
    6:   return n2
 def main():
    7:   a = 3
    8:   b = 4
    9:   c = A(a, b)
    10:   print(c)
    11:   return
```

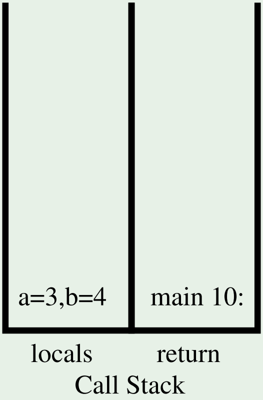


x=3,y=4,x2=9,y2=16,z=25

The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
 def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
 def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```

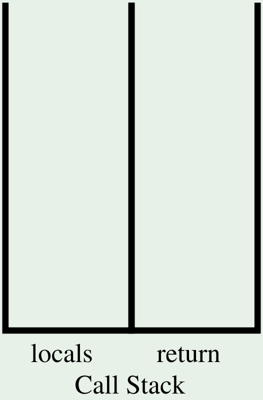


x=3,y=4,x2=9,y2=16,z=25

The Call Stack

Example

```
def A(x, y):
    1:   x2 = B(x)
    2:   y2 = B(y)
    3:   z = x2 + y2
    4:   return z
def B(n):  'squares n '
    5:   n2 = n * n
    6:   return n2
def main():
    7:   a = 3
    8:   b = 4
    9:   c = A(a, b)
    10:   print(c)
    11:   return
```

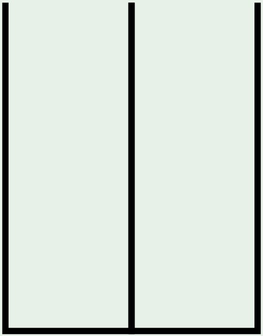


a = 3, b = 4, c = 25

The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```

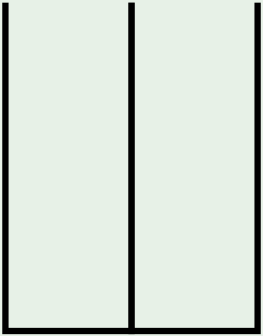


a = 3, b = 4, c = 25

The Call Stack

Example

```
def A(x, y):
   1:   x2 = B(x)
   2:   y2 = B(y)
   3:   z = x2 + y2
   4:   return z
 def B(n):  'squares n '
   5:   n2 = n * n
   6:   return n2
 def main():
   7:   a = 3
   8:   b = 4
   9:   c = A(a, b)
   10:   print(c)
   11:   return
```



locals return

Call Stack

a = 3, b = 4, c = 25

In-class work

- Re-write expression $7 * (2 + 5) - 3 * (6 - 7)$ in postfix notation
- re-write the expression 3 2 5 7 3 $- \ + \ * \ -$ (it is in postfix notation) in infix notation (common way)
- Do *unit testing* of methods push and size in Stack.py.

  For example, to test the `push` function:
  `push` a value onto the stack, retrieve it immediately (using `pop` or `top`) and check whether the retrieved value is equal to the one you just pushed.