Outline

1. Chapter 5: Stacks and Queues
   - Stacks
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.
Chapter 5: Stacks and Queues

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: \( \frac{(x + y)}{(2 \times x)} - 10 \times z \) - 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:

\[(x + y)/(2 \times x) - 10 \times z\]

\[x \times \times 3 - 2 \times (2 \times x \times \times 5 - 19x \times \times 3)\]

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

\[\{x - y\}/\{x + y\}\]
Stack Applications: Grouping Symbols

Reasoning

Questions:

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

Examples:

\[
(((x + y)/(2 \times x) - 10 \times z) \\
[x \times 3 - 2 \times (2 \times x \times 5 - 19x \times 3)] \\
\{2 - x \times ([a - b] \times 2 - 10 \times g) + 7 \times (2 - 5 \times [a \times 2 - b \times 2])\} - 10 \times x
\]
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 \times (7 - 4) + 2] + 3 \} \times 4
Balanced Grouping Symbols

\{ [2 \ast (7 - 4) + 2] + 3 \ast 4 \}
Stack Applications: Grouping Symbols

Balanced Grouping Symbols

\[ \{ [2 \ast (7 - 4) + 2] + 3 \} \ast 4 \]
**Balanced Grouping Symbols**

\[
\{ [2 \cdot (7 - 4) + 2] + 3 \} \cdot 4
\]
Stack Applications: Grouping Symbols

Balanced Grouping Symbols

\[
\{ [2 \times (7 - 4) + 2] + 3 \} \times 4
\]
Chapter 5: Stacks and Queues

Stack Applications: Grouping Symbols

Balanced Grouping Symbols:

\{ [2 * (7 - 4) + 2] + 3 \} * 4
Balanced Grouping Symbols

\[
\{ [2 \times (7 - 4) + 2] + 3 \} \times 4
\]
Stack Applications: Grouping Symbols

Balanced Grouping Symbols

\[ \{ [2 \times (7 - 4) + 2] + 3 \} \times 4 \]
Stack Applications: Grouping Symbols

Balanced Grouping Symbols

\[
\{ [2 \times (7 - 4) + 2] + 3 \} \times 4
\]
Chapter 5: Stacks and Queues

Stack Applications: Grouping Symbols

Balanced Grouping Symbols

\{ [2 \ast (7 - 4) + 2] + 3 \} \ast 4
Stack Applications: Grouping Symbols

Balanced Grouping Symbols

\[
\{ [2 * (7 - 4) + 2] + 3 \} * 4
\]
Balanced Grouping Symbols

\{ [2 \times (7 - 4) + 2] + 3 \} \times 4
Stack Applications: Grouping Symbols

Balanced Grouping Symbols

\{ [2 \times (7 - 4) + 2] + 3 \} \times 4
Chapter 5: Stacks and Queues

Stack Applications: Grouping Symbols

Balanced Grouping Symbols

\[
\{ [2 \times (7 - 4) + 2] + 3 \} \times 4
\]
### Balanced Grouping Symbols

\[
\{ [2 \times (7 - 4) + 2] + 3 \} \times 4
\]
Stack Applications: Grouping Symbols

Balanced Grouping Symbols

\{ [2 \times (7 - 4) + 2] + 3 \} \times 4
Balanced Grouping Symbols

\{ [2 \times (7 - 4) + 2] + 3 \} \times 4
Balanced Grouping Symbols

\[
\left\{ \left[ 2 \times (7 - 4) + 2 \right] + 3 \right\} \times 4
\]
Stack Applications: Grouping Symbols

Balanced Grouping Symbols

```python
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?
```
Chapter 5: Stacks and Queues

An Application: Expression Manipulation

Notations For Operations

- **infix notation**: \((2 + 3) \times 4\)
  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 =
### Prefix (Polish) Notation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 2 3 4</td>
<td>= 20</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An Application: Expression Manipulation

postfix (reverse Polish) notation

2 3 + 4 * =
### An Application: Expression Manipulation

#### Postfix (reverse Polish) notation

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>+</th>
<th>4</th>
<th>*</th>
<th>=</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>4</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

2 + 3 * 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.
Chapter 5: Stacks and Queues

Stacks

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.
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 * +

4
3
AN APPLICATION: EXPRESSION MANIPULATION

EVALUATING A POSTFIX EXPRESSION

3 4 5 + * 2 - 3 6 * +
Evaluating a Postfix Expression

3 4 5 + * 2 - 3 6 * +
Evaluating a Postfix Expression

\[ 3 \ 4 \ 5 \ + \ * \ 2 \ - \ 3 \ 6 \ * \ + \]

27
AN APPLICATION: EXPRESSION MANIPULATION

Evaluating a Postfix Expression

3 4 5 + * 2 - 3 6 * +

2
27
An Application: Expression Manipulation

Evaluating a Postfix Expression

3 4 5 + * 2 - 3 6 * +

25
AN APPLICATION: EXPRESSION MANIPULATION

EVALUATING A POSTFIX EXPRESSION

3 4 5 + * 2 − 3 6 * +

3

25
AN APPLICATION: EXPRESSION MANIPULATION

EVALUATING A POSTFIX EXPRESSION

3 4 5 + * 2 − 3 6 * +

6
3
25
Chapter 5: Stacks and Queues

AN APPLICATION: EXPRESSION MANIPULATION

EVALUATING A POSTFIX EXPRESSION

\[
3 \quad 4 \quad 5 \quad + \quad * \quad 2 \quad - \quad 3 \quad 6 \quad * \quad +
\]

18

25
AN APPLICATION: EXPRESSION MANIPULATION

EVALUATING A POSTFIX EXPRESSION

3 4 5 + * 2 - 3 6 * +

43
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.
Chapter 5: Stacks and Queues

The Call Stack

**Function Calls Can Be Nested**

- function A calls function B
- function B returns
- function A continues
**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.
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
**Example**

```python
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**
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

Call Stack

a = 3, b = 4
Chapter 5: Stacks and Queues

The Call Stack

Example

```python
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
```

Call Stack

```
a=3,b=4
main 10:
  locals
  return
  Call Stack
    x = 3, y = 4
```

CSI33 Data Structures
Chapter 5: Stacks and Queues

The Call Stack

**Example**

```python
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
```

Call Stack:
- ```x=3, y=4```  A 2:
- ```a=3, b=4```
- main 10:
  - locals
  - return
  - Call Stack
  - n = 3
```
**Example**

```python
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  A 2:  a=3,b=4  main 10:  
   locals  return  
Call Stack  
   n = 3, n2 = 9
```
**Example**

```python
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
```

Call Stack:
- x = 3, y = 4
- A (2): a = 3, b = 4
- main (10):
  - locals
  - return
  - Call Stack
  - n = 3, n2 = 9
Chapter 5: Stacks and Queues

The Call Stack

Example

```python
def A(x, y):
    x2 = B(x)
    y2 = B(y)
    z = x2 + y2
    return z

def B(n):
    n2 = n * n
    return n2

def main():
    a = 3
    b = 4
    c = A(a, b)
    print(c)
    return
```

Call Stack

```
a=3,b=4
Call Stack
def main():
a=3,b=4
Call Stack
    return
locals
Call Stack
    x = 3, y = 4, x2 = 9
```
Chapter 5: Stacks and Queues

The Call Stack

Example

```python
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
```

Call Stack

```
A 3:
 x=3, y=4
 x2=9
 a=3, b=4
 main 10: locals return
 Call Stack
 n = 4
```
The Call Stack

Example

```python
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
A 3:
a = 3, b = 4
main 10:
locals
Call Stack
n = 4, n2 = 16
```
Chapter 5: Stacks and Queues

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

Call Stack

\[
\begin{array}{c|c}
\text{n} & \text{n2} \\
\hline
4 & 16 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{x} & \text{y} \\
\hline
3 & 4 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{x2} & \text{y2} \\
\hline
9 & \text{A 3:} \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{locals} & \text{return} \\
\hline
a=3,b=4 & \text{Call Stack} \\
\end{array}
\]

n = 4, n2 = 16

CSI33 Data Structures
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
```
Chapter 5: Stacks and Queues

**The Call Stack**

**Example**

```python
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
```

<table>
<thead>
<tr>
<th>Call Stack</th>
<th>locals</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=3, b=4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>main 10:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x=3, y=4, x2=9, y2=16, z=25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CSI33 Data Structures
The Call Stack

Example

```python
def A(x, y):
    x2 = B(x)
    y2 = B(y)
    z = x2 + y2
    return z

def B(n):
    n2 = n * n
    return n2

def main():
    a = 3
    b = 4
    c = A(a, b)
    print(c)
    return
```

```
a=3,b=4 main 10:
  a=3,b=4
  main 10:
    a=3,b=4
    main 10:
      a=3,b=4
      main 10:
        locals
      return

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

```python
def A(x, y):
    1: x2 = B(x)
    2: y2 = B(y)
    3: z = x2 + y2
    4: return z

def B(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
```

**Call Stack**

```
locals  return
```

```plaintext
a = 3, b = 4, c = 25
```
The Call Stack

Example

```python
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
```

Call Stack

a = 3, b = 4, c = 25
**Example**

```python
def A(x, y):
    x2 = B(x)
    y2 = B(y)
    z = x2 + y2
    return z

def B(n): 'squares n'
    n2 = n * n
    return n2

def main():
    a = 3
    b = 4
    c = A(a, b)
    print(c)
    return

def main():
    a = 3, b = 4, c = 25
```

**The Call Stack**

![Call Stack Diagram]

```
locals  return
```

```
a = 3, b = 4, c = 25
```
In-class work

- Re-write expression $7 \times (2 + 5) - 3 \times (6 - 7)$ 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.