Data Structures in Python
Chapter 3

1. Stack Concept and ADT
2. **Stack Example - Matching**
3. Stack Example - Postfix
4. Queue
5. Deque & Profiling
6. Circular Queue
7. Linked list
8. Unordered List
9. Ordered List and Iterator

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University

Agenda

- Example Applications
- Checking for Balanced Braces
- Bracket Matching

Checking for Balanced Braces

- Algorithm
  - Initialize the stack to empty.
  - For every char read
    - If it is a non-bracket character, skip it.
    - If it is an open bracket, then push onto stack.
    - If it is a close bracket,
      - If the stack is empty, return ERROR.
      - Else, pop an element out from the stack.
  - If the stack is NON-EMPTY, ERROR.

Checking for Balanced Braces

- Examples

Input Strings

Stack as algorithm executes

	1	2	3	4	
`{a {b} c}`	{	{ {	{		1. push "{" 2. push "{" 3. pop 4. pop Stack empty → balanced
`{a {bc}`	{	{ {	}		1. push "{" 2. push "{" 3. pop Stack not empty → unbalanced
`{ab} c}`	{				1. push "{" 2. pop 3. pop Stack empty when last "}" encountered → unbalanced

Checking for Balanced Braces

- Coding

```
def bracesBalanced(expr):
    st = Stack()
    for i in range(len(expr)):
        if token[i] == '{':      #open bracket
            st.push(token[i])
        elif token[i] == '}':    #close bracket
            if st.is_empty():
                return False
            else:
                st.pop()         #pop an item
    return st.is_empty()
```

```
Expressions to test
{a{b}c}
{a{bc}
{ab}c}

Result:
{a{b}c}: True
{a{bc}: False
{ab}c}: False
```

Bracket Matching

- Ensure that pairs of brackets are properly matched
  - Examples:

```
{ a, ( b + f[ 4 ] ) * 3, d + f[ 5 ] }
```

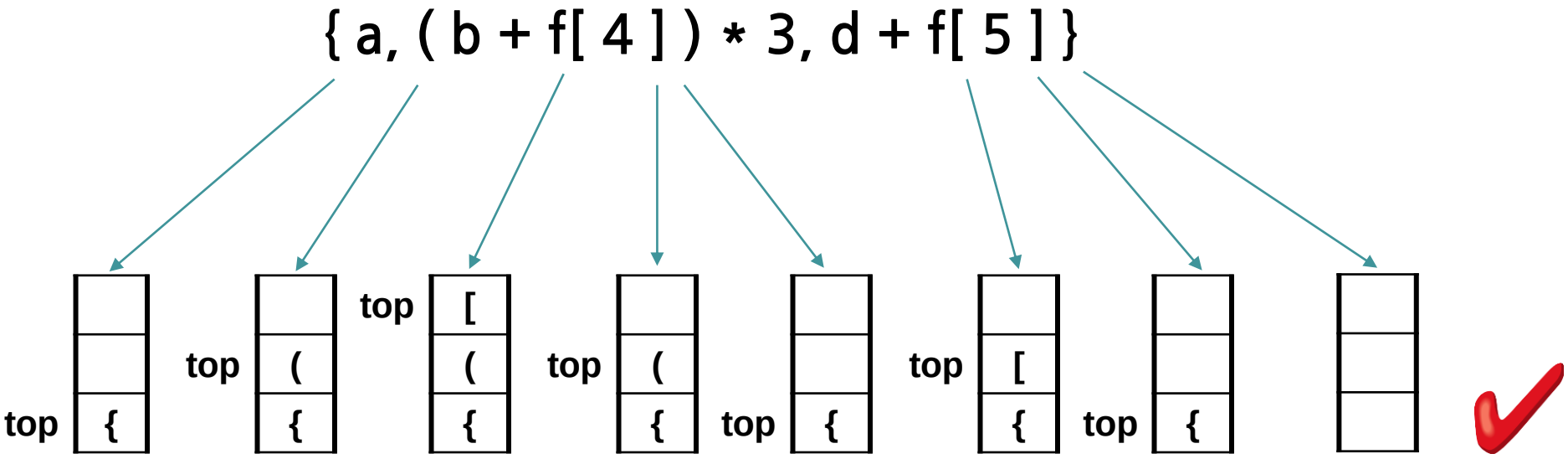
```
( ... ) ... )          // too many closing brackets
( ... ( ... )          // too many open brackets
[ ... ( ... ] ... )    // mismatch brackets
```

Bracket Matching

- Algorithm:
  - **Initialize the stack to empty.**
  - **For every char read**
    - if it is a non-bracket, skip the character
    - if it is an open bracket then push onto stack
    - if it is a close bracket, then
      - If the stack is empty, return ERROR
      - pop from the stack
      - if they don't match then return ERROR
  - **If the stack is NON-EMPTY, ERROR.**

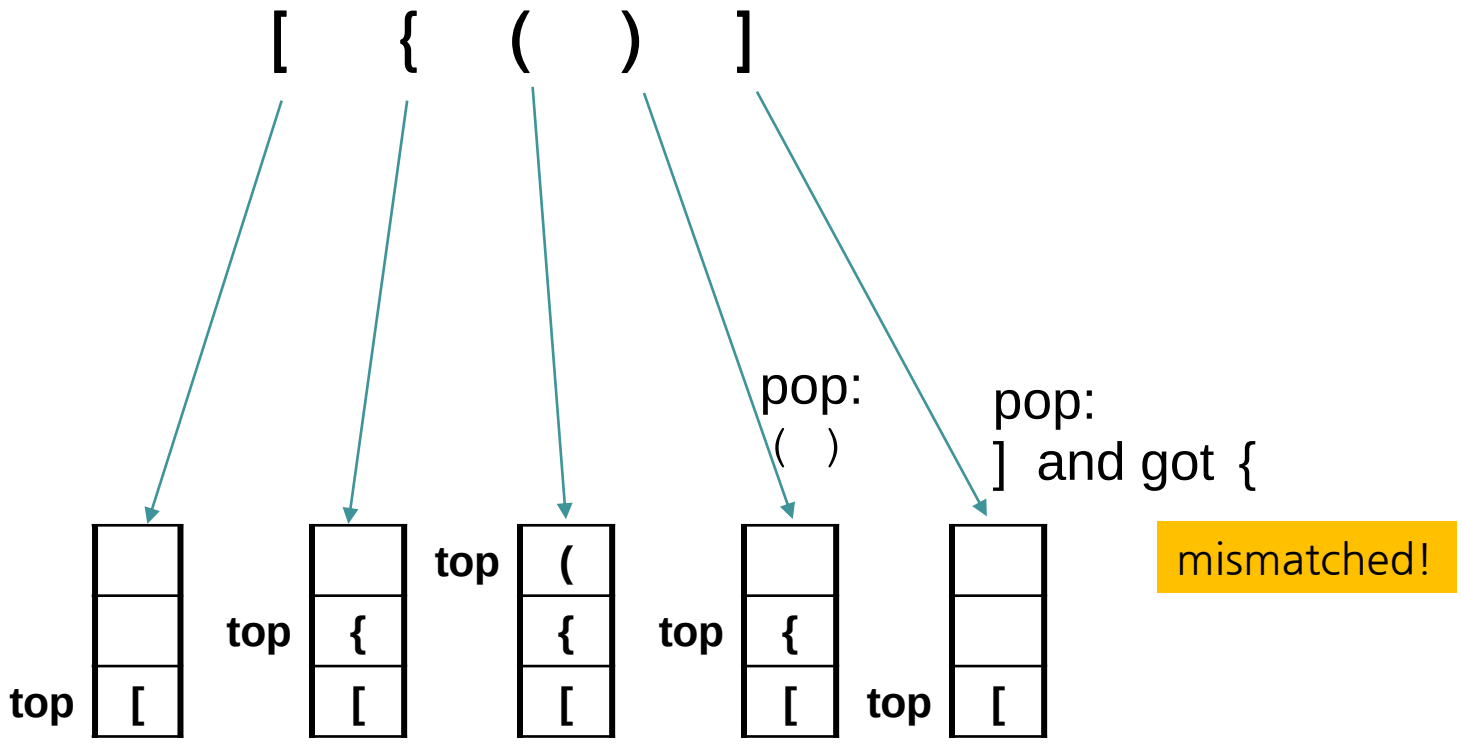
Bracket Matching

- Example 1:



Bracket Matching - Algorithm

- Example 2:



Bracket Matching

- Coding

```
def bracketsMatched(expr):
    st = Stack()
    balanced = True
    index = 0
    while index < len(expr) and balanced:
        token = expr[index]
        if token in "([{":
            st.push(token)
        elif token in ")}]":
            if st.is_empty():
                balanced = False
            else:
                top = st.pop()
            if not matches(top, token):
                balanced = False
        index = index + 1
    if balanced and st.is_empty():
        return True
    return False
```

a function to check whether the brackets are matched

```
Expressions to test:
[{()]
[{()}]
[{()}]}

Results:
[{()]: False
[{()}] : True
[{()}]} : False
```

Bracket Matching - Exercise 1

- Complete the function `matches(a, b)`
  - It is a function to check whether the brackets are matched
  - Examples:
    - `matches('(', ')')` returns `True`
    - `matches('(', '(')` returns `False`

```
def bracketsMatched(expr):
    ...
            ...
            if not matches(top, token):
                balanced = False
        index = index + 1
        ...
```

Summary

- Stacks are used in applications that manage data items in LIFO manner, such as:
  - Checking for Balanced Braces
  - Matching brackets in expressions
  - Evaluating postfix expressions
  - Conversion from Infix to Postfix