香 港 中 文 大 學 (深 圳)
The Chinese University of Hong Kong, Shenzhen

Introduction to Computer Science: Programming Methodology

Lecture 9
Recursion, Stack and Queue

Tongxin Li
School of Data Science

Linear Recursion

- If a recursive function is designed so that each invocation of the body makes at most one new recursive call, this is known as linear recursion

- Finding the smallest number and binary search are both linear recursive algorithms

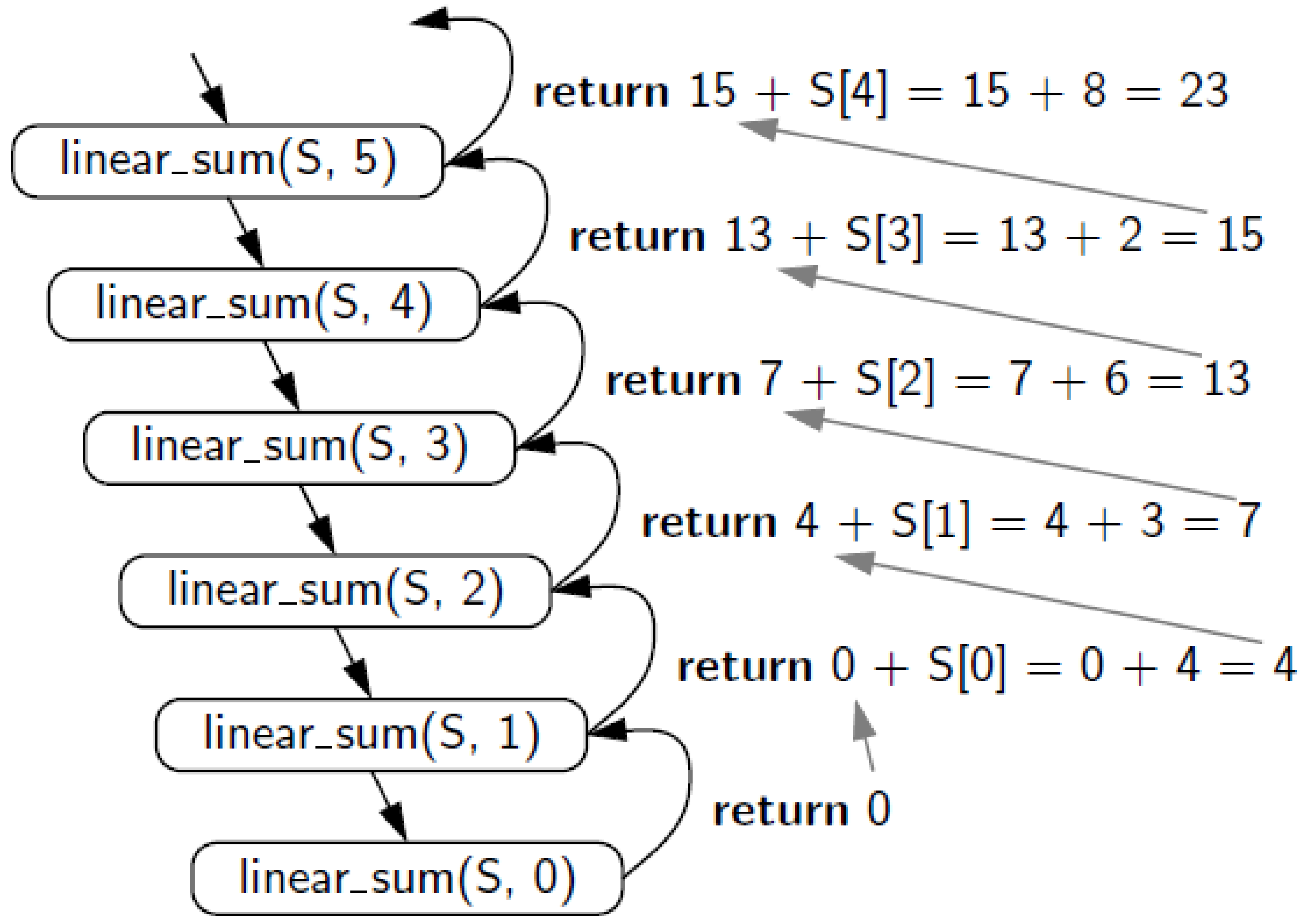
Practice: Sum of a list

- Given a list of numbers, write a program to calculate the sum of this list using recursion

Solution:

```python
def linearSum(L,n):
    if n==0:
        return 0
    else:
        return linearSum(L,n-1)+L[n-1]

def main():
    L = [1,2,3,4,5,9,100,46,7]
    print(’The sum is:’,linearSum(L,len(L)))
```

The recursive trace for recursive sum


linear_sum(S, 5)
linear_sum(S, 4)
linear_sum(S, 3)
linear_sum(S, 2)
linear_sum(S, 1)
linear_sum(S, 0)
return 15 + S[4] = 15 + 8 = 23
return 13 + S[3] = 13 + 2 = 15
return 7 + S[2] = 7 + 6 = 13
return 4 + S[1] = 4 + 3 = 7
return 0 + S[0] = 0 + 4 = 4
return 0

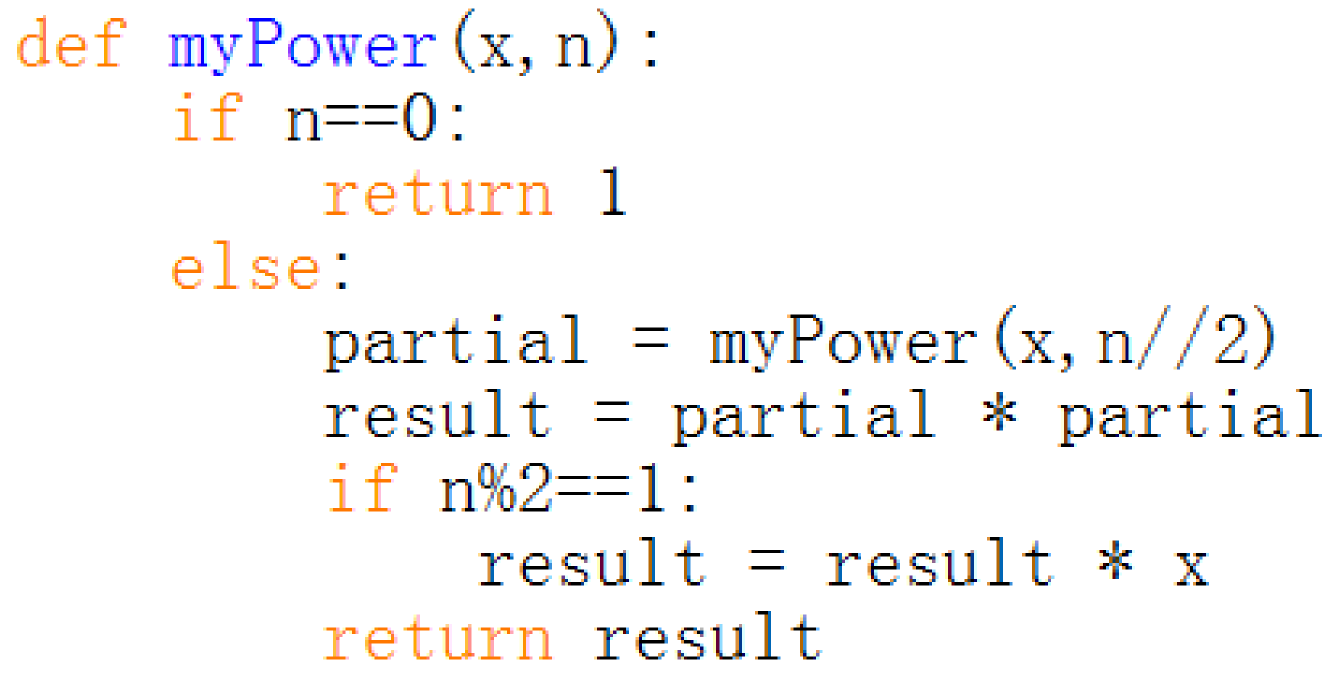
Practice: Power function

- Write a program to calculate the power function $f(x, n) = x^n$ using Recursion. The time complexity of the program should be O(logn)

A better recursive definition of power function

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 & \text{if } n > 0 \text{ is odd} \\ \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

Solution:



```python
def myPower(x,n):
    if n==0:
        return 1
    else:
        partial = myPower(x,n//2)
        result = partial * partial
        if n%2==1:
            result = result * x
        return result
```

Multiple recursion

- When a function makes two or more recursive calls, we say that it uses multiple recursion

- Drawing the English ruler is a multiple recursion program

Practice: Binary sum

- Write a function binarySum() to calculate the sum of a list of numbers. Inside binarySum() two recursive calls should be made

Solution:

```
def binarySum(L, start, stop):
    if start>=stop:
        return 0
    elif start==stop - 1:
        return L[start]
    else:
        mid = (start+stop)//2
        return binarySum(L, start, mid)+binarySum(L, mid, stop)

def main():
    L = [1, 2, 3, 4, 5, 6, 7]
    print(binarySum(L, 0, len(L)))
```

Exercise

- Print reversed numbers of an array using Recursion
  - [1,2,3] - > 3, 2, 1

```python
def merge_sort(arr):
    # Base case: arrays with less than 2 elements are already "sorted"
    if len(arr) <= 1:
        return arr

    # Divide the array into two halves
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Recursively sort both halves
    sorted_left = merge_sort(left_half)
    sorted_right = merge_sort(right_half)

    # Merge the sorted halves
    return merge(sorted_left, sorted_right)

def merge(left, right):
    merged = []
    i = j = 0

    # Merge the two arrays while comparing their elements
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    # Append any remaining elements from the left or right subarray
    merged.extend(left[i:])
    merged.extend(right[j:])

    return merged
```

Exercise

- Merge sort
  - Sort an array using Recursion
  - Worst-case time complexity?

Exercise

- Merge sort
  - Sort an array using Recursion
  - Worst-case time complexity? O(n*logn)
  - Space complexity?

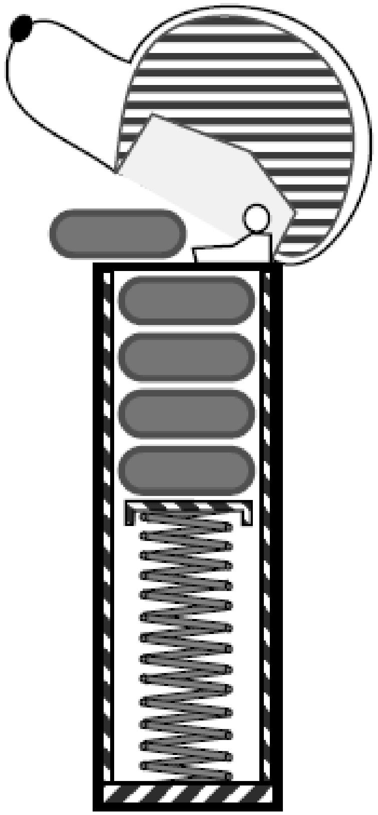
Exercise

- Merge sort
    - Sort an array using Recursion
    - Worst-case time complexity? O(n*logn)
    - Space complexity? O(n)!

Stack

- A stack is a collection of objects that are inserted and removed according to the last-in, first-out (LIFO) principle

- A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called "top" of the stack)



Example: Web Browser

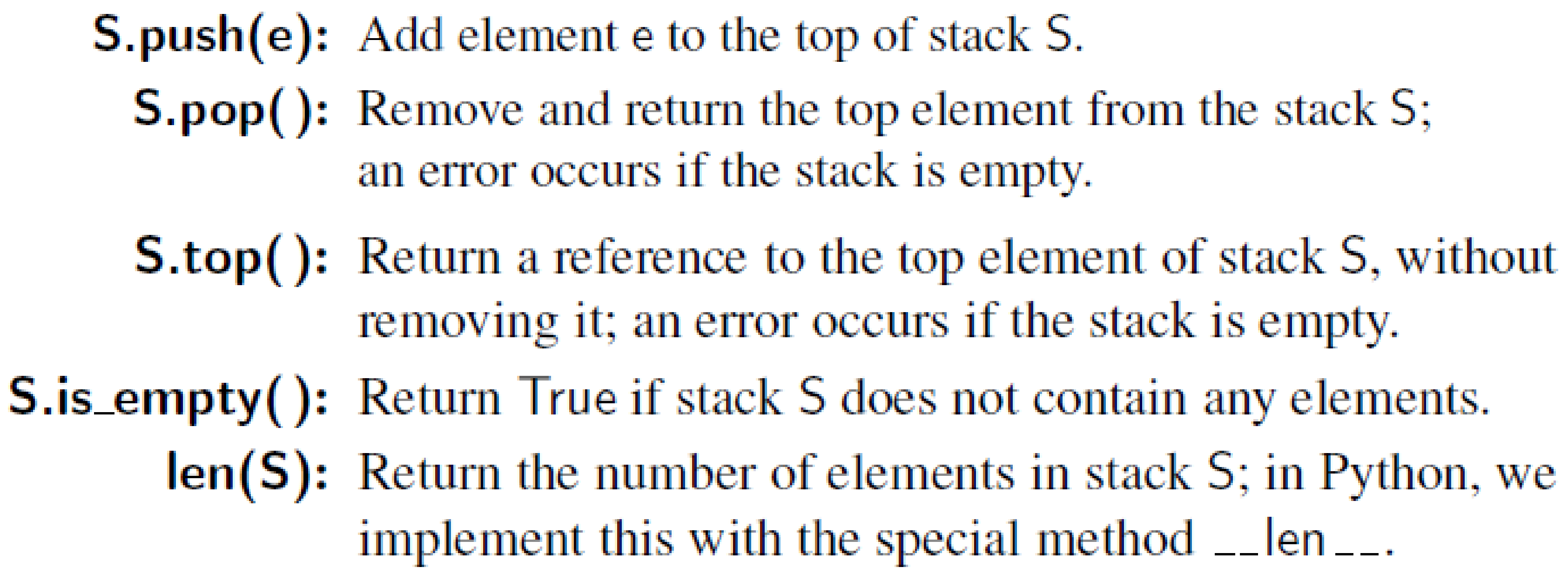
- Internet Web browsers store the addresses of recently visited sites in a stack. Each time a user visits a new site, that site’s address is “pushed” onto the stack of addresses. The browser then allows the user to “pop” back to previously visited sites using the “back” button.

Example: Text editor

- Text editors usually provide an “undo” mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack.

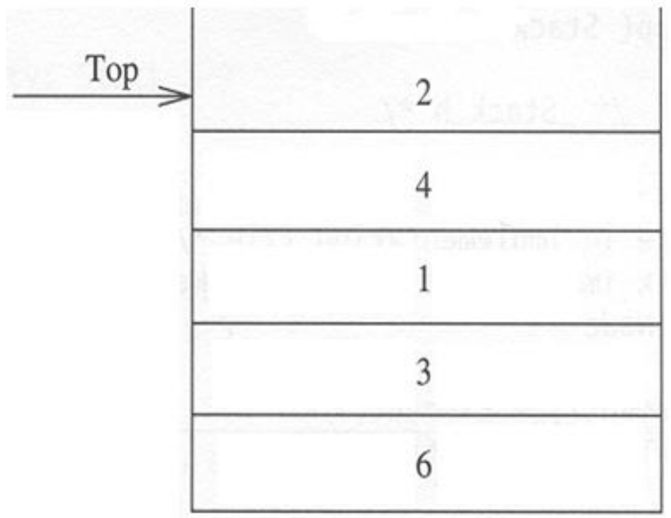
The stack class

- Generally, a stack may contain the following methods:



S.push(e):	Add element e to the top of stack S.
S.pop( ):	Remove and return the top element from the stack S; an error occurs if the stack is empty.
S.top( ):	Return a reference to the top element of stack S, without removing it; an error occurs if the stack is empty.
S.is_empty( ):	Return True if stack S does not contain any elements.
len(S):	Return the number of elements in stack S; in Python, we implement this with the special method __len__.

The Code of Stack Class



```
class ListStack:

    def __init__(self):
        self.__data = list()

    def __len__(self):
        return len(self.__data)

    def is_empty(self):
        return len(self.__data) == 0

    def push(self,e):
        self.__data.append(e)

    def top(self):
        if self.is_empty():
            print('The stack is empty.')
        else:
            return self.__data[self.__len__()-1]

    def pop(self):
        if self.is_empty():
            print('The stack is empty.')
        else:
            return self.__data.pop()
```

The code to use stack class

```python
def main():
    s = ListStack()
    print('The stack is empty? ',s.is_empty())
    s.push(100)
    s.push(200)
    s.push(300)
    print(s.top())
    print(s.pop())
    print(s.top())
```

Practice: Reverse a list using stack

- Write a program to reverse the order of a list of numbers using the stack class

Solution:

```
from stack import ListStack

def reverse_data(oldList):
    s = ListStack()
    newList = list()

    for i in oldList:
        s.push(i)

    while (not s.is_empty()):
        mid = s.pop()
        newList.append(mid)

    return newList

def main():
    oldList = [1,2,3,4,5]
    newList = reverse_data(oldList)
    print(newList)
```

Practice: Brackets match checking

- In correct arithmetic expressions, the opening brackets must match the corresponding closing brackets. Write a program to check whether all the opening brackets have matched closing brackets.

Brackets match checking

- In programing languages, there are many instances when symbols must be balanced
  - E.g., { } , [ ] , ( )
- Stack can be used for checking if the symbols are balanced
  - Balanced
    - (){[]}
    - ({{}})
    - ({[]})
  - Unbalanced
    - (]
    - (){([])}]
    - ()[[]{}

Balanced symbol checking

- **Observation**
  - If the next symbol is the opening symbol, e.g., (, [, {
    - Wait to see it matches closing symbols
  - If the next symbol is the closing symbol, e.g.,),], }
    - It needs to match previous symbols
    - E.g., if the next symbol is “)”, for a balanced expression, there must exist some “(“ in the prefix to match it

Balanced symbol checking algorithm

- **Step 1**: Create an empty stack

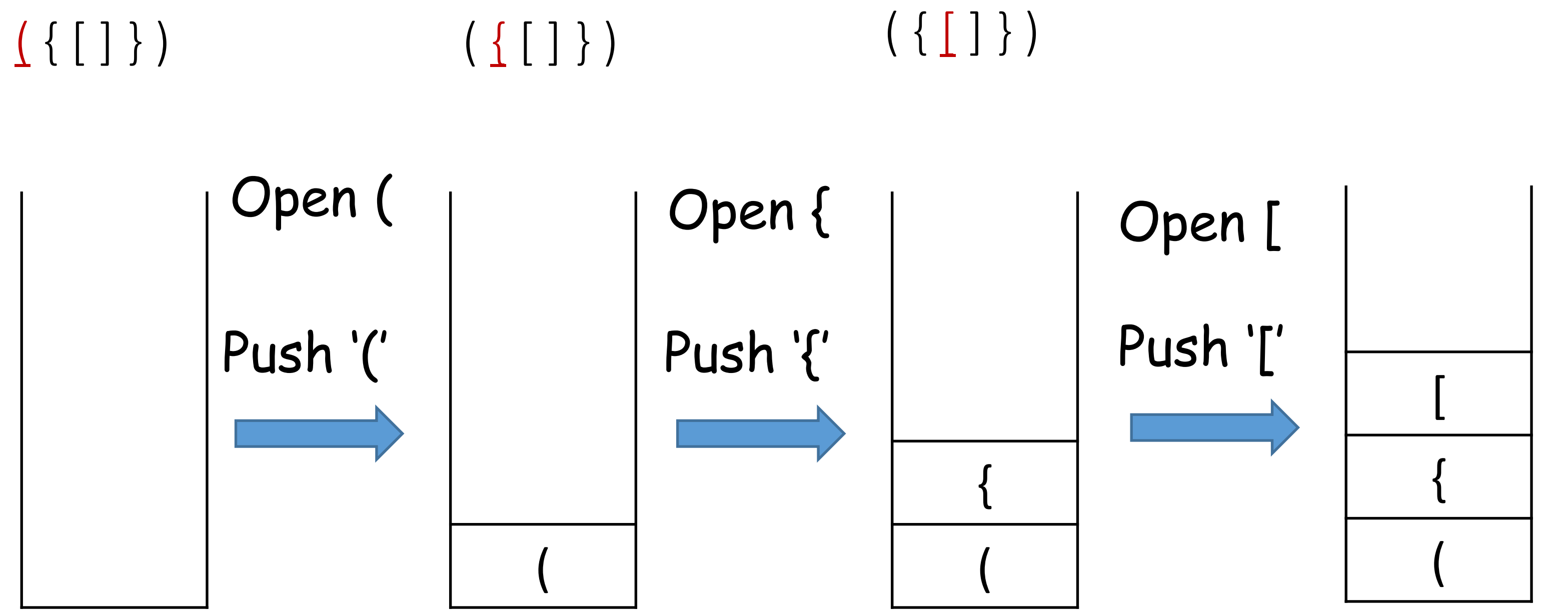
- **Step 2**: Read the symbols from the input text
  - If the symbol is an opening symbol, push it to the stack
  - If it is a closing symbol
    - If the stack is empty: return **FALSE**
    - Otherwise, pop from the stack. If the symbol popped does not match the closing symbol, return **FALSE**

- **Step 3**: At the end, if the stack is not empty, return **FALSE**

(unbalanced), else return **TRUE** (balanced)

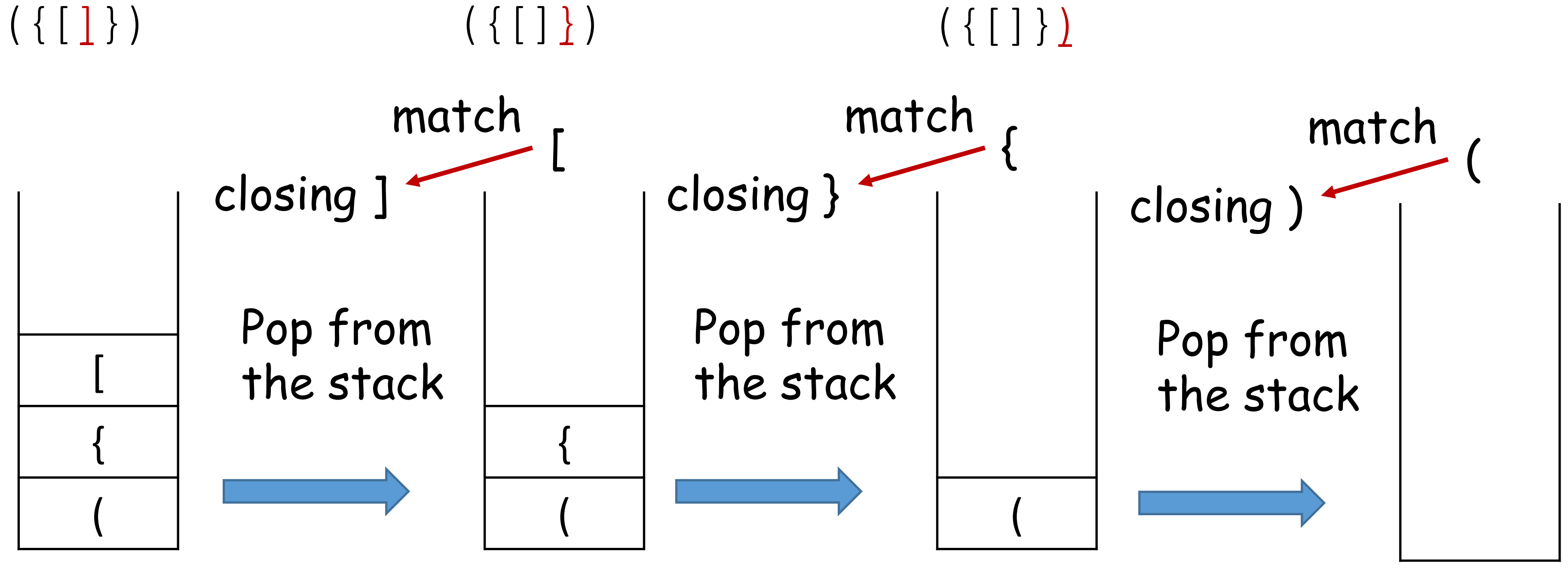
A running example

- Given an input symbol list: ({ [ ] }),
  - check if the symbols are balanced: show the status of the stack after each symbol checking



A running example

- Given an input symbol list: ({[]}),
  - check if the symbols are balanced: Show the status of the stack after each symbol checking



- After checking all symbols, the stack is empty: return TRUE

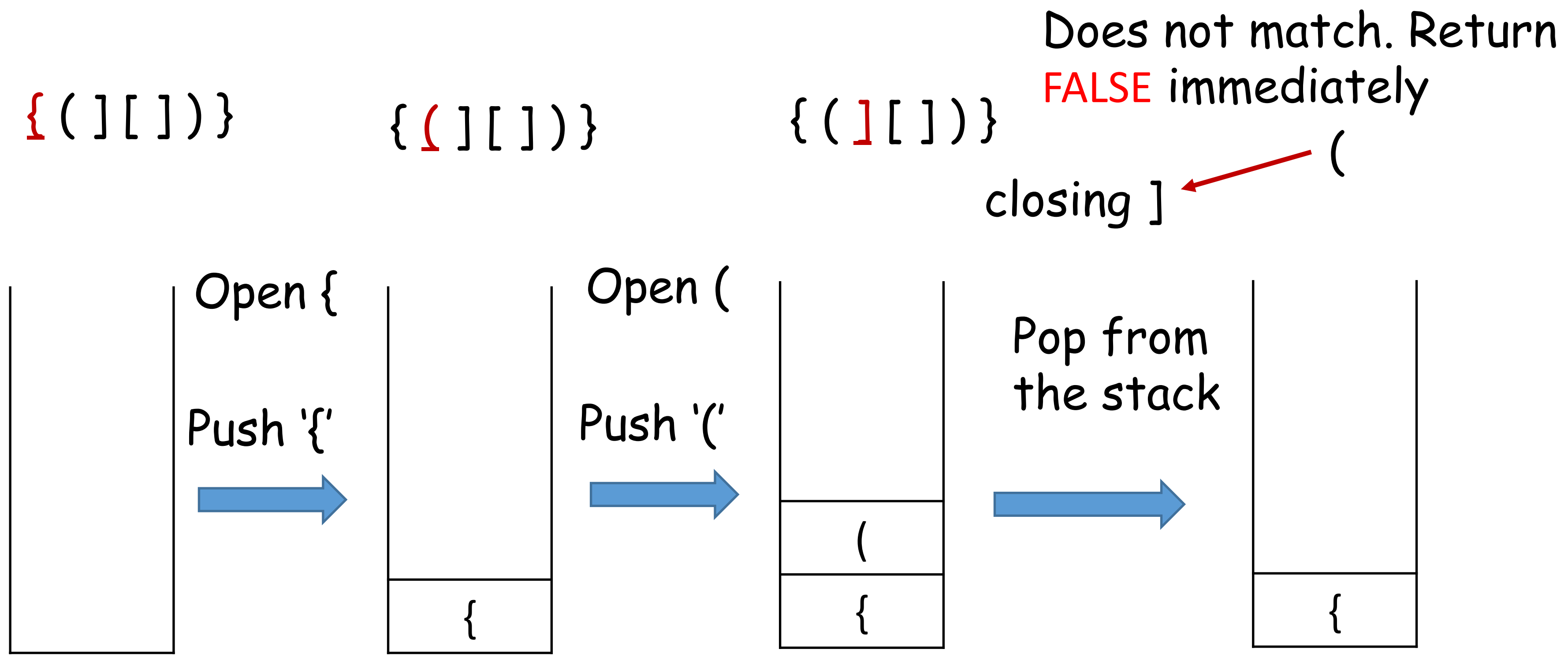
Practice

- Given an input symbol list: { ( ] [ ]) },
  - Check if the symbols are balanced
  - Show the status of the stack after each symbol checking

- Given an input symbol list: ( ) [[ ] { },
  - Check if the symbols are balanced
  - Show the status of the stack after each symbol checking

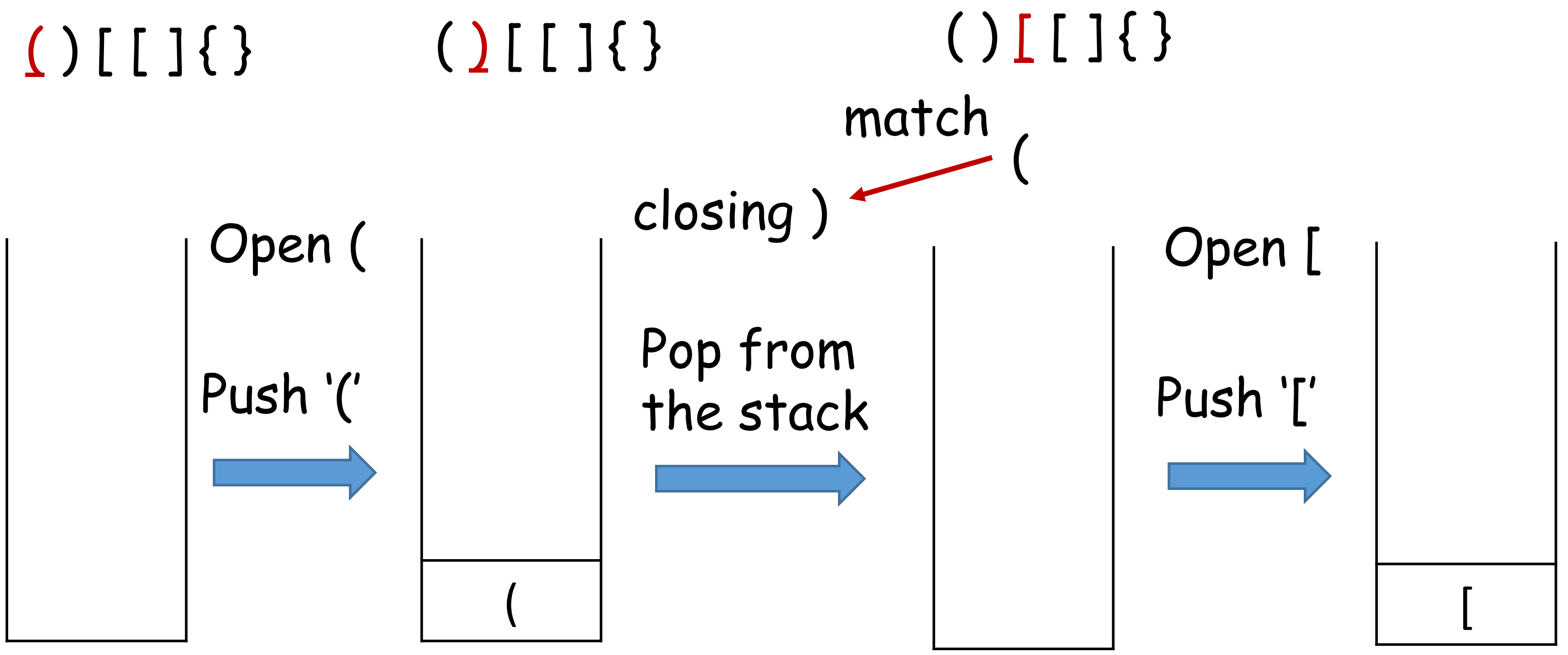
Practice

- Check if the symbol list { ( ] [ ]) } is balanced
  - Show the status of the stack after each symbol checking



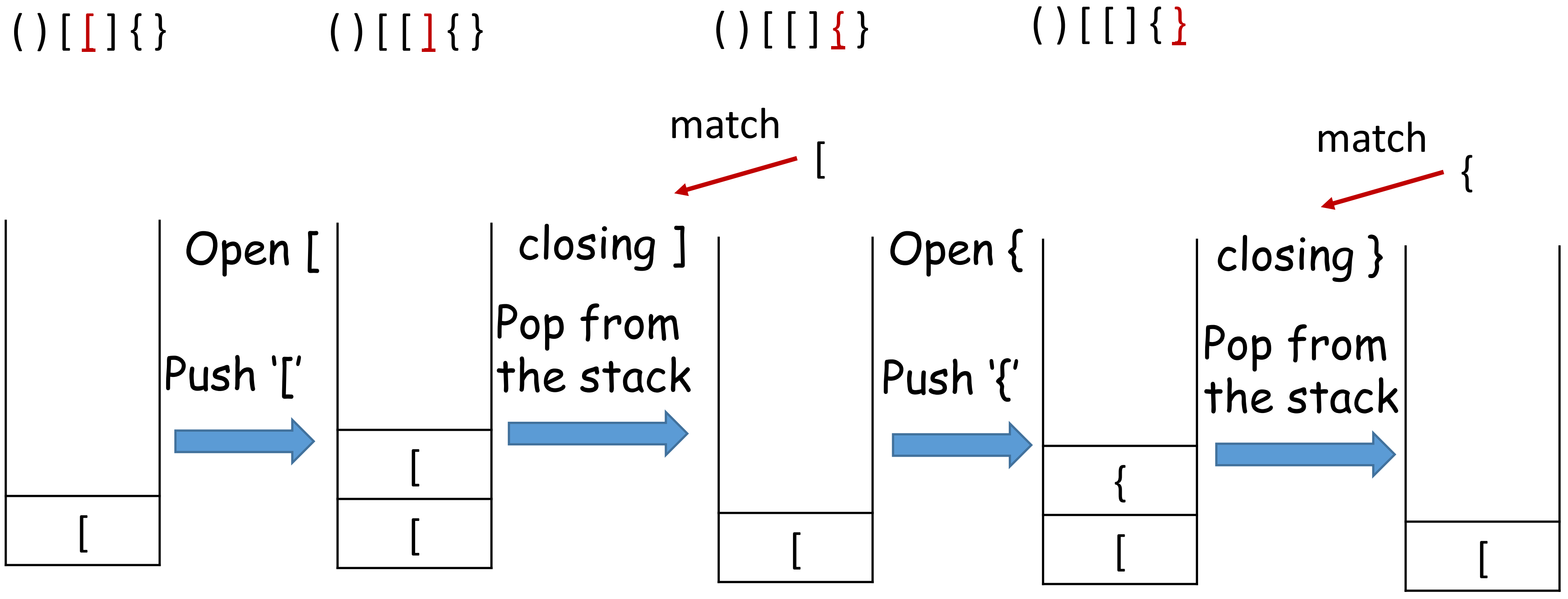
Practice

- Check if the symbol list ( ) [[ ] { } is balanced
  - Show the status of the stack after each symbol checking



Practice

- Check if the symbol list ( ) [[ ] { } is balanced
  - Show the status of the stack after each symbol checking



- Finally, the stack is not empty, so return FALSE

Solution:

```
from stack import ListStack

def is_matched(expr):
    lefty = '([{'
    righty = ')]}'

    s = ListStack()

    for c in expr:

        if c in lefty:
            s.push(c)
        elif c in righty:
            if s.is_empty():
                return False
            if righty.index(c)!=lefty.index(s.pop()):
                return False
    return s.is_empty()

def main():
    expr = '1+2*(3+4)-[5-6]'
    print(is_matched(expr))
    expr = '((())}]'
    print(is_matched(expr))
```

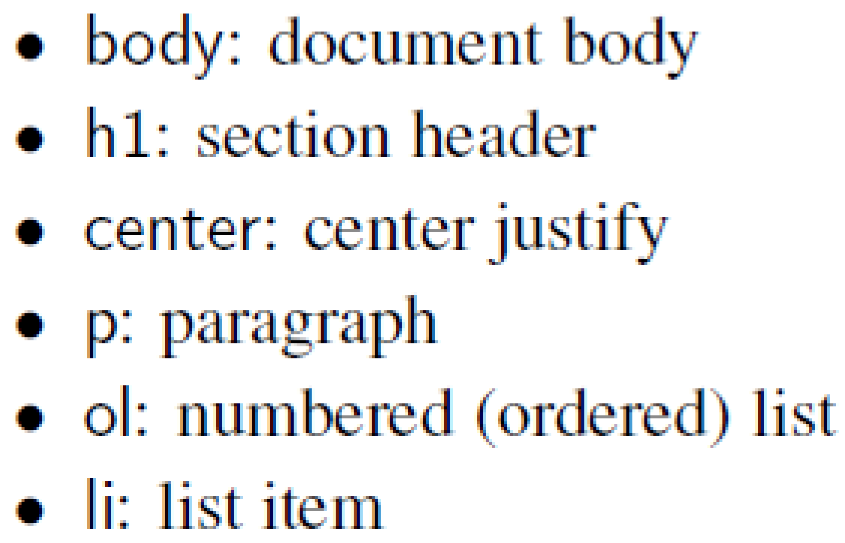
Practice: Matching Tags in HTML Language

- HTML is the standard format for hyperlinked documents on the Internet

- In an HTML document, portions of text are delimited by HTML tags. A simple opening HTML tag has the form “<name>” and the corresponding closing tag has the form “</name>”

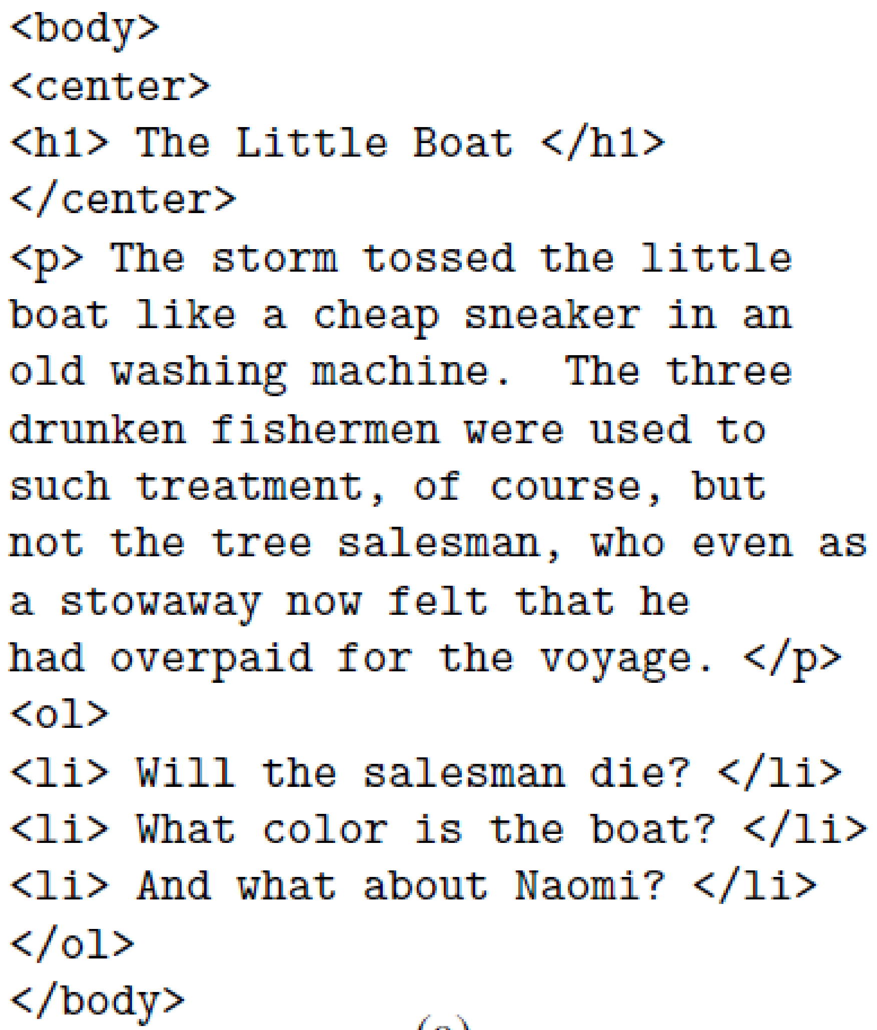
HTML Tags

- Commonly used HTML tags that are used in this example include



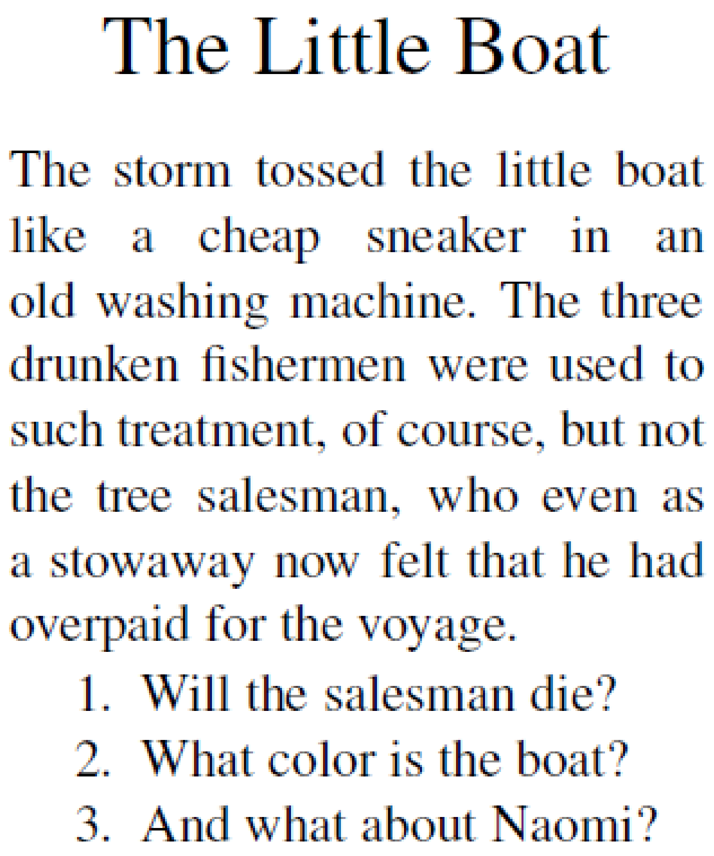
- body: document body
- h1: section header
- center: center justify
- p: paragraph
- ol: numbered (ordered) list
- li: list item

An example of HTML document



```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine.  The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

(a)



The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

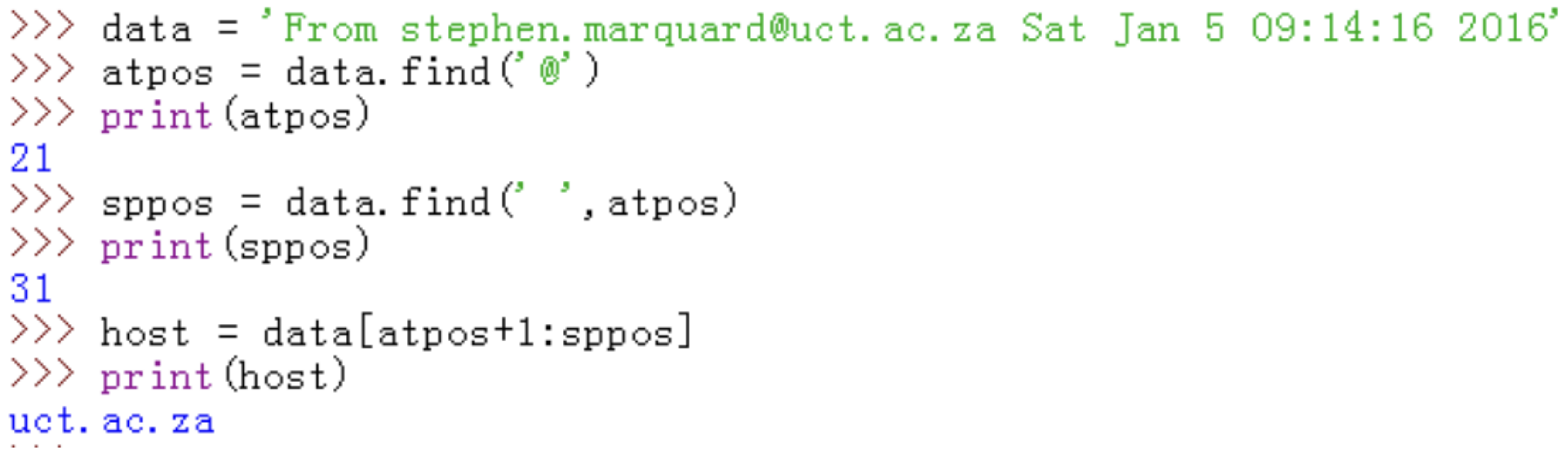
1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

(b)

Solution:

Recall: find() method for a string in **Lecture 4**

Example



```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2016'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ',atpos)
>>> print(sppos)
31
>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
```

Solution:

```
from stack import ListStack

def is_matched_html(raw):
    s = ListStack()
    j = raw.find('<')

    while j!=-1:
        k = raw.find('>', j+1)
        if k==-1:
            return False
        tag = raw[j+1:k]

        if not tag.startswith('/'):
            s.push(tag)
        else:
            if s.is_empty():
                return False
            if tag[1:]!=s.pop():
                return False
        j = raw.find('<',k+1)

    return s.is_empty()

def main():
    fhand = open('sampleHTML.txt','r')
    raw = fhand.read()
    print(raw)
    print(is_matched_html(raw))
```

smaller-than sign

greater-than sign

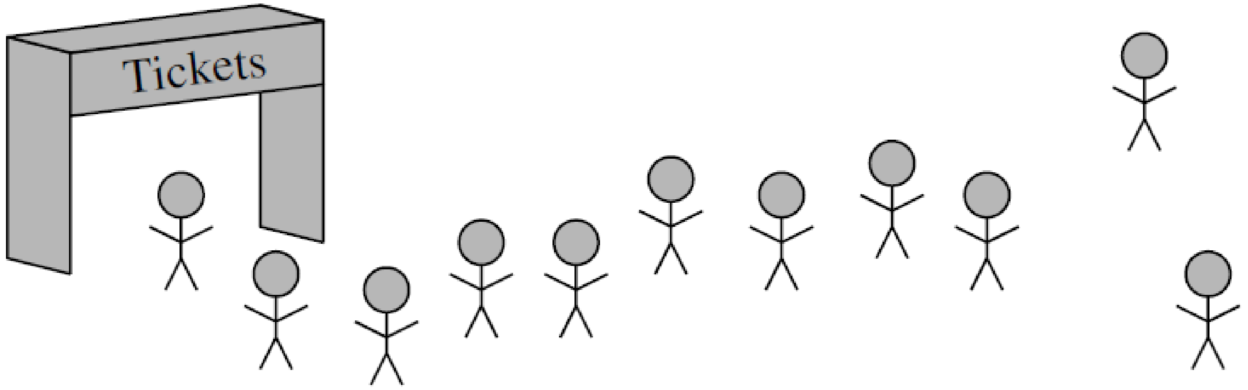
opening tag

closing tag

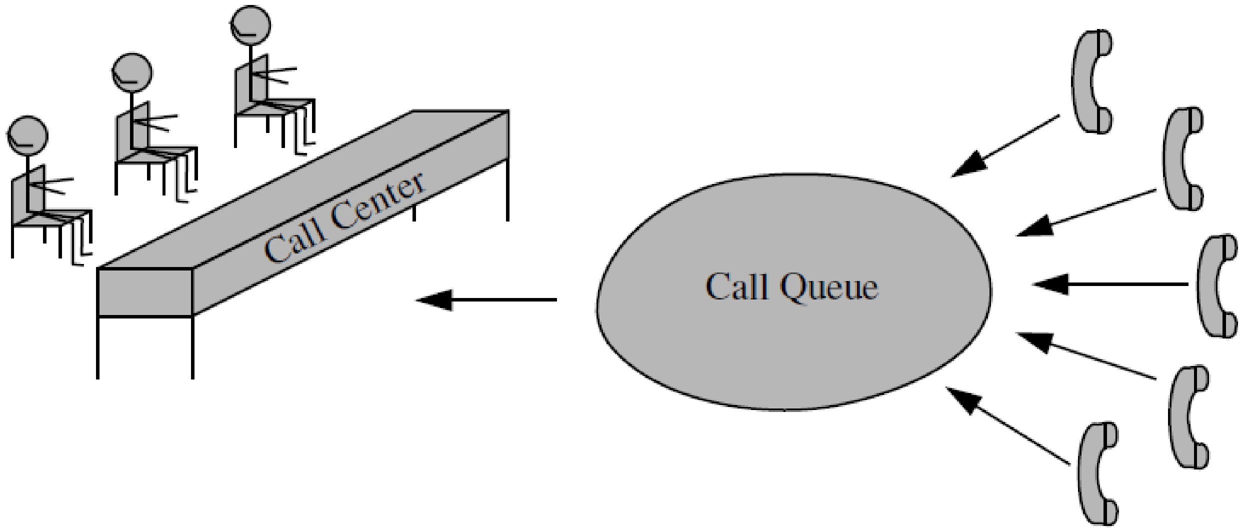
Queue

- Queue is another fundamental data structure
- A queue is a collection of objects that are inserted and removed according to the first-in, first-out (FIFO) principle
- Elements can be inserted at any time, but only the element that has been in the queue the longest can be next removed

Applications of Queue



(a)

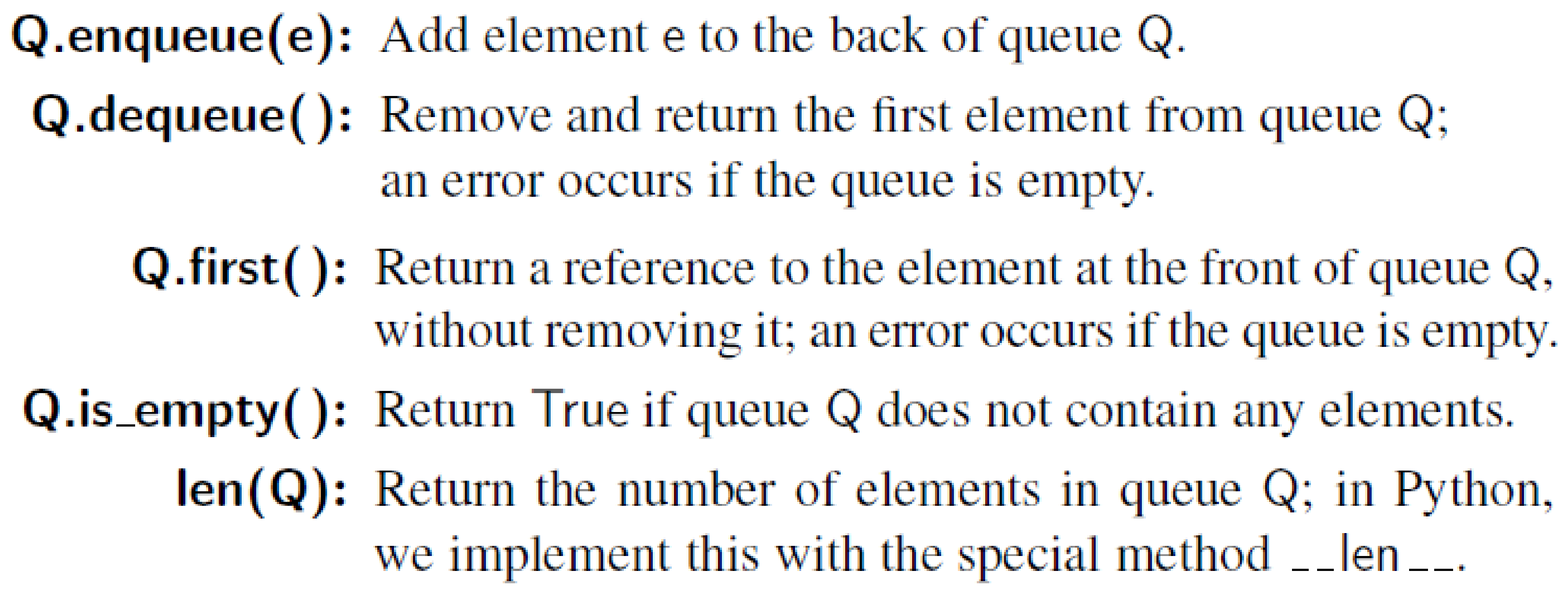


(b)

A long queue for covid19 test

The queue class

- The queue class may contain the following methods:



Q.enqueue(e):	Add element e to the back of queue Q.
Q.dequeue( ):	Remove and return the first element from queue Q; an error occurs if the queue is empty.
Q.first( ):	Return a reference to the element at the front of queue Q, without removing it; an error occurs if the queue is empty.
Q.is_empty( ):	Return True if queue Q does not contain any elements.
len(Q):	Return the number of elements in queue Q; in Python, we implement this with the special method __len__.

The code of queue class

```
class ListQueue:
    default_capacity = 5

    def __init__(self):
        self.__data = [None]*ListQueue.default_capacity
        self.__size = 0
        self.__front = 0
        self.__end = 0

    def __len__(self):
        return self.__size

    def is_empty(self):
        return self.__size ==0

    def first(self):
        if self.is_empty():
            print('Queue is empty.')
        else:
            return self.__data[self.__front]
```

```
def dequeue(self):

    if self.is_empty():
        print('Queue is empty.')
        return None

    answer = self.__data[self.__front]
    self.__data[self.__front] = None
    self.__front = (self.__front+1) \
                   % ListQueue.default_capacity
    self.__size -=1
    return answer

def enqueue(self,e):
    if self.__size == ListQueue.default_capacity:
        print('The queue is full.')
        return None

    self.__data[self.__end] = e
    self.__end = (self.__end+1) \
                 % ListQueue.default_capacity
    self.__size += 1

def outputQ(self):
    print(self.__data)
```

Practice: Simulating a web service

- An online video website handles service requests in the following way:
  1) It maintains a service queue which stores all the unprocessed service requests.
  2) When a new service request arrives, it will be saved at the end of the service queue.
  3) The server of the website will process each service request on a "first-come-first-serve" basis.

- Write a program to simulate this process. The processing time of each service request should be randomly generated.

Solution

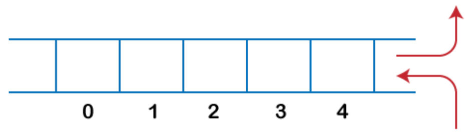
```
from ListQueue import ListQueue
from random import random
from math import floor

class WebService():
    default_capacity = 5
    def __init__(self):
        self.nameQ = ListQueue()
        self.timeQ = ListQueue()

    def taskArrive(self,taskName,taskTime):
        if self.nameQ.__len__() < WebService.default_capacity:
            self.nameQ.enqueue(taskName)
            self.timeQ.enqueue(taskTime)
            print(’A new task 《’+taskName+’》 has arrived and is waiting for processing...’)
        else:
            print(’The service queue of our website is full, the new task is dropped.’)

    def taskProcess(self):
        if (self.nameQ.is_empty() == False):
            taskName = self.nameQ.dequeue()
            taskTime = self.timeQ.dequeue()
            print(’Task 《’+taskName+’》 has been processed, it costs ’+str(taskTime)+’ seconds.’)
```

Solution

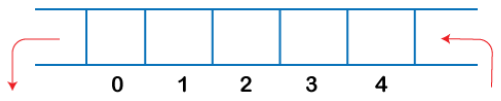
```python
def main():
    ws = WebService()
    taskNameList = ['Dark knight','X-man','Kungfu','Shaolin Soccer','Matrix','Walking in the clouds'\
                    ,'Casino Royale','Bourne Supremacy','Inception','The Shawshank Redemption']

    print('Simulation starts...')
    print('--------------------------------------')
    for i in range(1,31):
        rNum = random()
        if rNum<=0.6:
            taskIndex = floor(random()*10)
            taskTime  = floor(random()*1000)/100
            ws.taskArrive(taskNameList[taskIndex],taskTime)
        else:
            ws.taskProcess()
    print('--------------------------------------')
    print('Simulation finished.')
```

Stack vs. Queue

- **Stack**
  - The insertion and deletion operation can be performed from one side
  - The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end



- **Queue**
  - The insertion can be performed on one end, and the deletion can be done on another end
  - The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end



Practice: Simulating a stack using double queues

How to use double queues to implement a stack?

- idea?
- implementation?

Solution

```python
from collections import deque

class StackUsingQueuesAlt:
    def __init__(self):
        self.q1 = deque()
        self.q2 = deque()

    def push(self, x):
        self.q1.append(x)
        print(f"Pushed {x} onto q1: {list(self.q1)}")

    def pop(self):
        if self.is_empty():
            print("Stack is empty.")
            return None

        # Move elements except the last one to q2
        while len(self.q1) > 1:
            item = self.q1.popleft()
            self.q2.append(item)
            print(f"Moved {item} from q1 to q2: {list(self.q2)}")

        # The last element in q1 is the top of the stack
        popped_item = self.q1.popleft()
        print(f"Popped {popped_item} from q1")

        # Swap q1 and q2
        self.q1, self.q2 = self.q2, self.q1
        print(f"Swapped queues. New q1: {list(self.q1)}")
        return popped_item

    def top(self):
        if self.is_empty():
            print("Stack is empty.")
            return None

        while len(self.q1) > 1:
            self.q2.append(self.q1.popleft())

        # Get the last element
        top_item = self.q1[0]
        self.q2.append(self.q1.popleft())
        print(f"Top element is {top_item}")

        # Swap q1 and q2
        self.q1, self.q2 = self.q2, self.q1
        return top_item

    def is_empty(self):
        return not self.q1
```