Data Structures in Python

6. Abstract Data Types and Stacks

Prof. Moheb Ramzy Girgis
Department of Computer Science
Faculty of Science
Minia University

Abstract Data Types

- We refer to *the set of all possible values (the domain) of an encapsulated data "object," plus the specifications of the operations that are provided to create and manipulate the data*, as an ***abstract data type*** (***ADT***).
- In effect, all the Java built-in types are ADTs.
- A Java programmer can declare variables of those types without understanding the underlying implementation.
- The programmer can initialize, modify, and access the information held by the variables using the provided operations.
- In addition to the built-in ADTs, Java programmers can use the Java class mechanism to build their own ADTs.
- For example, the **Circle** class can be viewed as an ADT. Its developer needs to know about its underlying implementation, but its user needs only to know how to create a **Circle** object and how to invoke the methods to use the object.

Abstract Data Types *(Cont'd)*

- ***Data structure:*** *A collection of data elements whose logical organization reflects a relationship among the elements.*
- A data structure is characterized by accessing operations that are used to store and retrieve the individual data elements.
- The data elements that make up a data structure can be any combination of primitive types, unstructured composite types, and structured composite types.
- Lists, stacks, queues, trees, and graphs, are examples of data structures.
- It is best to design data structures as ADTs. We can then hide the detail of how we implement the data structure inside a class that provides methods for using the structure.

Abstract Data Types *(Cont'd)*

Basic operations that are performed on encapsulated data:

- A ***constructor*** is an operation that creates a new instance (object) of the data type.
- ***Transformers*** (sometimes called ***mutators***) are operations that change the state of one or more of the data values, such as inserting an item into an object, deleting an item from an object, or making an object empty.
- An ***observer*** is an operation that allows us to observe the state of one or more of the data values without changing them. Observers come in several forms: ***Predicates*** that ask if a certain property is true, ***Accessor*** or ***selector*** methods that return a value based on the contents of the object, and ***Summary*** methods that return information about the object as a whole.
- An ***iterator*** is an operation that allows us to process all the components in a data structure sequentially.

Stacks

- 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).
- The name "stack" is derived from the metaphor of a stack of plates in a spring-loaded, *cafeteria plate dispenser*.
- In this case, the fundamental operations involve the "pushing" and "popping" of plates on the stack.
- When we need a new plate from the dispenser, we "***pop***" the top plate off the stack.
- When we add a plate, we "***push***" it down on the stack to become the new top plate.

The Stack Abstract Data Type

- Stacks are the simplest of all data structures, yet they are also among the most important.
- They are used in a host of different applications, and as a tool for many more sophisticated data structures and algorithms.
- Formally, a stack is an abstract data type (ADT) such that an instance ***S*** supports the following two 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.
- Additionally, let us define the following accessor methods for convenience:
  - ***S.top()***: Return a reference to the top element of stack ***S***, without removing it; an error occurs if the stack is empty.

The Stack Abstract Data Type

- ***S.is_empty( )***: Return ***True*** if stack ***S*** does not contain any elements.
- ***S.len()***: Return the number of elements in stack ***S***; in Python, we implement this with the special method ***len***.

- By convention, we assume that a newly created stack is empty, and that there is no a priori bound on the capacity of the stack.
- Elements added to the stack can have arbitrary type.

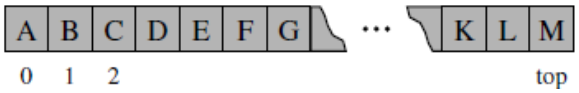
➢ **Example 1**: The following table shows a series of stack operations and their effects on an initially empty stack ***S*** of integers.

The Stack Abstract Data Type

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
S.len()	2	[5, 3]
S.pop( )	3	[5]
S.is_empty( )	False	[5]
S.pop( )	5	[ ]
S.is_empty( )	True	[ ]
S.pop( )	"error"	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top( )	9	[7, 9]
S.push(4)	–	[7, 9, 4]
S.len()	3	[7, 9, 4]
S.pop( )	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop( )	8	[7, 9, 6]

Simple Array-Based Stack Implementation

- We can implement a ***stack*** quite easily by storing its elements in a Python ***list***.
- The ***list*** class already supports:
  - adding an element to the end with the ***append*** method,
  - removing the last element with the ***pop*** method,
- So it is natural to align the top of the stack at the end of the list, as shown in the figure:



- Although a programmer could directly use the ***list*** class in place of a formal ***stack*** class, lists also include behaviors (e.g., adding or removing elements from arbitrary positions) that would break the abstraction that the ***stack ADT*** represents.

Simple Array-Based Stack Implementation

- Also, the terminology used by the ***list*** class does not precisely align with traditional terminology for a ***stack ADT***, in particular the distinction between ***append*** and ***push***.
- Instead, we will use a ***list*** for internal storage while providing a public interface consistent with a ***stack***.
- We will create a new class that performs some of the same functions as the Python's ***list*** class, using the correspondences shown in Table 1.

Table 1

Stack Method	*Realization with Python list*
S.push(e)	L.append(e)
S.pop( )	L.pop( )
S.top( )	L[−1]
S.is_empty( )	len(L) == 0
S.len()	len(L)

Simple Array-Based Stack Implementation

- We define an ***ArrayStack*** class that uses an underlying Python ***list*** for storage.
- One question that remains is what our code should do if a user calls ***pop*** or ***top*** when the stack is empty.
- Our ADT suggests that an error occurs, but we must decide what type of error.
- When ***pop*** is called on an empty Python list, it formally raises an ***IndexError***, as lists are index-based sequences.
- That choice does not seem appropriate for a stack, since there is no assumption of indices.
- Instead, we can define a new exception class that is more appropriate.
- The following code fragment defines such an ***Empty*** class as a trivial subclass of the Python ***Exception*** class:

Simple Array-Based Stack Implementation

```
class Empty(Exception):
  """Error attempting to access an element from an empty
  container. """
  pass
```

- The formal definition for our ***ArrayStack*** class is given below.
- The constructor establishes the member ***self._data*** as an initially empty Python ***list***, for internal storage.
- The rest of the public ***stack*** behaviors are implemented, using the corresponding adaptation that was outlined in Table 1.

```
class ArrayStack:
  """LIFO Stack implementation using a Python list as
  underlying storage."""
  def __init__ (self):
    """Create an empty stack."""
    self._data = [ ]        # nonpublic list instance
```

Simple Array-Based Stack Implementation

```
def len(self):
    """Return the number of elements in the stack."""
    return len(self._data)
def is_empty(self):
    """Return True if the stack is empty."""
    return len(self._data) == 0
def push(self, e):
    """Add element e to the top of the stack."""
    self._data.append(e) # new item stored at end of list
def top(self):
    """Return (but do not remove) the element at the top of the
    stack. Raise Empty exception if the stack is empty."""
    if self.is_empty( ):
        raise Empty('Stack is empty')
    return self._data[-1]   # the last item in the list
```


Simple Array-Based Stack Implementation

```
def pop(self):
    """Remove and return the element from the top of the
    stack (i.e., LIFO). Raise Empty exception if the stack is
    empty."""
    if self.is_empty( ):
        raise Empty('Stack is empty')
    return self._data.pop( ) # remove last item from list
```

- **Example 2**: Following is an example of the use of our ***ArrayStack*** class, mirroring the operations at the beginning of Example 1 on Slide 8.

```
S = ArrayStack( )      # contents: [ ]
S.push(5)              # contents: [5]
S.push(3)              # contents: [5, 3]
print(S.len())         # contents: [5, 3];      outputs 2
print(S.pop( ))        # contents: [5];         outputs 3
print(S.is_empty())    # contents: [5];         outputs False
print(S.pop( ))        # contents: [ ];         outputs 5
```

Simple Array-Based Stack Implementation

```
print(S.is_empty())  # contents: [ ];          outputs True
S.push(7)            # contents: [7]
S.push(9)            # contents: [7, 9]
print(S.top( ))      # contents: [7, 9];       outputs 9
S.push(4)            # contents: [7, 9, 4]
print(S.len())       # contents: [7, 9, 4];    outputs 3
print(S.pop( ))      # contents: [7, 9];       outputs 4
S.push(6)            # contents: [7, 9, 6]
```

- **Example 3:** *Reversing data using a stack*
- As a consequence of the ***LIFO*** protocol, a ***stack*** can be used as a general tool to reverse a data sequence.
- For example, if the values 1, 2, and 3 are pushed onto a stack in that order, they will be popped from the stack in the order 3, 2, and then 1.
- This idea can be applied in a variety of settings.

Simple Array-Based Stack Implementation

- For example, we might wish to print lines of a file in reverse order in order to display a data set in decreasing order rather than increasing order.
- This can be accomplished by reading each line and pushing it onto a ***stack***, and then writing the lines in the order they are popped.
- An implementation of such a process is given below.

```
from ArrayStack import *
def reverse_file(filename):
    """Overwrite given file with its contents line-by-line reversed."""
    S = ArrayStack()
    original = open(filename)
    for line in original:
        S.push(line.rstrip( '\n' )) # we will re-insert newlines when writing
    original.close( )
```

Simple Array-Based Stack Implementation

```
    # now we overwrite with contents in LIFO order
    output = open(filename, 'w' ) # reopening file overwrites original
    while not S.is_empty( ):
        output.write(S.pop( ) + '\n' ) # re-insert newline characters
    output.close( )
def main():
    reverse_file('myfile1.txt')
if __name__ == '__main__':
    main()
```

- **Example 4:** *Matching Delimiters*
- Consider arithmetic expressions that may contain various pairs of grouping symbols, such as
  - Parentheses: “(” and “)”
  - Braces: “{” and “}”
  - Brackets: “[” and “]”
- Each opening symbol must match its corresponding closing symbol.

Simple Array-Based Stack Implementation

- For example, a left bracket, “[,” must match a corresponding right bracket, “],” as in the expression

  [(5+x)-(y+z)]

- The following code presents a Python implementation of an algorithm for matching delimiters.

```
from ArrayStack import *
def is_matched(expr):
    """Return True if all delimiters are properly match; False
    otherwise."""
    lefty = '({[‘      # opening delimiters
    righty = ')}]'   # respective closing delims
    S = ArrayStack()
    for c in expr:
        if c in lefty:
            S.push(c) # push left delimiter on stack
```

Simple Array-Based Stack Implementation

```
        elif c in righty:
          if S.is_empty( ):
              return False # nothing to match with
          if righty.index(c) != lefty.index(S.pop( )):
              return False # mismatched
    return S.is_empty( ) # were all symbols matched?
def main():
    exp = input('Enter arithmetic expression ')
    if is_matched(exp):
        print('Delimiters are matched')
    else:
        print('Delimiters are not matched')
if __name__ == '__main__':
    main()
```

- We assume the input is a sequence of characters, such as [(5+x)-(y+z)] .

Simple Array-Based Stack Implementation

- We perform a left-to-right scan of the original sequence, using a stack ***S*** to facilitate the matching of grouping symbols.
  - Each time we encounter an *opening symbol*, we push that symbol onto ***S***
  - Each time we encounter a *closing symbol*, we pop a symbol from the stack ***S*** (assuming ***S*** is not empty), and check that these two symbols form a valid pair.
  - If we reach the end of the expression and the stack is empty, then the original expression was properly matched.
  - Otherwise, there must be an opening delimiter on the stack without a matching symbol.