6.1 Stacks

- A Stack is a collection of objects that are inserted and removed according to the last-in, first-out (LIFO) principle.
- A user can insert objects into a stack at anytime, but may only access or remove the most recently inserted object that remains (at the so-called "top" of the stack).
- Think like a PEZ candy dispenser.
- Stacks are an fundamental data structure

Examples:

1: Web browsing back and forth b/w a page.

2: Text Editors, like notability provide an "undo" mechanism, keeping changes in a stack.

The Stack Abstract Data Type:

- Stacks are the simplest of all data structures, yet among the most important
- Stacks are an Abstract Data Type (ADT), that an instance S supports:

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__`.

Simple Array-Based Stack Implementation

- Can implement a stack using a Python list.

The Adapter Pattern

- The Adaptor design pattern applies to any context where we effectively want to modify an existing class

The ***adapter*** design pattern applies to any context where we effectively want to modify an existing class so that its methods match those of a related, but different, class or interface. One general way to apply the adapter pattern is to define a new class in such a way that it contains an instance of the existing class as a hidden field, and then to implement each method of the new class using methods of this hidden instance variable. By applying the adapter pattern in this way, we have created a new class that performs some of the same functions as an existing class, but repackaged in a more convenient way. In the context of the stack ADT, we can adapt Python's list class using the correspondences shown in Table 6.1.

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
len(S)	len(L)

Example of an adapter class to provide a stack interface to a Python list:

exceptions.py

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

Stacks.py

```
1  class ArrayStack:
2    """LIFO Stack implementation using a Python list as underlying storage."""
3
4    def __init__(self):
5      """Create an empty stack."""
6      self._data = [ ]                       # nonpublic list instance
7
8    def __len__(self):
9      """Return the number of elements in the stack."""
10     return len(self._data)
11
12   def is_empty(self):
13     """Return True if the stack is empty."""
14     return len(self._data) == 0
15
16   def push(self, e):
17     """Add element e to the top of the stack."""
18     self._data.append(e)                   # new item stored at end of list
19
20   def top(self):
21     """Return (but do not remove) the element at the top of the stack.
22
23     Raise Empty exception if the stack is empty.
24     """
25     if self.is_empty( ):
26       raise Empty('Stack is empty')
27     return self._data[-1]                  # the last item in the list
28
29   def pop(self):
30     """Remove and return the element from the top of the stack (i.e., LIFO).
31
32     Raise Empty exception if the stack is empty.
33     """
34     if self.is_empty( ):
35       raise Empty('Stack is empty')
36     return self._data.pop( )               # remove last item from list
```

Call class from other file Empty.py

Example Usage

Below, we present an example of the use of our ArrayStack class, mirroring the operations at the beginning of Example 6.3 on page 230.

```
S = ArrayStack( )          # contents: [ ]
S.push(5)                  # contents: [5]
S.push(3)                  # contents: [5, 3]
print(len(S))              # 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
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(len(S))              # contents: [7, 9, 4];    outputs 3
print(S.pop( ))            # contents: [7, 9];       outputs 4
S.push(6)                  # contents: [7, 9, 6]
```

~ Creating an instance

Analyzing the Array-Based Stack Implementation

Operation	Running Time
S.push(e)	$O(1)^*$
S.pop( )	$O(1)^*$
S.top( )	$O(1)$
S.is_empty( )	$O(1)$
len(S)	$O(1)$

*amortized

- Running times of ArrayStack methods

~ O(n)-time worst case, where n is the # of elements in the stack

~ Space storage usage for a stack is O(n)

Revising 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. 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 in Code Fragment 6.3.

```
1  def reverse_file(filename):
2    """Overwrite given file with its contents line-by-line reversed."""
3    S = ArrayStack( )
4    original = open(filename)
5    for line in original:
6      S.push(line.rstrip('\n'))          # we will re-insert newlines when writing
7    original.close( )
8
9    # now we overwrite with contents in LIFO order
10   output = open(filename, 'w')       # reopening file overwrites original
11   while not S.is_empty( ):
12     output.write(S.pop( ) + '\n')  # re-insert newline characters
13   output.close( )
```

Code Fragment 6.3: A function that reverses the order of lines in a file.

- Made a test file and tested it, it works!