built-in `max` function of the built-in namespace. Thus, when called from within the `class-grades` program, it also produces a truncated highest grade, thus returning the actual highest grade of 100 instead of 102.

This example shows the care that must be taken in the use and naming of global identifiers, especially with the `from-import *` form of import. Note that if function `max` were named as a private member of the module, `__max`, then it would not have been imported into the main module and the actual highest grade displayed would have been correct.

LET'S TRY IT

Enter the following in the Python shell:

```
>>> sum([1, 2, 3])
???

>>> def sum(n1, n2, n3):
        total = n1 + n2 + n3

        return total

>>> sum([1, 2, 3])
???

>>> sum(1, 2, 3)
???
```

Create a file with the following module:

```
# module max_test_module
def test_max():
    print 'max =', max([1, 2, 3])
```

Create and execute the following program:

```
import max_test_module

def max():
    print('max:local namespace called')

print(max_test_module.test_max())
```

At any given point in a Python program's execution, there are three possible namespaces referenced ("active")—the **built-in namespace**, the **global namespace**, and the **local namespace**.

7.3.6 A Programmer-Defined Stack Module

In order to demonstrate the development of a programmer-defined module, we present an example stack module. A **stack** is a very useful mechanism in computer science. Stacks are used to temporarily store and retrieve data. They have the property that the last item placed on the stack is the first to be retrieved. This is referred to as LIFO—"last in, first out." A stack can be viewed as a list that can be accessed only at one end, as depicted in Figure 7-14.

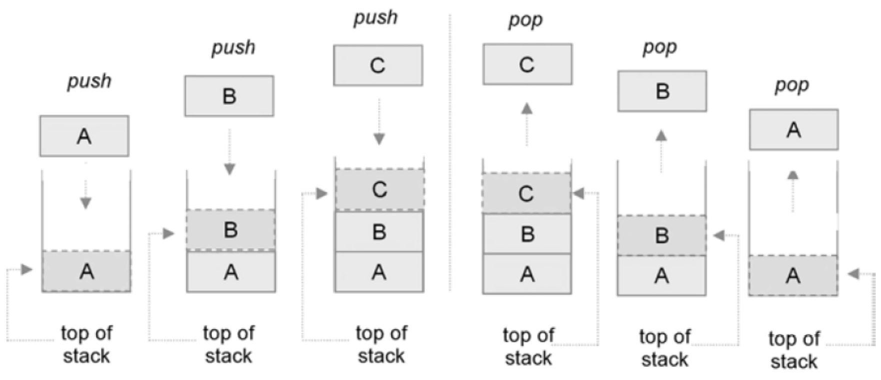

FIGURE 7-14 Stack Mechanism

In this example, three items are *pushed* on the stack, denoted by A, B, and C. First, item A is pushed, followed by item B, and then item C. After the three items have been placed on the stack, the only item that can be accessed or removed is item C, located at the *top of stack*. When C is retrieved, it is said to be *popped* from the stack, leaving item B as the top of stack. Once item B is popped, item A becomes the top of stack. Finally, when item A is popped, the stack becomes empty. It is an error to attempt to pop an empty stack.

In Figure 7-15 is a Python module containing a set of functions that implements this stack behavior. For demonstration purposes, the program displays and pushes the values 1 through 4 on the stack. It then displays the numbers popped off the stack, retrieved in the reverse order that they were pushed.

The stack module consists of five functions—`getStack`, `isEmpty`, `top`, `push`, and `pop`. The stack is implemented as a list. Only the last element in the list is accessed—that is where

```python
# stack Module

def getStack():

    """Creates and returns an empty stack."""

    return []

def isEmpty(s):

    """Returns True if stack empty, otherwise returns False. """

    if s == []:
        return True
    else:
        return False

def top(s):

    """Returns value of the top item of stack, if stack not empty.
       Otherwise, returns None.
    """

    if isEmpty(s):
        return None
    else:
        return s[len(s) - 1]

def push(s, item):

    """Pushes item on the top of stack. """

    s.append(item)

def pop(s):

    """Returns top of stack if stack not empty. Otherwise, returns None."""

    if isEmpty(s):
        return None
    else:
        item = s[len(s) - 1]
        del s[len(s) - 1]
        return item
```

FIGURE 7-15 Programmer-Defined Stack Module

all items are "pushed" and "popped" from. Thus, the end of the list logically functions as the "top" of stack.

Function `getStack` **(lines 3–7)** creates and returns a new empty stack as an empty list. Function `isEmpty` **(lines 9–16)** returns whether a stack is empty or not (by checking if an empty list). Function `top` (**lines 18–27**) returns the top item of a stack without removing it. Functions `push` **(lines 29–33)** and `pop` **(lines 35–44)** provide the essential stack operations. The `push` function pushes an item on the stack by appending it to the end of the list. The `pop` function removes the item from the top of stack by retrieving the last element of the list, and then deleting it. If either `pop` or `top` are called on an empty stack, the special value `None` is returned.

```
    # main
 1  import stack
 2
 3  mystack = stack.getStack()
 4
 5  for item in range(1, 5):
 6      stack.push(mystack, item)
 7      print('Pushing', item, 'on stack')
 8
 9  while not stack.isEmpty(mystack):
10      item = stack.pop(mystack)
11      print('Popping', item,'from stack')
```

FIGURE 7-16 Demonstration of the Programmer-Defined Stack Module

The small program in Figure 7-16 demonstrates the use of the stack module. First, a new stack is created by assigning variable `mystack` to the result of the call to function `getStack` (**line 3**). Even though a list is returned, it is intended to be more specifically a stack type. Therefore, only the stack-related functions should be used with this variable—the list should not be directly accessed, otherwise the stack may become corrupted. Then the values 1 through 4 are pushed on the stack, and popped in the reverse order,

```
Pushing 1 on stack
Pushing 2 on stack
Pushing 3 on stack
Pushing 4 on stack
Popping 4 from stack
Popping 3 from stack
Popping 2 from stack
Popping 1 from stack
```

Ensuring that only the provided functions can be used on the stack represents a fundamental advantage of objects. Since an object consists of data and methods (routines), only the methods of the object can be used to access and alter the data. Thus, the stack type is best implemented as an object, as all other values in Python are. We will see how do to this after the introduction of object-oriented programming in Chapter 10.

7.3.7 Let's Apply It—A Palindrome Checker Program

The program in Figure 7-18 determines if a given string is a palindrome. A palindrome is something that reads the same forwards and backwards. For example, the words "level" and "radar" are palindromes. The program imports the stack module developed in the previous section. This program utilizes the following programming feature:

➤ Programmer-defined module

Example execution of the program is given in Figure 7-17.

Program execution ...

```
This program can determine if a given string is a palindrome

(Enter return to exit)
Enter string to check: look
look is NOT a palindrome

Enter string to check: radar
radar is a palindrome

Enter string to check:
>>>
```

FIGURE 7-17 Execution of the Palindrome Checker Program

The stack module is imported on **line 3** of the program. The "`import` *`modulename`*" form of import is used. Therefore, each stack function is referenced by `stack.`*`function_name`*. **Lines 6–7** displays a simple program welcome. The following lines perform the required initialization for the program. **Line 10** sets `char_stack` to a new empty stack by call to `getStack()`. **Line 11** initializes variable `empty_string` to the empty string. This is used in the program for determining if the user has finished entering all the words to check.

The string to check is input by the user on **line 14**. If the string is of length one, then by definition the string is a palindrome. This special case is handled in **lines 17–18**. Otherwise, the complete string is checked. First, variable `palindrome` is initialized to `True`. On **line 24**, variable `compare_length` is set to half the length of the input string, using integer division to truncate the length to an equal number of characters. This represents the number of characters from the front of the string (working forward) that must match the number of characters on the rear of the string (working backwards). If there are an odd number of characters, then the middle character has no other character to match against.

On **lines 27–28** the second half of the string chars are pushed character-by-character onto the stack. Then, on lines **31–37** the characters are popped from the stack one by one, returning in the reverse order that they were pushed. Thus, the first character popped (the *last* character pushed on the stack) is compared to the *first* character of the complete string. This continues until there are no more characters to be checked. If characters are found that do not match, then `is_palindrome` is set to `False` (**lines 34–35**) and the while loop terminates. Otherwise, `is_palindrome` remains `True`. **Lines 40–43** output whether the input string is a palindrome or not, based on the final value of `is_palindrome`. **Lines 45–46** prompt the user for another string to enter, and control returns to the top of the while loop.

```python
# Palindrome Checker Program

import stack

# welcome
print('This program can determine if a given string is a palindrome\n')
print('(Enter return to exit)')

# init
char_stack = stack.getStack()
empty_string = ''

# get string from user
chars = input('Enter string to check: ')

while chars != empty_string:
    if len(chars) == 1:
        print('A one letter word is by definition a palindrome\n')
    else:
        # init
        is_palindrome = True

        # to handle strings of odd length
        compare_length = len(chars) // 2

        # push second half of input string on stack
        for k in range(compare_length, len(chars)):
            stack.push(char_stack, chars[k])

        # pop chars and compare to first half of string
        k = 0
        while k < compare_length and is_palindrome:
            ch = stack.pop(char_stack)
            if chars[k].lower() != ch.lower():
                is_palindrome = False

            k = k + 1

        # display results
        if is_palindrome:
            print(chars, 'is a palindrome\n')
        else:
            print(chars, 'is NOT a palindrome\n')

    # get next string from user
    chars = input('Enter string to check: ')
```

FIGURE 7-18 Palindrome Checker Program

It is important to mention that the problem of palindrome checking could be done more efficiently without the use of a stack. A for loop can be used that compares the characters `k` locations from each end of the given string. Thus, our use of a stack for this problem was for demonstration purposes only. We leave the checking of palindromes by iteration as a chapter exercise.

Self-Test Questions

1. Any initialization code in a Python module is only executed once, the first time that the module is loaded. (TRUE/FALSE)