More Python Modules

Module Private Variables

In Python, all identifiers in a module are “public” - that is, accessible by any other module that imports it. Sometimes, however, entities (variables, functions, etc.) in a module are meant to be “private” - used within the module, but not meant to be accessed from outside it.
Python does not provide any means for preventing access to variables or other entities meant to be private. Instead, there is a convention that names beginning with two underscores (_) are intended to be private. Such entities, therefore, *should not* be accessed. It does not mean that they *cannot* be accessed, however. There is one situation in which access to private variables is restricted. When the `from` *`modulename`* `import *` form of import is used to import all the identifiers of a module’s namespace, names beginning with double underscores are not imported. Thus, such entities become inaccessible from within the importing module.

Module Loading and Execution

Each imported module of a Python program needs to be located and loaded into memory. Python first searches for modules in the current directory. If the module is not found, it searches the directories specified in the PYTHONPATH environment variable. If the module is still not found (or PYTHONPATH is not defined), a Python installation-specific path is searched (e.g., `C:\Python32\Lib`). If the program still does not find the module, an error (`ImportError` exception) is reported. For our purposes, all of the modules of a program will be kept in the same directory. However, if you wish to develop a module made available to other programs, then the module can be saved in your own Python modules directory specified in the PYTHONPATH, or stored in the particular Python installation Lib directory.

When a module is loaded, a compiled version of the module with file extension `.pyc` is automatically produced. Then, the next time that the module is imported, the compiled `.pyc` file is loaded, rather than the `.py` file, to save the time of recompiling. A new compiled version of a module is automatically produced whenever the compiled version is out of date with the source code version of the module when loading, based on the dates that the files were created/modified.

Built-in Function dir()

Built-in function `dir()` is very useful for monitoring the items in the namespace of the main module for programs executing in the Python shell. For example, the following gives the namespace of a newly started shell,

```
>>>dir()
['  builtins  ', '  doc  ', '  name  ', '  package ']
```

The following shows the namespace after importing and defining variables,

```
>>> import random
>>> n = 10
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'n', 'random']
```

Selecting Shell → Restart Shell (Ctrl-F6) in the shell resets the namespace,

```
(after Restart Shell selected)
>>>dir()
['  builtins  ', '  doc  ', '  name  ', '  package ']
```

Your Turn

Create the following Python module named `simplemodule`, import it, and call function `display-Greeting` as shown from the Python shell and observe the results.

```
# simplemodule

def displayGreeting():
   print('Hello World!')
```

```
>>>import simplemodule
>>>simplemodule.displayGreeting()
```

Modify module `simplemodule` to display `'Hey there world!'`, import and again execute function
`displayGreeting` as shown. Observe the results.

```
>>>import simplemodule
>>>simplemodule.displayGreeting()
```

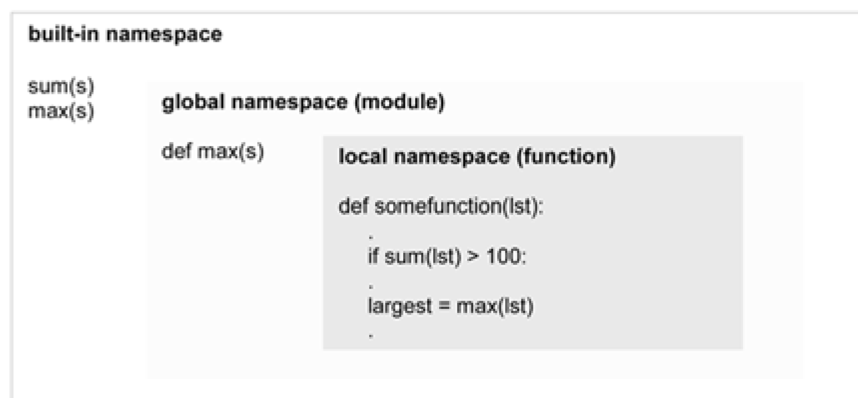
Finally, reload the module as shown and again call function `displayGreeting`.

```
>>> reload(simplemodule)
>>>simplemodule.displayGreeting()
???
```

Part II - Local, Global, and Built-in Namespaces in Python

During a Python program's execution, there are as many as three namespaces that are referenced ("active") - the built-in namespace, the global namespace, and the local namespace. The **built-in namespace** contains the names of all the built-in functions, constants, and so on, in Python. The **global namespace** contains the identifiers of the currently executing module. And the **local namespace** is the namespace of the currently executing function (if any).

When Python looks for an identifier, it first searches the local namespace (if defined), then the global namespace, and finally the built-in namespace. Thus, if the same identifier is defined in more than one of these namespaces, it becomes masked, as shown here:



Functions sum and max are built-in functions in Python, and thus in the built-in namespace. Built-in function sum returns the sum of a sequence (list or tuple) or integers. Built-in function max returns the largest value of a string, list, tuple, and other types.

In the module (and thus part of the global namespace) is defined another function named max. This programmer-defined function returns the index of the largest value from an ordered collection of items, not the value itself as built-in function max is designed to do. This is demonstrated below.

`max([4, 2, 7, 1, 9, 6])` → `9` (built-in function max)

`max([4, 2, 7, 1, 9, 6])` → `4` (programmer-defined function max)

Which specific functions are called from within `somefunction` depends on where the functions are defined. Function `sum`, for example, is not defined within the global namespace. Therefore, built-in function `sum` of the built-in namespace is called. The call to function `max`, on the other hand, does not access the built-in function max; rather, it calls function max of the more closely defined global namespace. This demonstrates how issues of scope, if not clearly considered, can result in subtle and unexpected program errors.

```
# grade_calc module

def max(grades):
    largest = 0

    for k in grades:
        if k > 100:
            largest = 100
        elif k > largest:
            largest = k

    return largest

def grades_highlow(grades):
    return (min(grades), max(grades))
```

```
# classgrades (main module)

from grade_calc import *

class_grades = [86, 72, 94, 102, 89, 76, 96]

low_grade, high_grade = grades_highlow(class_grades)
print('Highest adjusted grade on the exam was', high_grade)
print('Lowest grade on the exam was', low_grade)

print('Actual highest grade on exam was', max(class_grades))
```

This program is meant to read in the exam grades of a class. (The grades are hard-coded here for the sake of an example.) The main module imports module `grade_calc` that contains function `grades_ highlow`, which returns as a tuple the highest grade (with extra credit grades over 100 returned as 100) and lowest grade in a list of grades. Upon executing this program, we find the following results,

```
Highest adjusted grade on the exam was 100
Lowest grade on the exam was 72
The actual highest grade on the exam was 100
>>>
```

For the list of grades 86, 72, 94, 102, 89, 76, 96, the high and low grades of 72 and 100 is correct (counting the grade 102 as a grade of 100). Then the program is to display the actual highest grade of 102. However, a grade of 100 is displayed instead.

The problem is that the `grade_calc` module was imported using `from grade_calc import *`. Thus, all of the entities of the module were imported, including the defined `max` function. This function returns a "truncated" maximum grade of 100 from a list of grades, used as a supporting function for function `grades_highlow`. However, since it is defined in the global (module) namespace of the `classgrades` program, that definition of function max masks the 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.

Your Turn

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:

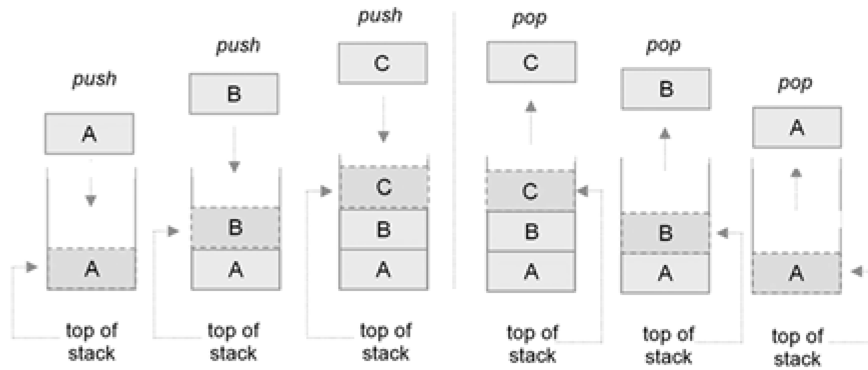
```
# module max_test_module
def test max():
     print 'max =', max ([1,2,3])
```

Create and execute the following program:

```
import max_test_module
def max():
    print ('max:local namespace called')
print (max_test_module())
```

Part III - 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 below:



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.

The following 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.

```
1  # stack Module
2
3  def getStack():
4
5      """Creates and returns an empty stack."""
6
7      return []
8
9  def isEmpty(s):
10
11     """Returns True if stack empty, otherwise returns False. """
12
13     if s == []:
14         return True
15     else:
16         return False
17
18 def top(s):
19
20     """Returns value of the top item of stack, if stack not empty.
21        Otherwise, returns None.
22     """
23
24     if isEmpty(s):
25         return None
26     else:
27         return s[len(s) - 1]
28
29 def push(s, item):
30
31     """Pushes item on the top of stack. """
32
33     s.append(item)
34
35 def pop(s):
36
37     """Returns top of stack if stack not empty. Otherwise, returns None."""
38
39     if isEmpty(s):
40         return None
41     else:
42         item = s[len(s) - 1]
43         del s[len(s) - 1]
44         return item
```

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 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')
```

The small program above 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.

Concepts and Procedures

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

2. With the "import *moduleName*" form of import, any utilized entities from the imported module must be prefixed with the module name. (TRUE/FALSE)

3. When importing modules, all Python Standard Library modules must be imported before any programmer-defined modules, otherwise a runtime error will occur. (TRUE/FALSE)

4. If a particular module is imported more than once in a Python program, the Python interpreter will ensure that the module is only loaded and executed the first time that it is imported. (TRUE/FALSE)

5. The_______________ command can be used to force the reloading of a given module, useful for when working interactively in the Python shell.

Problem Solving

1. Depict what is left on stacks after the following series of push and pop operations (starting with the first column of operations and continuing with the second). Assume that the stack is initially empty.

```
push(s,10)        push(s,50)
push(s,20)        push(s,60)
push(s,40)        push(s,80)
pop(s)            pop(s)
pop(s)            pop(s)
```