Python Modules

What Is a Python Module?

A Python module is a file containing Python definitions and statements. When a Python file is directly executed, it is considered the *main module* of a program. Main modules are given the special name `main` . Main modules provide the basis for a complete Python program. They may *import* (include) any number of other modules (and each of those modules import other modules, etc.). Main modules are not meant to be imported into other modules.

As with the main module, imported modules may contain a set of statements. The statements of imported modules are executed only once, the first time that the module is imported. The purpose of these statements is to perform any initialization needed for the members of the imported module. The Python Standard Library contains a set of predefined *Standard (built-in) modules*. We have in fact seen some of these modules already, such as the `math` and `random` Standard Library modules.

Python modules provide all the benefits of modular software design we have discussed. By convention, modules are named using all lower-case letters and optional underscore characters. We will look more closely at Python modules in the next section.

Your Turn

Create a Python module by entering the following in a file name `simple.py`. Then execute the instructions in the Python shell as shown and observe the results.

```
# module simple
print('module simple loaded')

def func1():
   print('func1 called')

def func2():
```

```
>>> import simple
???
>>>simple.func1()
???

>>>simple.func2()
???
```

Part II - Modules and Namespaces

A **namespace** is a container that provides a named context for a set of identifiers. Namespaces enable programs to avoid potential *name clashes* by associating each identifier with the

namespace from which it originates. In software development, a **name clash** is when two otherwise distinct entities with the same name become part of the same scope. Name clashes can occur, for example, if two or more Python modules contain identifiers with the same name and are imported into the same program, as shown below:

```
# module1

def double(lst):

    """Returns a new list with each
       number doubled, for example,
       [1, 2, 3] returned as [2, 4, 6]
    """
```

```
# module2

def double(lst):

    """Returns a new list with each
       number duplicated, for example,
       [1, 2, 3] returned as
       [(1, 1), (2, 2), (3, 3)]
    """
```

```
import module1
import module2

# main
   .
  num_list = [3, 8, 14]
  result = double(num_list)
   .
```

ambiguous reference for identifier `double`

In this example, `module1` and `module2` are imported into the same program. Each module contains an identifier named `double`, which return very different results. When the function call `double(num_list)` is executed in main, there is a name clash. Thus, it cannot be determined which of these two functions should be called. Namespaces provide a means for resolving such problems.

In Python, each module has its own namespace. This includes the names of all items in the module, including functions and global variables—variables defined within the module and outside the scope of any of its functions. Thus, two instances of identifier `double`, each defined in their own module, are distinguished by being *fully qualified* with the name of the module in which each is defined: `module1.double` and `module2.double`. The figure below illustrates the use of fully qualified identifiers for calls to function `double`.

```
import module1
import module2

# main
   ans1 = module1.double(...)

   ans2 = module2.double(...)
```

references function double from module1's namespace

references function double from module2's namespace

The use of namespaces to resolve problems associated with duplicate naming is not restricted to computer programming. In fact, it occurs in everyday situations. Imagine, for instance, that you run into a friend who tells you that "Paul is getting married." In fact, you have two friends in common named Paul. Because you are not certain which Paul your friend is referring to, you may re- spond "Paul from back home, or Paul from the dorm?" In this case, you are asking your friend to respond with a fully qualified name to resolve the ambiguity: "`home:Paul`" vs. "`dorm:Paul`."

Your Turn

Enter each of the following functions in their own modules named `mod1.py` and `mod2.py`. Enter and execute the following and observe the results.

```
# mod1
def average(lst):
    print('average of mod1 called')
```

```
# mod2
def average(lst):
    print('average of mod2 called')
```

```
>>> import mod1, mod2
>>>mod1.average([10, 20, 30])
   ???

 >>>mod2.average([10, 20, 30])
  ???
  >>>average([10, 20, 30])
                          ???
```

Part III - Importing Modules

In Python, the **main module** of any program is the first ("top-level") module executed. When working interactively in the Python shell, the Python interpreter functions as the main module, containing the *global namespace*. The namespace is reset every time the interpreter is started (or when selecting `Shell → Restart Shell`. Next we look at various means of importing modules in Python. (We note that module `builtins` is automatically imported in Python programs, providing all the built-in constants, functions, and classes.)

The "import modulename" Form of Import

When using the `import` *`modulename`* form of import, the namespace of the imported module becomes *available to*, but not *part of*, the importing module. Identifiers of the imported module, therefore, must be fully qualified (prefixed with the module's name) when accessed. Using this form of import prevents any possibility of a name clash. Thus, as we have seen, if two modules, `module1` and `module2`, both have the same identifier, `identifier1`, then `module1.identifier1` denotes the entity of the first module and `module2.identifier1` denotes the entity of the second module.

Your Turn

Enter the following into the Python shell and observe the results.

```
>>> factorial(5)
???

>>>math.factorial(5)
???

???
```

```
>>>import math
>>>factorial(5)
 ???

>>>math.factorial(5)
```

Part IV - The “from-import” Form of Import

Python also provides an alternate import statement of the form

from *modulename* **import** *something*

where *something* can be a list of identifiers, a single renamed identifier, or an asterisk, as shown below,

```
(a) from modulename import func1, func2
(b) from modulename import func1 as new_func1
(c) from modulename import *
```

In example (a), only identifiers func1 and func2 are imported. In example (b), only identifier `func1` is imported, renamed as `new_func1` in the importing module. Finally, in example (c), *all* of the identifiers are imported, except for those that begin with two underscore characters, which are meant to be private in the module.
There is a fundamental difference between the `from` *`modulename`* `import` and `import` *`modulename`* forms of import in Python. When using `import` *`modulename`*, the namespace of the imported module does not become part of the namespace of the importing module, as mentioned. Therefore, identifiers of the imported module must be fully qualified (e.g., *`modulename`*`.func1`) in the importing module. In contrast, when using `from-import`, the imported module’s namespace becomes part of the importing module’s namespace. Thus, imported identifiers are referenced without being fully qualified (e.g., `func1`).

The `from` *`modulename`* `import func1 as new_func1` form of import is used when identifiers in the imported module’s namespace are known to be identical to identifiers of the importing module. In such cases, the renamed imported function can be used without needing to be fully qualified. Finally, using the `from` *`modulename`* `import *` form of import in example (c), although convenient, makes name clashes more likely. This is because the names of the imported identifiers are not explicitly listed in the import statement, creating a greater

chance that the programmer will unintentionally define an identifier with the same name as in the importing module. And since the `from-import` form of import allows imported identifiers to be accessed without being fully qualified, it is unclear in the importing module where these identifiers come from. Here is an example of this.

```
# module somemodule

def func1(n):
  return n * 10

def func2(n1, n2):
  return n1 * n2
```

```
# ---- main

from somemodule import *

# NAMESPACES AND FUNCTION USE

def func2(n):
    return n * n            # definition of func2 masks imported func2

print(func1(8))             # outputs 80 (func1 of somemodule called)
print(somemodule.func1(8))  # NameError: name 'somemodule' is not defined

print(func2(5))             # outputs 25 (func2 of MAIN called)
print(func2(3, 8))          # TypeError: func2() takes exactly 1 argument

print(somemodule.func2(3, 8)) # NameError: name 'somemodule' is not
                                defined
```

Module `somemodule` contains functions `func1` and `func2`. Since `somemodule` is imported with `from somemodule import *`, identifiers `func1` and `func2` become part of the main module's namespace. However, since the module's namespace already contains identifier `func2` (denoting the function defined there), access to `func2` of `somemodule` is *masked*, and therefore is inaccessible. Using the fully qualified form `somemodule.func2` does not work either, since `somemodule` is not part of the imported namespace for this form of import.

Finally, it is recommended Python style that standard modules be imported before the programmer-defined ones, with each section of imports separated by a blank line as shown below.

```
import standardmodule1          # standard modules
import standardmodule2

import somemodule1             # programmer-defined modules
import somemodule2
```

Your Turn

Enter the definition of function `ordered` given above into the Python Shell. Then enter the following and observe the results.

```
>>> from math import factorial
>>> factorial(5)
???

>>> def factorial(n):
        print ('my factorial')

>>> factorial(5)
???

>>> math.factorial(5)
???
```

```
>>>from math import factorial as fact
>>> fact(5)
???

>>> def factorial(n):
        print ('my factorial')

>>> factorial(5)
???

>>> fact(5)
???
```

Concepts and Procedures

1. By convention, variables names in a module beginning with two_____ characters are meant to be treated as private variables of the module.

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

3. The three active namespaces that may exist during the execution of any given Python program are the_____ ,_____ and_____ namespaces.

Problem Solving

1. For `module1`, `module2`, and the client module shown below, indicate which of the imported identifiers would result in a name clash if the imported identifiers were not fully qualified.

```
# module1

def func_1(n):
    etc.

def func_2(n):
    etc.
```

```
# module2

def func_2(n):
    etc.

def func_3(n):
    etc.
```

```
from module1 import *
from module2 import *

def func_3(n):
    etc.
```

2. For the program in Figure 7-9 that imports modules `module1` and `module1`, indicate how many total namespaces exist for this program.

3. For the Palindrome Checker program in section 7.3.7, describe the changes that would be needed in the program if the import statement were changed from `import Stack` to `from Stack import *`.

4. For the following program and the imported modules, describe any name clashes that would occur for both program version1 and version 2.

```
# module m1

def total(items):

def convert(items):

def show(items)
```

```
# module m2

def totalSum(items):

def convert(items):

def display(items)
```

```
from m1 import *
from m2 import *

def display()

def calc()

def getItems()

# ---- main

items = getItems()
items = convert(items)
show(items)
```

Version 1

```
import m1
import m2

def display()

def calc()

def getItems()

# ---- main

items = getItems()
items = m2.convert(items)
display(items)
```

Version 2