PYTHON MODULES

A module is a file containing Python definitions and statements. A module can define functions, classes and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. Modules make life easier for programmers by allowing code reuse. Program code making use of a given module is called a client of the module. Please take note, the total size of the program remains the same regardless of whether modules are used or not. Modules simply divide the program

Let's take for example, we want the program to greet the person every time, so we can create a function called greeting and save the code as mymodule. Main modules are not meant to be imported into other modules.

The *import* statement

We can use any Python source file as a module by executing an import statement in some other Python source file. When interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches for importing a module.

Code the following program save and start a new python file within the same folder

```
def hi(name):
  print("Hello, " + name,".", "How are you today")
```

Now, you can start a new program and type the following:

```
import mymodule
mymodule.hi("Aden")
```

When using a function from a module, use the syntax: module_name.function_name. In the example above, we use mymodule.hi("Aden"). Mymodule was the python name of the module created earlier, hi was the name of the function.

Modules can contain variables of all type.

Let's us add an array of attributes to object person as below:

Person={"name": "John", "Age": 36, "Country":"New Zealand"}

Note, you can use alias fro modules when calling them e.g

For mymodule you can use mygreetings.When calling the module, you can type as below:

import mymodule as mygreetings:

```
x=mygreeting.person1[“country”]

print(x)
```

The output is still going to be the same—Norway.

The from...import * Statement

It is also possible to import all names from a module into the current namespace by using the following import statement −

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

LOCATING MODULES

When you import a module, the Python interpreter searches for the module in the following sequences −

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

BUILT-IN MODULES

There are several built-in modules in Python, which you can import whenever you like. For instance, one can use platform as shown below.

```
import platform
x = platform.system()
print(x)
```

Using the dir() Function

The dir() built-in function returns a sorted list of strings containing the names defined by a module. The list contains the names of all the modules, variables and functions that are defined in a module.import platform

# Import built-in module random

```
import  random
print  dir(random)
```

NAMESPACES AND SCOPING

Variables are names (identifiers) that map to objects. A namespace is a dictionary of variable names (keys) and their corresponding objects (values). A Python statement can access variables in a local namespace and in the global namespace. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions. Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must first use the global statement. The statement global VarName tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable Money in the global namespace. Within the function Money, we assign Money a value, therefore Python assumes Money as a local variable. However, we accessed the value of the local variable Money before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem. See code below:

```
Money = 2000
def AddMoney():
   # Uncomment the following line to fix the code:
   # global Money
   Money = Money + 1

print Money
AddMoney()
print Money
```