PES UNIVERSITY

Introduction to Modules

- At the end of this class, students will be able to-
  - Understand Module interface
  - Module Specification

Modules

- A module is a file containing Python definitions and statements. A module can define functions, classes and variables.
- Consider a module to be the same as a code library.
- A file containing a set of functions you want to include in your application.

Module Specification

Every module needs to provide a *specification* of how it is used. This is referred to as the module's **interface**.

Any program code making use of a given module is referred to as a **client** of the module.

A module's specification should be sufficiently *clear* and *complete* so that its clients can effectively utilize it. For example, `numPrimes` is a function that returns the number of primes in a given integer range.

```
def numPrimes(start, end):
    """ Returns the number of primes between start and end. """
```

The function's specification is provided by the line immediately following the function header, called a *docstring* in Python. A **docstring** is a string literal denoted by triple quotes given as the first line of certain program elements.

The docstring of a particular program element can be displayed by use of the **__doc__** extension.

```
>>> print(numPrimes.__doc__)
Returns the number of primes between start and end.
```

This provides a convenient way for discovering how to use a particular function without having to look at the function definition itself. Some software development tools also make use of docstrings.

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 may 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 in Python are named using all lower case letters and optional underscore characters.

We look more closely at Python modules.

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.

```
# 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`

Example of a Name Clash

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.

The 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`**.

```
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

Example Use of Fully Identified Function Names

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 respond "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."

Next, we look at different ways that Python modules can be imported.

Importing Modules

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 restarted. 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.

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.

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

In **(a)**, only identifiers **func1** and **func2** are imported

In **(b)**, only identifier **func1** is imported, renamed as **new_func1** in the importing module.

In **(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, which will be discussed later.

There is a fundamental difference between the

```
from modulename import ident1, ident2, ...
```

and

```
import modulename
```

forms of import in Python.

Create a Module

- def greeting(name):
  print("Hello, " + name)

- Save this code in a file named mymodule.py

Use a Module

- Import the module named mymodule, and call the greeting function:

```
import mymodule

  mymodule.greeting(“HI")
```

Re-naming a Module

- You can create an alias when you import a module, by using the as keyword:

```
import module as m

name=input('enter your name')

m.example(name)
```

```
def example(name):
    print("hello",name)
```

Saved as module.py

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.
- import statement access the definitions inside it using the dot operator.

The *from import* Statement

- We can import specific names from a module without importing the module as a whole. Here is an example.

```
from module import example
name=input('enter your name')
example(name)
```

Import all names

- We can import all names(definitions) from a module using the following construct.

```
from module import *
name=input('enter your name')
example(name)
```

Module Private Variables

- In Python, all identifiers in a module are "public"—that is, accessible by any other module that imports it.

- Sometimes 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.

__name__

- **__name__ is a built-in variable which evaluates to the name of the current module.**
- If the source file is executed as the main program, the interpreter sets the __name__ variable to have a value “__main__”. If this file is being imported from another module, __name__ will be set to the module’s name.

Check file is imported or running directly

```
print ("File1 __name__ = " ,__name__ )

if __name__ == "__main__":
    print( "File1 is being run directly")
else:
    print ("File1 is being imported")
```

```
def area1(n):
        print("area of Circle=",3.14*n*n)
def carea(n):
        print("cir of a circle=",2*3.14*n)
def tarea(a,b):
        print("Area of triangle",.5*a*b)

if __name__=='__main__':
        print("without import")
else:
        print("with import")
```

__call__

- Python runs a **__call__** method when an instance is called as a function form.

```
class my:


        def __str__(self):
                return "this is call"
        def __call__(self,*x,**y):
                print(x,y)
person=my()
print(person)
person("hi","hello",a=1,b=2)
```

```
import area
x=int(input("enter the number"))
area.area1(x)
area.carea(x)
```

Module Loading and Execution

Each imported module needs to be located and loaded into memory. Python searches for modules in the following order:

- the current directory
- the directories listed in the PYTHONPATH environment var
- a Python installation-specific directory (e.g., C:\Python32\Lib)

If still not found, 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 environment variable, 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. 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.

```
a=input("enter the number")
import mod1
print(area(a))
'ModuleNotFoundError: No module named 'mod1
```

```
a=int(input("enter the number"))
import sys
print(sys.path)
sys.path.insert(0,r'C:\Users\monika\Desktop\home')
import mod1
print(mod1.area(a))
```