PES UNIVERSITY

Department of Computer Science and Engineering (UG Studies)

PES University, Bangalore, India

Introduction to Computing using Python (UE19CS101)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Create and Access a Python Package

Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or **modules**. Individual modules can then be cobbled together like building blocks to create a larger application.

There are several advantages to **modularizing** code in a large application:

- **Simplicity:** Rather than focusing on the entire problem at hand, a module typically focuses on one relatively small portion of the problem. If you're working on a single module, you'll have a smaller problem domain to wrap your head around. This makes development easier and less error-prone.
- **Maintainability:** Modules are typically designed so that they enforce logical boundaries between different problem domains. If modules are written in a way that minimizes interdependency, there is decreased likelihood that modifications to a single module will have an impact on other parts of the program. (You may even be able to make changes to a module without having any knowledge of the application outside that module.) This makes it more viable for a team of many programmers to work collaboratively on a large application.
- **Reusability:** Functionality defined in a single module can be easily reused (through an appropriately defined interface) by other parts of the application. This eliminates the need to recreate duplicate code.
- **Scoping:** Modules typically define a separate **namespace**, which helps avoid collisions between identifiers in different areas of a program.

Functions, **modules** and **packages** are all constructs in Python that promote code modularization.

Packages are a way of structuring many packages and modules which helps in a well-organized hierarchy of data set, making the directories and modules easy to access. Just like there are different drives and folders in

an OS to help us store files, similarly packages help us in storing other sub-packages and modules, so that it can be used by the user when necessary.

Creating and Exploring Packages

To tell Python that a particular directory is a package, we create a file named **__init__.py** inside it and then it is considered as a package and we may create other modules and sub-packages within it. This **__init__.py** file can be left blank or can be coded with the initialization code for the package.

To create a package in Python, we need to follow these three simple steps:

1. First, we create a directory and give it a package name, preferably related to its operation.
2. Then we put the classes and the required functions in it.
3. Finally we create an __init__.py file inside the directory, to let Python know that the directory is a package.

Example of Creating Package

Let's look at this example and see how a package is created. Let's create a package named Cars and build three modules in it namely, Bmw, Audi and Nissan.

1. **First we create a directory and name it Cars.**

2. **Then we need to create modules**. To do this we need to create a file with the name **Bmw.py** and create its content by putting this code into it.

```
# Python code to illustrate the Modules
class Bmw:
    # First we create a constructor for this class
    # and add members to it, here models
    def __init__(self):
        self.models = ['i8', 'x1', 'x5', 'x6']

    # A normal print function
    def outModels(self):
        print('These are the available models for BMW')
        for model in self.models:
            print('\t%s ' % model)
```

Then we create another file with the name **Audi.py** and add the similar type of code to it with different members.

```
# Python code to illustrate the Module
class Audi:
    # First we create a constructor for this class
    # and add members to it, here models
    def __init__(self):
        self.models = ['q7', 'a6', 'a8', 'a3']
```

```
    # A normal print function
    def outModels(self):
        print('These are the available models for Audi')
        for model in self.models:
            print('\t%s ' % model)
```

Then we create another file with the name **Nissan.py** and add the similar type of code to it with different members.

```
# Python code to illustrate the Module
class Nissan:
    # First we create a constructor for this class
    # and add members to it, here models
    def __init__(self):
        self.models = ['altima', '370z', 'cube', 'rogue']

    # A normal print function
    def outModels(self):
        print('These are the available models for Nissan')
        for model in self.models:
            print('\t%s ' % model)
```

3. **Finally we create the __init__.py file.** This file will be placed inside Cars directory and can be left blank or we can put this initialisation code into it.

```
from Bmw import Bmw
from Audi import Audi
from Nissan import Nissan
```

Now, let's use the package that we created. To do this make a **sample.py** file in the same directory where Cars package is located and add the following code to it:

```
# Import classes from your brand new package
from Cars import Bmw
from Cars import Audi
from Cars import Nissan

# Create an object of Bmw class & call its method
ModBMW = Bmw()
ModBMW.outModels()

# Create an object of Audi class & call its method
ModAudi = Audi()
ModAudi.outModels()

# Create an object of Nissan class & call its method
ModNissan = Nissan()
ModNissan.outModels()
```

Various ways of Accessing the Packages

Let's look at this example and try to relate packages with it and how can we access it.

1. ## import in Packages

   Suppose the cars and the brand directories are packages. For them to be a package they all must contain __init__.py file in them, either blank or with some initialization code. Let's assume that all the models of the cars to be modules. Use of packages helps importing any modules, individually or whole.

   Suppose we want to get Bmw x5. The syntax for that would be:

```
'import' Cars.Bmw.x5
```

   While importing a package or sub packages or modules, Python searches the whole tree of directories looking for the particular package and proceeds systematically as programmed by the dot operator.

   If any module contains a function and we want to import that. For e.g., a8 has a function get_buy(1) and we want to import that, the syntax would be:

```
import Cars.Audi.a8

Cars.Audi.a8.get_buy(1)
```

   While using just the import syntax, one must keep in mind that the last attribute must be a subpackage or a module, it should not be any function or class name.

2. ‘from…import’ in Packages

Now, whenever we require using such function we would need to write the whole long line after importing the parent package. To get through this in a simpler way we use ‘from’ keyword. For this we first need to bring in the module using ‘from’ and ‘import’:

```
from Cars.Audi import a8
```

Now we can call the function anywhere using

```
a8.get_buy(1)
```

There’s also another way which is less lengthy. We can directly import the function and use it wherever necessary. First import it using:

```
from Cars.Audi.a8 import get_buy
```

Now call the function from anywhere:

```
get_buy(1)
```

3. ‘from…import *’ in Packages

While using the **from…import** syntax, we can import anything from submodules to class or function or variable, defined in the same module. If the mentioned attribute in the import part is not defined in the package then the compiler throws an ImportError exception.

Importing sub-modules might cause unwanted side-effects that happens while importing sub-modules explicitly. Thus we can import various modules at a single time using * syntax. The syntax is:

```
from Cars.Chevrolet import *
```

This will import everything i.e., modules, sub-modules, function, classes, from the sub-package.