PYTHON PACKAGES

What are Packages?

We don't usually store all of our files in our computer in the same location. We use a well-organized hierarchy of directories for easier access.

Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.

As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.

Similar, as a directory can contain sub-directories and files, a Python package can have sub-packages and modules.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)
```

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

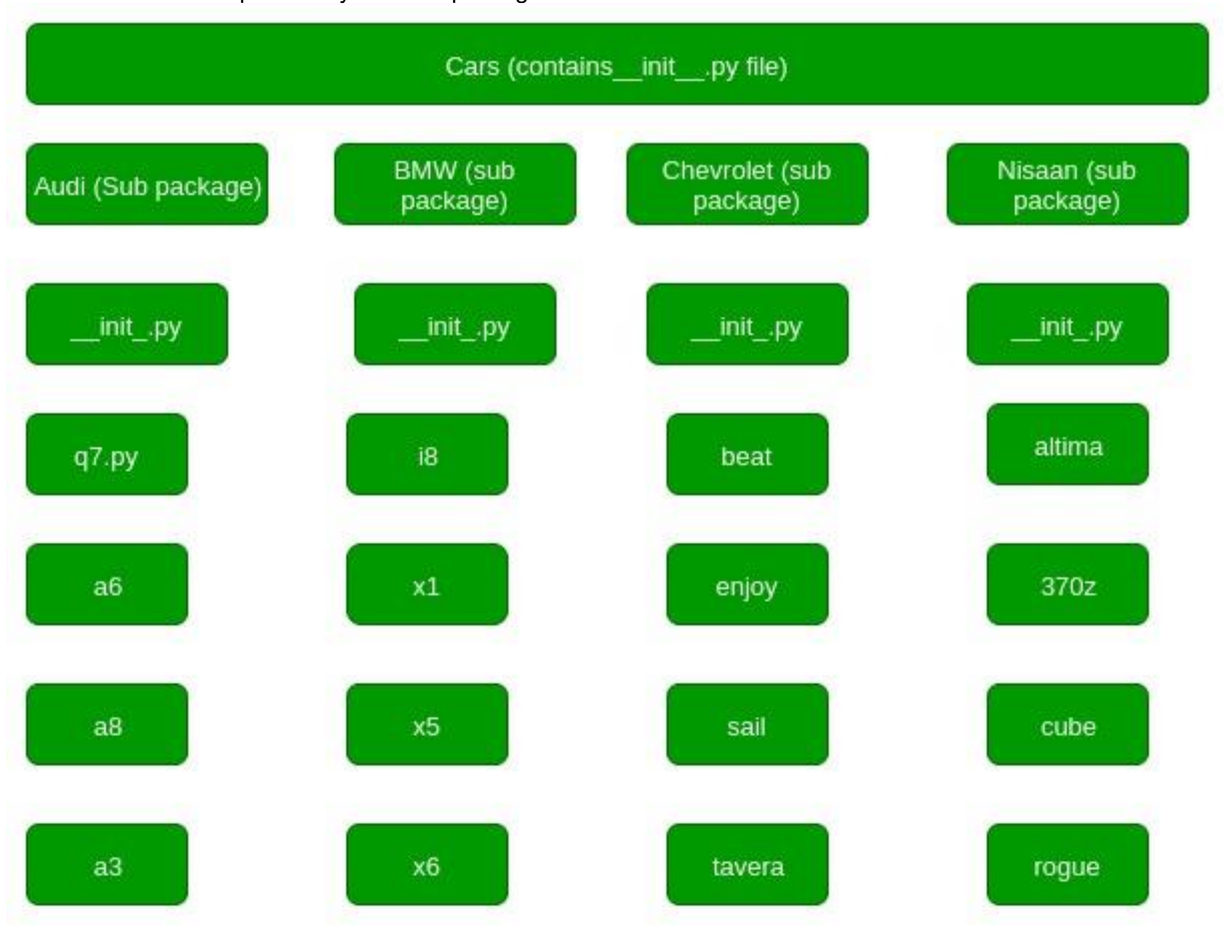
```
from Bmw import Bmw
from Audi import Audi
from Nissan import Nissan
```

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 i8. 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':

```
1.  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)
```

2. **'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.

The *time* Module-

Please refer this link- https://www.programiz.com/python-programming/time

There is a popular **time** module available in Python which provides functions for working with times and for converting between representations. Here is the list of all available methods −

Sr.No.	Function with Description
1	time.altzone The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.
2	time.asctime([tupletime]) Accepts a time-tuple and returns a readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.
3	time.clock( ) Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time().
4	time.ctime([secs]) Like asctime(localtime(secs)) and without arguments is like asctime( )
5	time.gmtime([secs]) Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm_isdst is always 0
6	time.localtime([secs]) Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules).
7	time.mktime(tupletime) Accepts an instant expressed as a time-tuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch.
8	time.sleep(secs) Suspends the calling thread for secs seconds.
9	time.strftime(fmt[,tupletime]) Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.
10	time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')

	Parses str according to format string fmt and returns the instant in time-tuple format.
11	time.time( ) Returns the current time instant, a floating-point number of seconds since the epoch.
12	time.tzset() Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.

Let us go through the functions briefly −

There are following two important attributes available with time module −

Sr.No.	Attribute with Description
1	**time.timezone** Attribute time.timezone is the offset in seconds of the local time zone (without DST) from UTC (>0 in the Americas; <=0 in most of Europe, Asia, Africa).
2	**time.tzname** Attribute time.tzname is a pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively.

The *calendar* Module

Refer this link also- https://www.geeksforgeeks.org/python-calendar-module/

The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.

By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call calendar.setfirstweekday () function.

Here is a list of functions available with the *calendar* module −

Sr.No.	Function with Description
1	**calendar.calendar(year,w=2,l=1,c=6)** Returns a multiline string with a calendar for year year formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length 21*w+18+2*c. l is the number of lines for each week.
2	**calendar.firstweekday( )** Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, this is 0, meaning Monday.

3	**calendar.isleap(year)** Returns True if year is a leap year; otherwise, False.
4	**calendar.leapdays(y1,y2)** Returns the total number of leap days in the years within range(y1,y2).
5	**calendar.month(year,month,w=2,l=1)** Returns a multiline string with a calendar for month month of year year, one line per week plus two header lines. w is the width in characters of each date; each line has length 7*w+6. l is the number of lines for each week.
6	**calendar.monthcalendar(year,month)** Returns a list of lists of ints. Each sublist denotes a week. Days outside month month of year year are set to 0; days within the month are set to their day-of-month, 1 and up.
7	**calendar.monthrange(year,month)** Returns two integers. The first one is the code of the weekday for the first day of the month month in year year; the second one is the number of days in the month. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 to 12.
8	**calendar.prcal(year,w=2,l=1,c=6)** Like print calendar.calendar(year,w,l,c).
9	**calendar.prmonth(year,month,w=2,l=1)** Like print calendar.month(year,month,w,l).
10	**calendar.setfirstweekday(weekday)** Sets the first day of each week to weekday code weekday. Weekday codes are 0 (Monday) to 6 (Sunday).
11	**calendar.timegm(tupletime)** The inverse of time.gmtime: accepts a time instant in time-tuple form and returns the same instant as a floating-point number of seconds since the epoch.
12	**calendar.weekday(year,month,day)** Returns the weekday code for the given date. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (January) to 12 (December).

The Datetime module-

Please refer these link-

https://www.geeksforgeeks.org/python-datetime-module-with-examples/

https://www.programiz.com/python-programming/datetime