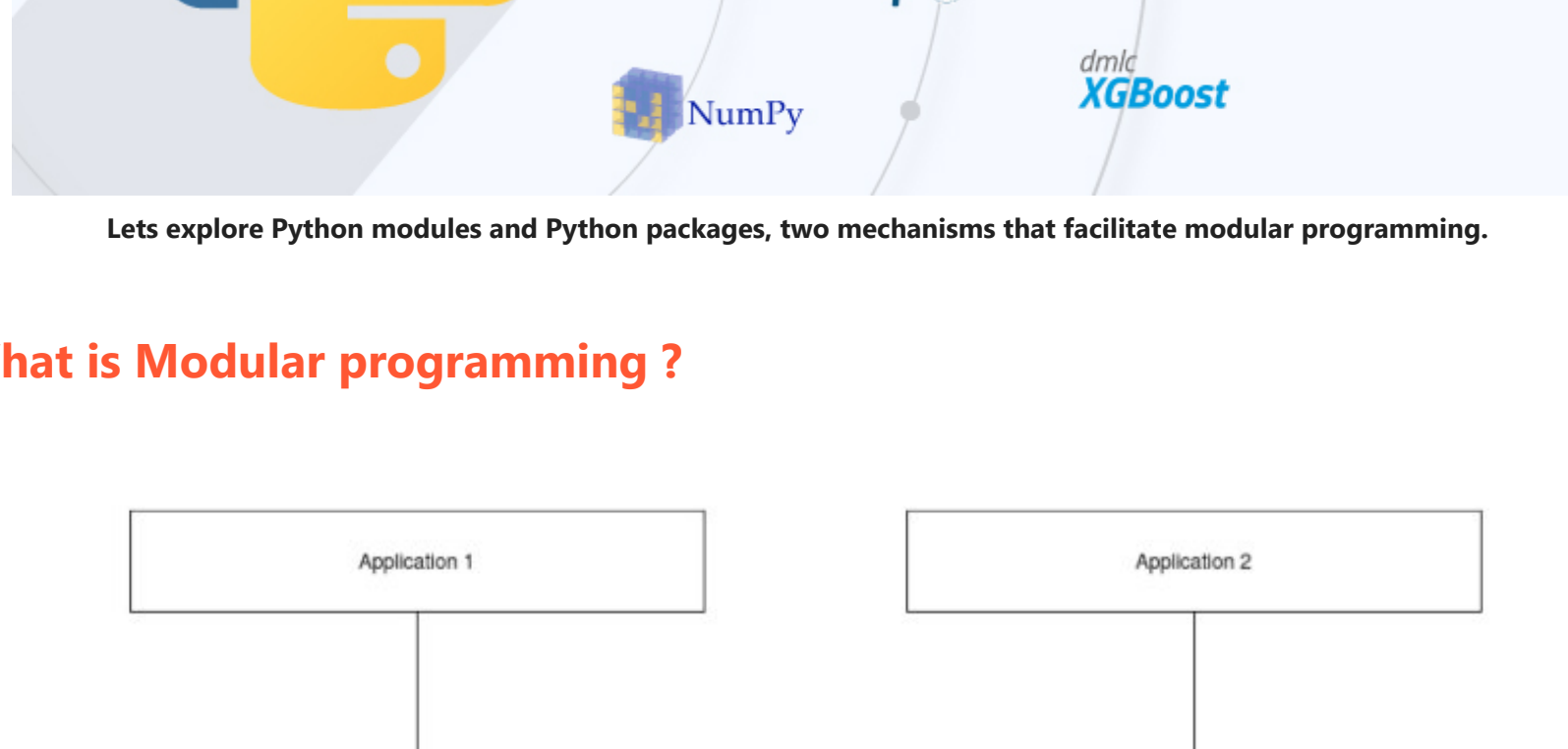
Packages and Modules in Python

Lets explore Python modules and Python packages, two mechanisms that facilitate modular programming.

What is Modular programming ?



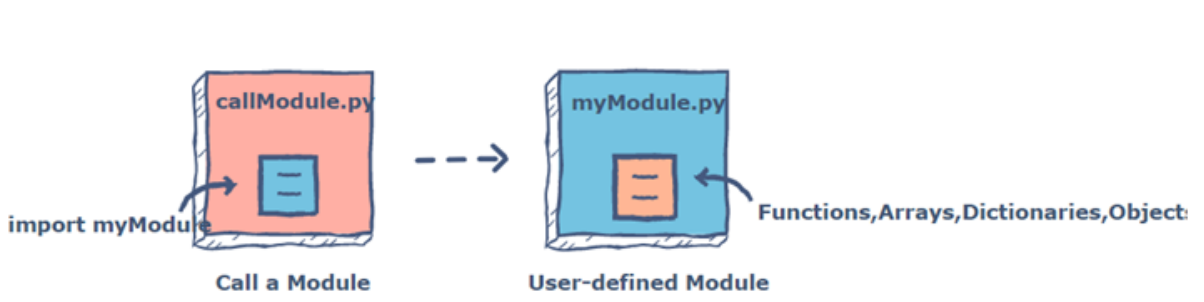
- It is a process of subdividing a computer program into separate, independent and interchangeable sub-programs (modules).
- Similar functions are grouped in the same module which improves e.g. the testability, maintainability and reusability by other applications.
- Individual modules can then be cobbled together like building blocks to create a larger application..

Some of the advantages of modular programming :

- Allow multiple developers to contribute to your project effectively.
- Modular programming makes testing and debugging more efficient.
- Allows you to focus on a small part of problem at a time without disturbing the whole.
- Organize the code and maintain the project a lot more easier.
- Developed modules can be re-used for other applications.

What is a Module ?

- A module is basically a bunch of related code saved in a file with the extension .py.
- It often defines members like classes, functions, and variables intended to be used in other files that import it.
- It's also fine to include runnable code in modules.
- There are actually three different ways to define a module in Python :
  - A user defined module that can be written in Python itself.
  - A module can be written in C and loaded dynamically at run-time, like the re (regular expression) module.
  - A built-in module is intrinsically contained in the interpreter, like the itertools module.



- To use the functionality of the module to another program, we must import the specific module.
- Which is done the same way in all three cases using the following ways:
  - The import statement.
  - The from-import statement.

The import statement

The import statement is used to load all the functionality of one module into another

```
import module 1[, module 2[,... module n]
```

In [1]:
```python
# Import the mod.py file as a 'mod' module
import mod
'''
Several objects are defined in mod.py:
s (a string)
a (a list)
foo() (a function)
Foo (a class)
'''
type(mod)
```

Out[1]: module

In [2]:
```python
#Acessing string
print(mod.s)
```

```
If Comrade Napoleon says it, it must be right.
```

In [3]:
```python
#Acessing the list
print(mod.a)
```

```
[100, 200, 300]
```

In [4]:
```python
#acessing the function
mod.foo(['quux', 'corge', 'grault'])
```

```
arg = ['quux', 'corge', 'grault']
```

In [5]:
```python
#acessing the class
x = mod.Foo()
x
```

Out[5]: <mod.Foo at 0x204a1954f70>

The from-import statement

A module can contain various attributes, and if we require a specific attribute of a module, then it can be done by the from-import statement.

```
from module import name 1[, name 2[, ... name n]]
```

In [6]:
```python
# Import a class from the file
from cars import Car
```

In [7]:
```python
# object for car
car1 = Car(make="Toyota", model="Camry")

# Start driving
car1.start()

#stop the car
car1.stop()

#check the speed
car1.check_speed_and_gear()
```

```
...VROOOOM....Started!
I'm driving at: 0 in gear: 0
```

What exactly happens when we write an import statement ?

- The python interpreter tries to look for the directory containing the module we are trying to import in sys.path
- It is a list of directories that Python will search once it is done looking at the cached modules and Python standard library modules.

Let's see what our system path contains at this very moment

In [8]:
```python
import sys
sys.path
```

Out[8]:
```
['C:\\Users\\nithe\\Desktop\\TNR\\Data Science\\Course Bundle\\01 - Python\\12 - Packages and Modules',
 'C:\\Users\\nithe\\anaconda3\\python38.zip',
 'C:\\Users\\nithe\\anaconda3\\DLLs',
 'C:\\Users\\nithe\\anaconda3\\lib',
 'C:\\Users\\nithe\\anaconda3',
 '',
 'C:\\Users\\nithe\\anaconda3\\lib\\site-packages',
 'C:\\Users\\nithe\\anaconda3\\lib\\site-packages\\win32',
 'C:\\Users\\nithe\\anaconda3\\lib\\site-packages\\win32\\lib',
 'C:\\Users\\nithe\\anaconda3\\lib\\site-packages\\Pythonwin',
 'C:\\Users\\nithe\\anaconda3\\lib\\site-packages\\IPython\\extensions',
 'C:\\Users\\nithe\\.ipython']
```

- The output from sys.path will always contain the current directory at index 0!.
- This is the reason importing is fairly straightforward when both the caller and callee modules reside within the same directory.
- The next thing is The list of directories contained in the PYTHONPATH environment variable
- PYTHONPATH is an environment variable which you can set to add additional directories where python will look for modules and packages
- An installation-dependent list of directories configured at the time Python is installed
- Thus, to ensure your module is found, you need to do one of the following :
  - Put .py file in the directory where the input script is located or the current directory, if interactive
  - Modify the PYTHONPATH environment variable to contain the directory where .py file is located before starting the interpreter

    Or: Put .py file in one of the directories already contained in the PYTHONPATH variable
  - Put .py file in one of the installation-dependent directories, which you may or may not have write-access to, depending on the OS

There is actually one additional option:

We can put the module file in any directory of your choice and then modify sys.path at run-time so that it contains that directory

In [9]:
```python
#lets put our mod.py in other location and see
import sys
sys.path.append(r"C:\Users\nithe\mods.py")
sys.path
```

Out[9]:
```
['C:\\Users\\nithe\\Desktop\\TNR\\Data Science\\Course Bundle\\01 - Python\\12 - Packages and Modules',
 'C:\\Users\\nithe\\anaconda3\\python38.zip',
 'C:\\Users\\nithe\\anaconda3\\DLLs',
 'C:\\Users\\nithe\\anaconda3\\lib',
 'C:\\Users\\nithe\\anaconda3',
 '',
 'C:\\Users\\nithe\\anaconda3\\lib\\site-packages',
 'C:\\Users\\nithe\\anaconda3\\lib\\site-packages\\win32',
 'C:\\Users\\nithe\\anaconda3\\lib\\site-packages\\win32\\lib',
 'C:\\Users\\nithe\\anaconda3\\lib\\site-packages\\Pythonwin',
 'C:\\Users\\nithe\\anaconda3\\lib\\site-packages\\IPython\\extensions',
 'C:\\Users\\nithe\\.ipython',
 'C:\\Users\\nithe\\mods.py']
```

Wouldn't it be great if we could just do import utils

In [10]:
```python
#Restart the kernal
import utils
txt = "Hello"
res = utils.length.get_length(txt)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-10-01d099a78ca2> in <module>
      2 import utils
      3 txt = "Hello"
----> 4 res = utils.length.get_length(txt)

AttributeError: module 'utils' has no attribute 'length'
```

In [11]:
```python
import utils.length
res = utils.length.get_length("Hello")
print("The length of the string is: ",res)
```

```
The length of the string is:  5
```

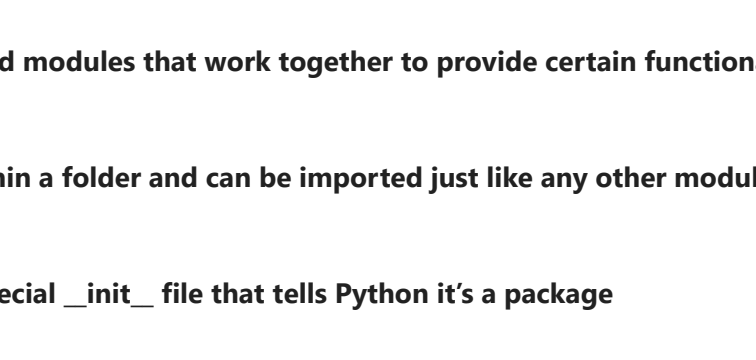
In [12]:
```python
#Importing with alias
import utils.lower as low
res = low.to_lower("HELLO")
print("The lower case of the string is: ",res)
```

```
The lower case of the string is:  hello
```

To achieve single import We can turn this directory into a package by introducing init.py file within utils folder.

What is Packaging ?

- A package is a collection of related modules that work together to provide certain functionality
- These modules are contained within a folder and can be imported just like any other modules.
- This folder will often contain a special __init__ file that tells Python it's a package
- Potentially containing more modules nested within subfolders



In [ ]:
```python
#from lower import to_lower won't workso the sys.path will only have example3_outer.py's current directory
# utils/__init__.py (incorrect way of importing)
from length import get_length
from lower import to_lower
from upper import to_upper
```

In [14]:
```python
#Lets create __init__.py in utils and call the functions
import utils
txt = "Hello"
res_low = utils.to_lower(txt)
print(res_low)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-326b898481b9> in <module>
      2 import utils
      3 txt = "Hello"
----> 4 res_low = utils.to_lower(txt)
      5 print(res_low)

AttributeError: module 'utils' has no attribute 'to_lower'
```

Relative Imports :

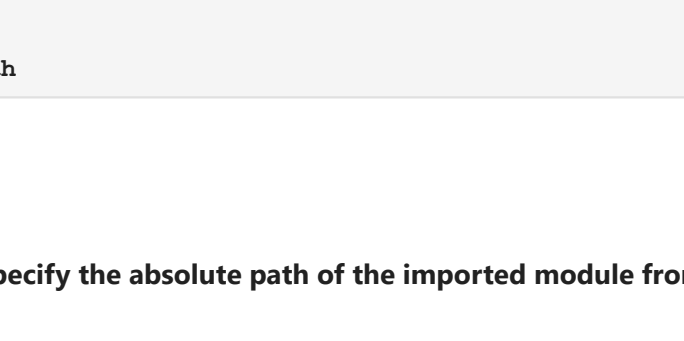
- Relative imports (not recommended): specify the path relative to the path of the calling script.
- We use the dot notation(. or ..) in specifying relative imports
- The single dot before lower refers to the same directory as the one from which the import is called.
- Similarly, double dots before a module name means moving up two levels from the current level.

In [ ]:
```python
# utils/__init__.py
from .lower import to_lower
from .upper import to_upper
from .length import get_length
```

Absolute imports :

Absolute imports (better choice) : specify the absolute path of the imported module from the project root

(or any other dir which sys.path has access to)



In [15]:
```python
# utils/__init__.py
from utils.lower import to_lower
from utils.upper import to_upper
from utils.length import get_length
```

What happens when we import a package with an __init__.py defined?

- This acts as an initialization step and it is the first file to be executed when we import the package
- Given that we do all the necessary imports in here, the code is much cleaner in the calling script

In [19]:
```python
import importlib
importlib.reload(utils)
```

Out[19]:
```
<module 'utils' from 'C:\\Users\\nithe\\Desktop\\TNR\\Data Science\\Course Bundle\\01 - Python\\12 - Packages a
nd Modules\\utils\\__init__.py'>
```

In [20]:
```python
import utils

txt = "Hello"

res_len = utils.get_length(txt)
print(res_len)

res_up = utils.to_upper(txt)
print(res_up)

res_low = utils.to_lower(txt)
print(res_low)
```

```
5
HELLO
hello
```

In [21]:
```python
#lets import from the cars package
from carspackage import cars as carsp
from carspackage import suv as suvp
```

```
__name__:  carspackage.cars
I am outside the guard!
```

In [22]:
```python
car2 = carsp.Car(make="Toyota", model="Camry")
car2
```

Out[22]: Make Toyota, model: Camry

In [23]:
```python
#Importing * From a Package
from carspackage import *
```

In [24]:
```python
#let's try running the SUV
suv1 = suv.SUV(make="Honda", model="CRV")
suv1.start_drive()
suv1.check_speed_and_gear()
```

```
Init success!!
Shift Up and Drive.
I am driving at  5 mph
I'm driving at: 5 in gear: 1
```

Purpose of dunder *main.py*

- Just like how you call a python script in terminal python my_code.py, we can call your package

from command prompt / terminal as well via python {pacakge_name}.

- But when called so, which module / code will be run ?
- That is the purpose of __main__.py
- When a package is called from terminal, the Python will look for executing the contents

of __main__.py file inside the package.

In [ ]:
```python
#Let's call carspackage from terminal / command prompt.
# !python carspackage !
```

Executing a Module as a Script

- A Script is a Python file that's intended to be run directly. When you run it, it should do something.
- When a .py file is imported as a module, Python sets the special dunder variable __name__ to the name of the module.
- However, if a file is run as a standalone script, __name__ is (creatively) set to the string '__main__'
- Using this fact, we can alter behavior accordingly at run-time and can run any module as a script

What does dunder *name == main* do ?

- When you import a python package or module, all the code present in the module is run.
- There is every chance that certain code present in mypackage you didn't want to execute may get executed on import.
- You can prevent this by checking the condition : __name__ == __main__.It acts as a guard.
- The parts of your code that you don't want to be run can be placed inside the condition block.
- If the code is run when getting imported from another package, the value of __name__ will bear the path/name of the module.
- Example : The value of __name__ for 'carspackage/cars.py' when called from other places will be

carspackage.cars

- Only when you are directly running python carspackage/cars.py, that is, only when you run the module

as the main program, the value of __name__ will be __main__.

In [ ]:
```python
#Run in terminal
#!python carspackage/cars.py
```

Packages with hierarchy :

- The package may contains folders(Sub Packages) that contain the python files.
- So, we need to point to that folder and then import the module.

In [4]:
```python
from carspackagedeep.Car import cars
#from carspackagedeep.Car import *
from carspackagedeep.Suv import suv
```

```
__name__:  carspackagedeep.Car.cars
I am outside the guard!
```

Reloading a Module : For reasons of efficiency, a module is only loaded once per interpreter session

In [5]:
```python
#Print Statements are exceuted only once
from carspackagedeep.Suv import suv
```

If you make a change to a module and need to reload it, you need to either restart the interpreter

or use a function called reload( ) from module importlib

In [6]:
```python
import importlib
importlib.reload(suv)
```

```
I am outside the guard!
```

Out[6]:
```
<module 'carspackagedeep.Suv.suv' from 'C:\\Users\\nithe\\Desktop\\TNR\\Data Science\\Course Bundle\\01 - Pytho
n\\12 - Packages and Modules\\carspackagedeep\\Suv\\suv.py'>
```

Script vs Module Vs Packages vs Libraries Vs Framework :

Scripts :

- A script is a single file of python code that is meant to be executed as the 'main' program.

Module :

- A module in python is a ".py file" that defines one or more function/classes which you intend to reuse in different codes of your program.
- Python in-built mdoules : random , datetime , re , html

Package :

- A Package is a collection of various modules with a path attribute.(__init___.py)
- Python in-built mdoules : Numpy , Pandas

Library :

- A library is an umbrella term referring to a reusable chunk of code
- While a package is a collection of modules, a library is a collection of packages.
- Python Standard Libraries : Statastics , Math
- Thrid Party Libraries : Pandas , beautifulsoap , matplotlib , PyTorch

Framework :

- It is a collection of various libraries which architects the code flow.
- Ex - Django : This has various in-built libraries like Auth, user, database connector etc. for web development.
- Python Frameworks : Django , Flask , FastAPI , Bottle

Application :

- If you have a set of code that spans multiple files, you probably have an application instead of script

Packages and Modules FAQ's :

- What are modules ?
- What is a package ?
- What is the special file that each package in Python must contain ?
- What is the difference between module and package in Python ?
- How can I import a module from the different directory ?
- What does __name__ == __main__ mean ?