

**Creating Modules**:

So far we have written all our code in one file, app.py. But this is not how we build real programs. We cannot write thousands of lines of code in a single file.

*It is like going to a supermarket and there is only one section. All products in one section. That is why in supermarkets we have various isles, sections or departments*.

We have the same concept in programming. As our program grows, we should be able to split our code across multiple files.

We refer to each file as a *module*. So *module is a file that contains some python code*.

Now how can we decide , which functions or which classes we should put in what modules.

*For this, Refer back to market. There each section contains highly related products. If you go to fruit section, you will only see fruits, you do not see cleaning products there*.

So, *A module must contain highly related objects*. These objects may be functions, classes, variables and so on.

def calc\_tax():

    pass

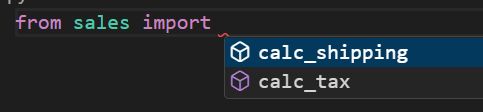
def calc\_shipping():

    pass

Like these two functions both related to concept of sales.

So we can put these two functions in a separate module called *sales*.

Now we can import these functions into our app module.



Notice it is not sales.py just sales in this statement and when we press ctrl + ENTER we can see all the objects defined in this module.

from sales import calc\_shipping

calc\_shipping()

We can call this function same way as we call it in any other place. As simple as that.

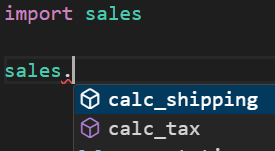
Note: It is a bad practice to import all objects from a module using asterisk like this,

from sales import \*

It will import all the objects and some of the objects might overwrite objects in the current module, which can cause bugs.

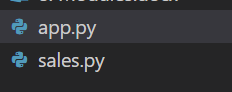
*Another way to import*:

We can import entire module as an object using import statement, then we can access its functions like methods.



**Compiled Python Files**:

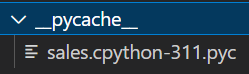
We have two files in our folder sales.py and app.py which is the entry point of our application.



When I run my app.py file using

python3 .\app.py

I get a new folder in my root called \_\_pycache\_\_.



In this folder we have the *compiled version of the modules* that we imported into our program. So currently we have the compiled version of *sales* module.

*The reason python stores these compiled files in this stored folder is to speed up module loading*.

So next time we run our python program, python will look at the content of this folder and if we do have the compiled version of this sales module, python will simply load that compiled version. So it will skip the compilation step.

Note: It only speeds loading of module, not the actual performance.

*How does python know that this compiled version is up to date*?

It basically checks the date – time of these two files, compiled version as well as source code of module. If date-time of source code is newer, it realizes that source code has changed, so it will recompile it.

Note: In the name of the compiled file, *sales.cpython-311.pyc*

311 stands for python version used to compile this file.

Also the content of this file is in *ByteCode*.

*The app module does not have compiled version because python always recompiles the module that we load directly from the command line*.

**Module Search Path**:

When I import sales module like this,

import sales

python will look for the sales.py file inside current directory.

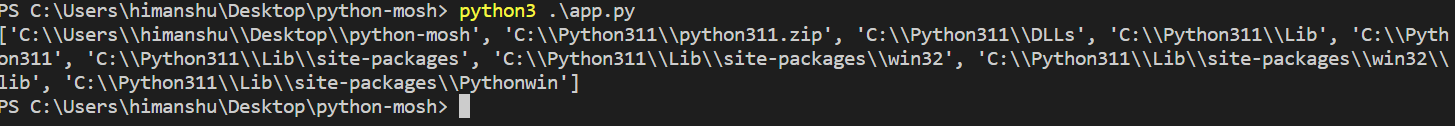
If it does not find the file there, it will look for this in a bunch of predefined directories that come with python installation.

You can use **sys**. *path* to see those directories,

import sys

print(sys.path)

O/P:

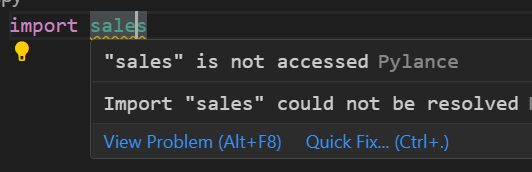


We get an array of strings, notice the first element in this array is my current project directory.

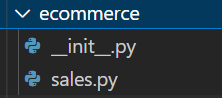
**Packages**:

Currently all our files or modules are in the current folder, but as our application grows, *we need to organize our files into sub – directories*.

So we create a new folder called *ecommerce* and move our sales module in to that, but as soon as we did that we see an error.



So to solve that, add a new file in the *ecommerce* folder called \_\_*init*\_\_.py.



When we add this file, python would treat *ecommerce* folder as a package. So *a package is container for one or more module*.

In file system terms, *a package is mapped to a directory and a module is mapped to a file*.

Now we will prefix the name of *sales* module with name of the package which is *ecommerce*,

import ecommerce.sales

Now to use any method in sales model,

from ecommerce.sales import calc\_shipping

calc\_shipping()

if we have multiple function(*better approach*)

from ecommerce import sales

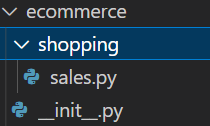
sales.calc\_shipping()

**Sub Packages**:

As our program grows, we might want to break down a package to into sub packages.

For example imagine if our ecommerce package has grown a lot, so that we have multiple files and modules here. We can decide to break this package into few sub packages.

For this, we can create a new folder called *shopping* inside *ecommerce* and move our *sales* module into that.



To convert this shopping directory into a package create a *\_\_init\_\_.py* file here.

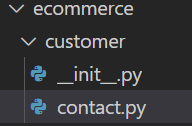
Now to import this package into our app.py

from ecommerce.shopping import sales

**Intra-Package References**:

There are when we want to import modules from sibling packages.

For example here in our ecommerce package, I have added a new sub-package *customer*.



In this package we have a contact module for contacting our customers.

Now let us say in our sales module we want to use contact module within customer package.

To import a module from another package, we can use an absolute or related import statement.

*Using absolute import (preferred approach🡪 recommended by PEP8)*:

Inside sales.py module,

from ecommerce.customer import contact

contact.contact\_customer()

def calc\_tax():

    pass

*Using relative import*: (*use only when import statement gets too verbose*)

Using double dots **(..)** Takes us to one level up from current working directory.

from ..customer import contact

contact.contact\_customer()

def calc\_tax():

    pass

**The dir () function**:

With *dir* function we can *get the list of attributes and methods defined in an object*.

from ecommerce.shopping import sales

Let us apply this function on this imported *sales* object

from ecommerce.shopping import sales

print(dir(sales))

O/P:



We get an array of strings. In this array we have all the attributes and methods defined in an object.

Here *calc\_shipping* and *calc\_tax* are methods defined by us while rest are magic methods automatically created for us.

# return name of the module

print(sales.\_\_name\_\_) # ecommerce.shopping.sales

# return name of the package

print(sales.\_\_package\_\_) #none (why)??

# return name of the file

print(sales.\_\_file\_\_) C:\Users\himanshu\Desktop\python-mosh\ecommerce\shopping\sales.pyc

**Also why I am getting .pyc files**?

**Executing modules as scripts**:

Here in our sales modules, currently we have defined two functions. But *we can also write any statements and these statements will be executed the first time this module is loaded*.

So if we import this module, in a few different modules in our program, python will load this module only once and then cache it in memory. So the statements we write here will be executed once.

#sales module

print("Sales Initialized")

def calc\_tax():

    pass

def calc\_shipping():

    pass

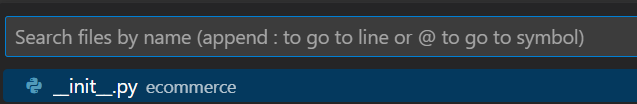
In app module,

from ecommerce.shopping import sales

We can see our sales module initialized



Using the same technique *we can also write the initialization code for our packages*.

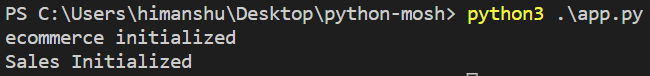


Open \_\_init\_\_.py file in ecommerce package and type

#\_\_init\_\_.py in ecommerce package

print("ecommerce initialized")

Now we first see statement from our ecommerce package then from sales module getting initialized.



Let us take it to next level,

In our sales module, add *\_\_name\_\_* parameter in the print statement to get the name of this module.

print("Sales Initialized", \_\_name\_\_)

We see the name of our module,



However if we run sales.py file instead of app.py,

, we see *\_\_main\_\_*.

Note: *The name of the module that starts our program is always \_\_main\_\_*.

Now we can do something interesting with this, in sales module

#in sales module

def calc\_tax():

    pass

def calc\_shipping():

    pass

if \_\_name\_\_ == "\_\_main\_\_":

    print("Sales started")

    calc\_tax()

With this piece of code at the end of sales module, we can make this file usable as a script as well as a reusable module that we can import into another module

So if we run this file directly, the name of this module will be \_\_main\_\_ and here we can have any initialization code or we can call any of the existing functions in this module.

However if we import this module into another module, this code will not be executed because at that point, the name of this module will no longer be \_\_main\_\_, it will be ecommerce.shopping.sales.