Assignment (Modules & Packages)

**Q1) Write Python code scripts to demonstrate absolute vs relative imports. Prove that relative imports should be explicit.**

**Directory:**

**Q1/**

**- - Package1**

**- - module1.py**

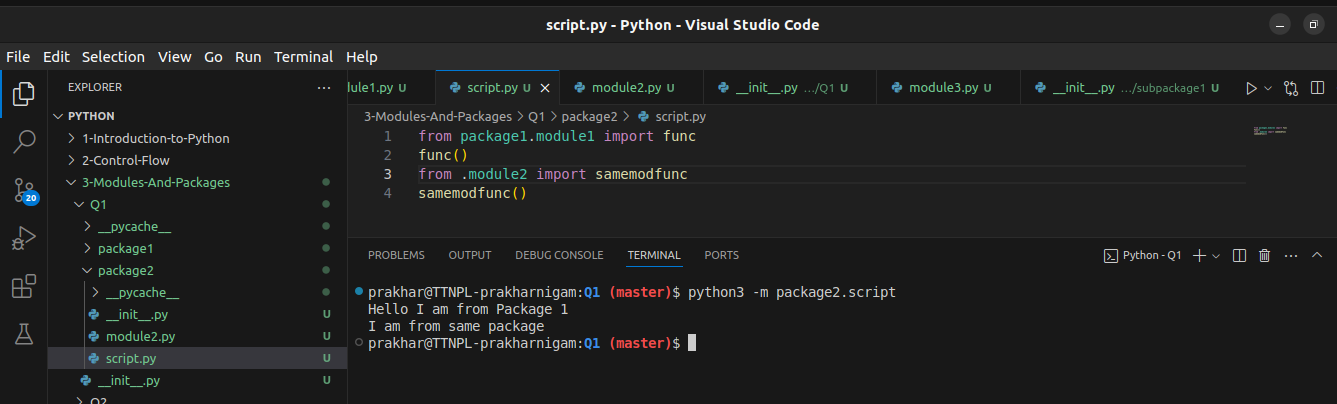
**- - \_\_init.py**

**- - Package2**

**- - module2.py**

**- - script.py**

**- - \_\_init\_\_.py**

****

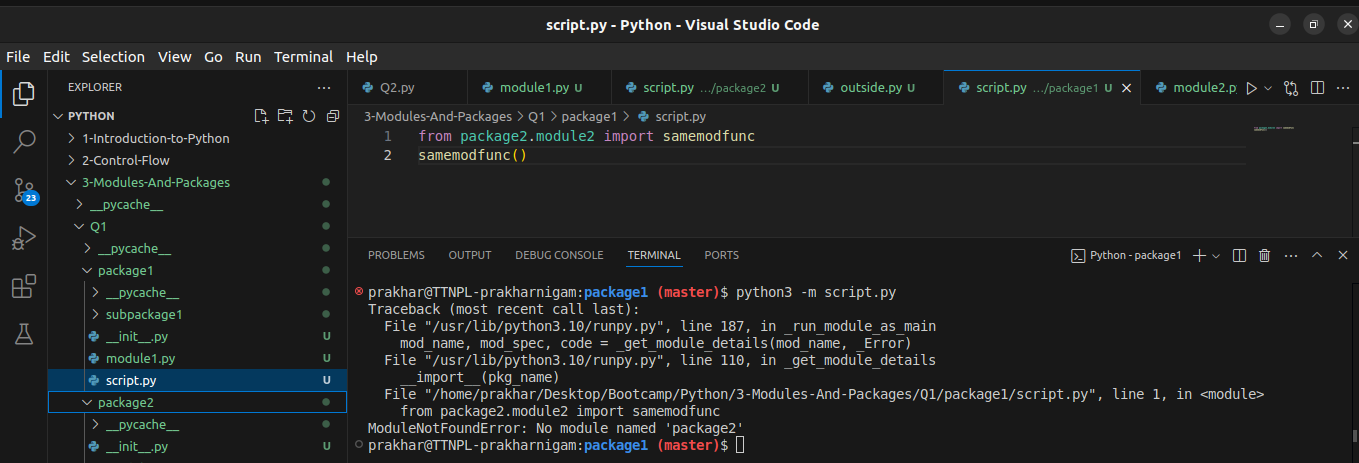
*Here in this current script.py, I have used absolute import to import a function from a module in package1.module1 and relative import to import a function from a module in package2.module2. This path is declared using “.” to navigate upto a parent package and then going down to module2.*

This statement probably refers to the idea of explicitly calling package modules, which prevents a module (or Python script) from running on its own if it contains a relative path. This allows programmers to write code inside of packages without worrying about default script behaviors. As a result, the module can only be invoked from outside of a built package (from the terminal with python3 -m).

**Also demonstrate use of PYTHONPATH**

The directories that the interpreter should search for packages and modules are specified by the Python environment variable PYTHONPATH. When we have unique modules and packages that aren't included in the standard Python library, we use it. Some directories are preconfigured in sys.path by default to search for packages. This includes a pwd configuration, so even if we build a package in a project, we can still access it from the project root directory.

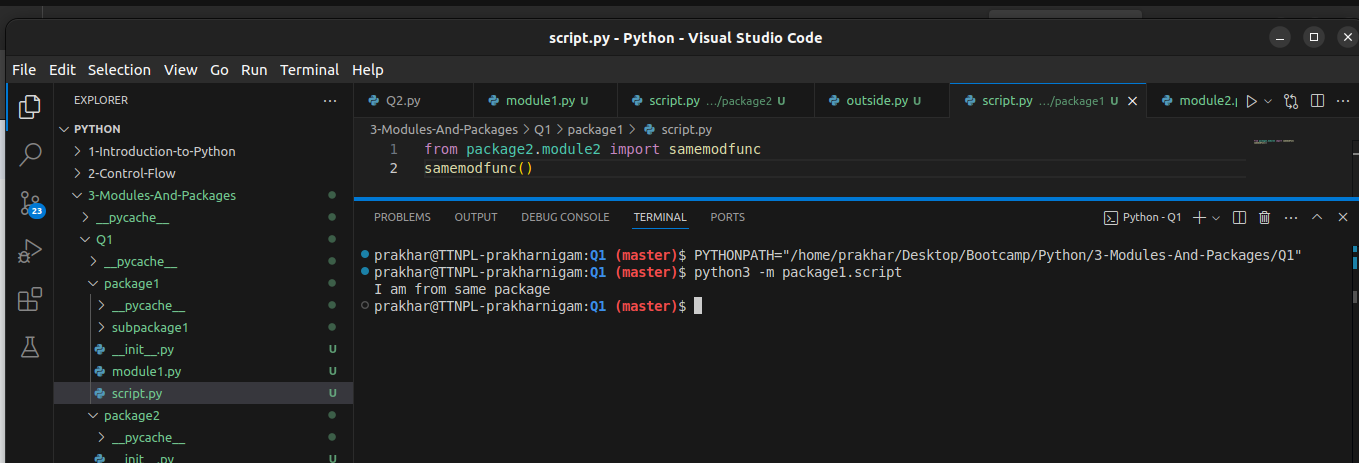
Let’s do vice-versa now. Let’s create a script in Package1 and try to import a function from the Package2



We see an error.

Python cant find the package. Now we can use PYTHONPATH variable to set project’s root directory to look for packages. We can do it directly through shell(as demonstrated below) or we can add the same line export in ~/.bashrc file to permanently add it to PYTHONPATH.

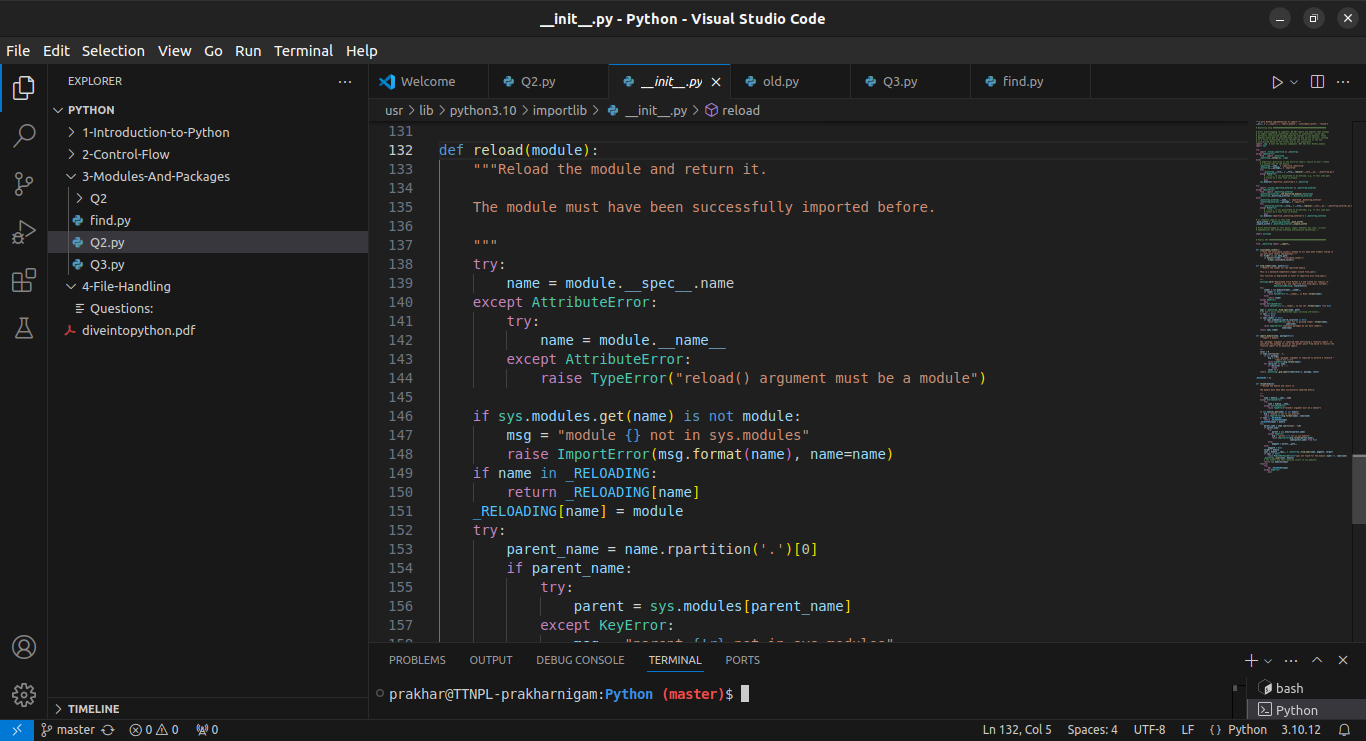
Let’s add it for a current session this script.py file and lets see if it gets executed.



It works now.

**Q2) Explain the use of `from importlib import reload**

***Importlib.reload(module)***

This module function is basically used to reload or get the new version of the file that was imported previously.

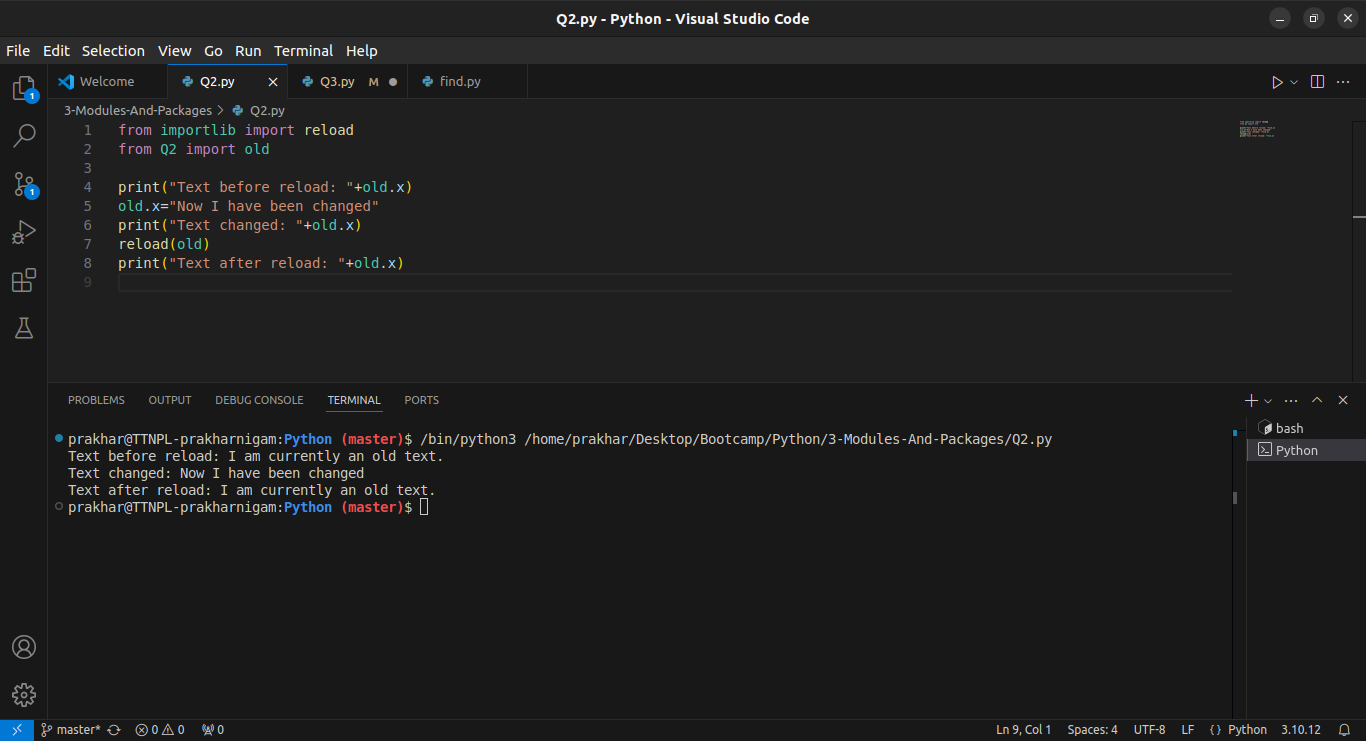
*Reload function from importlib*

In a nutshell, it is used to reload a previously imported module. The argument must be a module object, so it must have been successfully imported before.

This is useful if we have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (which can be different if re-importing causes a different object to be placed in [sys.modules](https://docs.python.org/3/library/sys.html#sys.modules)).

Lets take an example to get a better understanding:

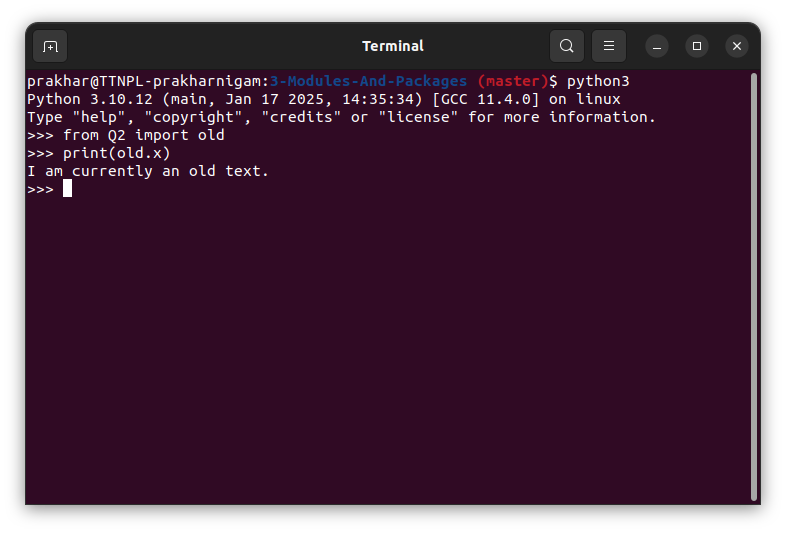
Suppose we have created old.py in a package with text “I am an old text”



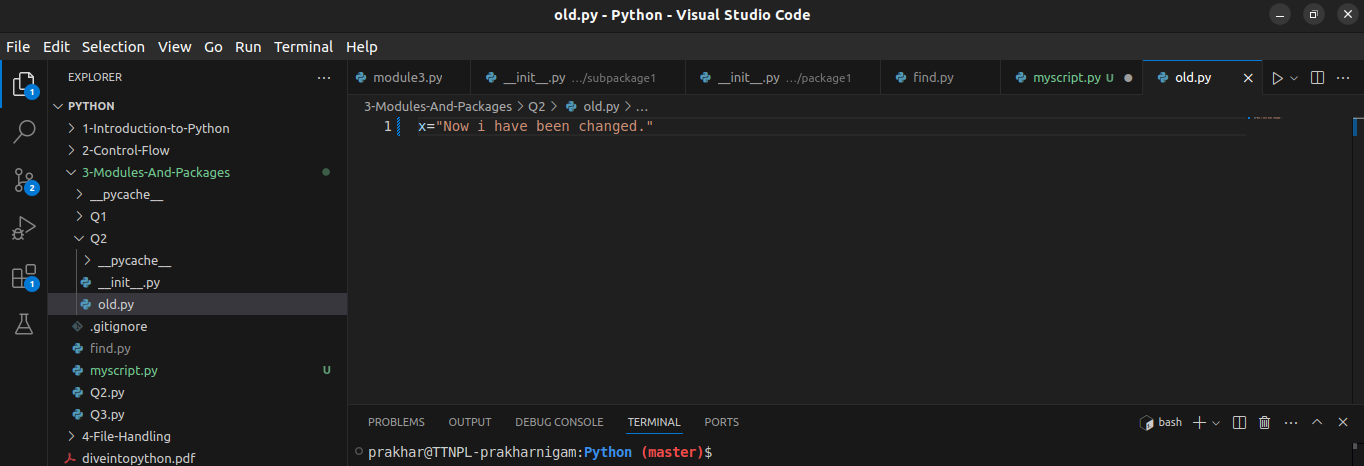
Here I changed the value of the variable during runtime. However after calling reload function, the old value which was originally there in the old.py file was recalled (if it was changed during the runtime, new value would have been imported. However in this case value wasn’t changed, so old value was returned.

Let’s look at it from a way where module was changed during runtime  
  

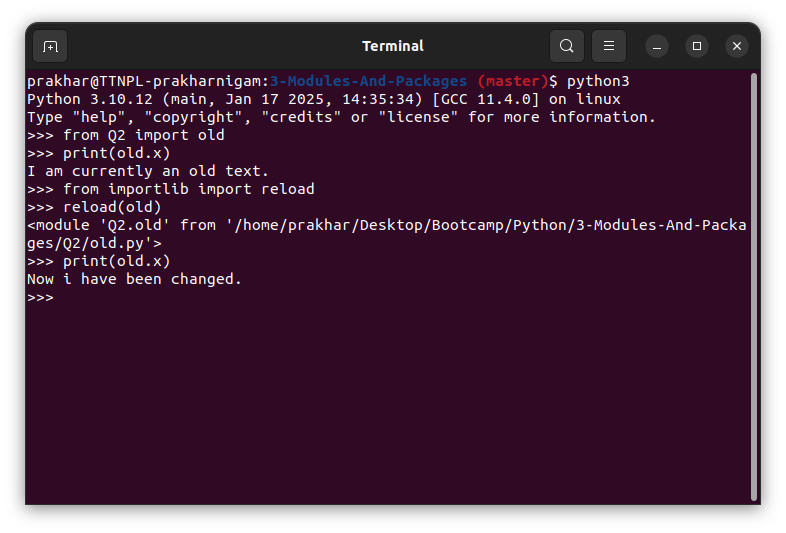

This is the old text that has been in the module. I will import this into the interpreter



If I import this and call the function, it displays the old value which was already there in the module



Now during the runtime of that interpreter, I change the value of ‘x’ in the module.



If I had called the same function again, it would have displayed the old value, however after calling reload function from importlib library, it reloaded the module with the latest values. Thus printing the value of ‘x’ again will give me the new value.

Everytime reload the function is called:

* Python module’s code is recompiled and the module-level code re-executed, defining a new set of objects which are bound to names in the module’s dictionary by reusing the [loader](https://docs.python.org/3/glossary.html#term-loader) which originally loaded the module. The init function of extension modules is not called a second time.
* As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
* The names in the module namespace are updated to point to any new or changed objects.
* Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

**Q3) Read about itertools.count(start=0, step=1) function which accepts options arguments start and end**

Based on this, implement a similar `datecount(start, step)` where start is a `datetime.date` object and step can we string values 'alternative', 'daily', 'weekly', 'monthly', 'Quarterly', 'yearly' (ignore case)

**example execution:**

>> dc = datecount(step='weekly')

>> for i in range(10):

print (next(dc))

**Output:**

2025-01-17

2025-01-24

2025-01-31

2025-02-07

2025-02-14

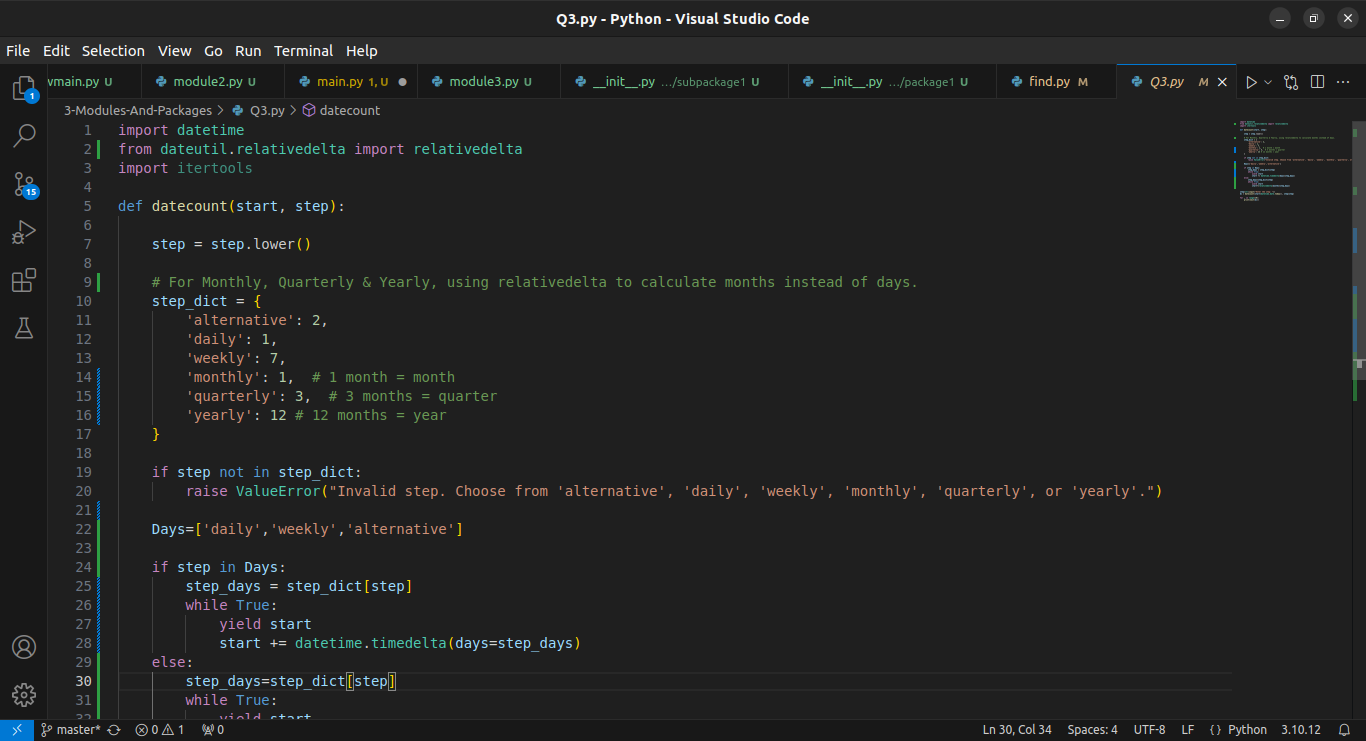
2025-02-21

2025-02-28

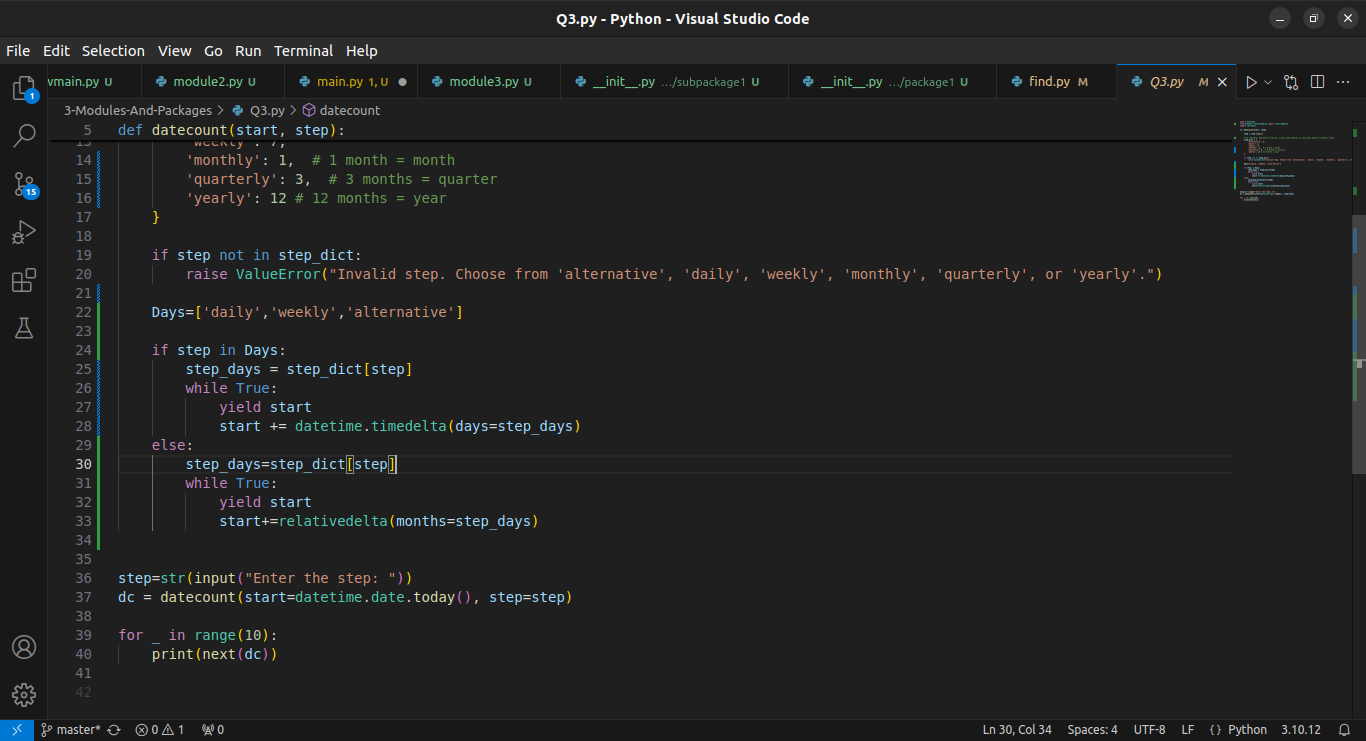
2025-03-07

2025-03-14

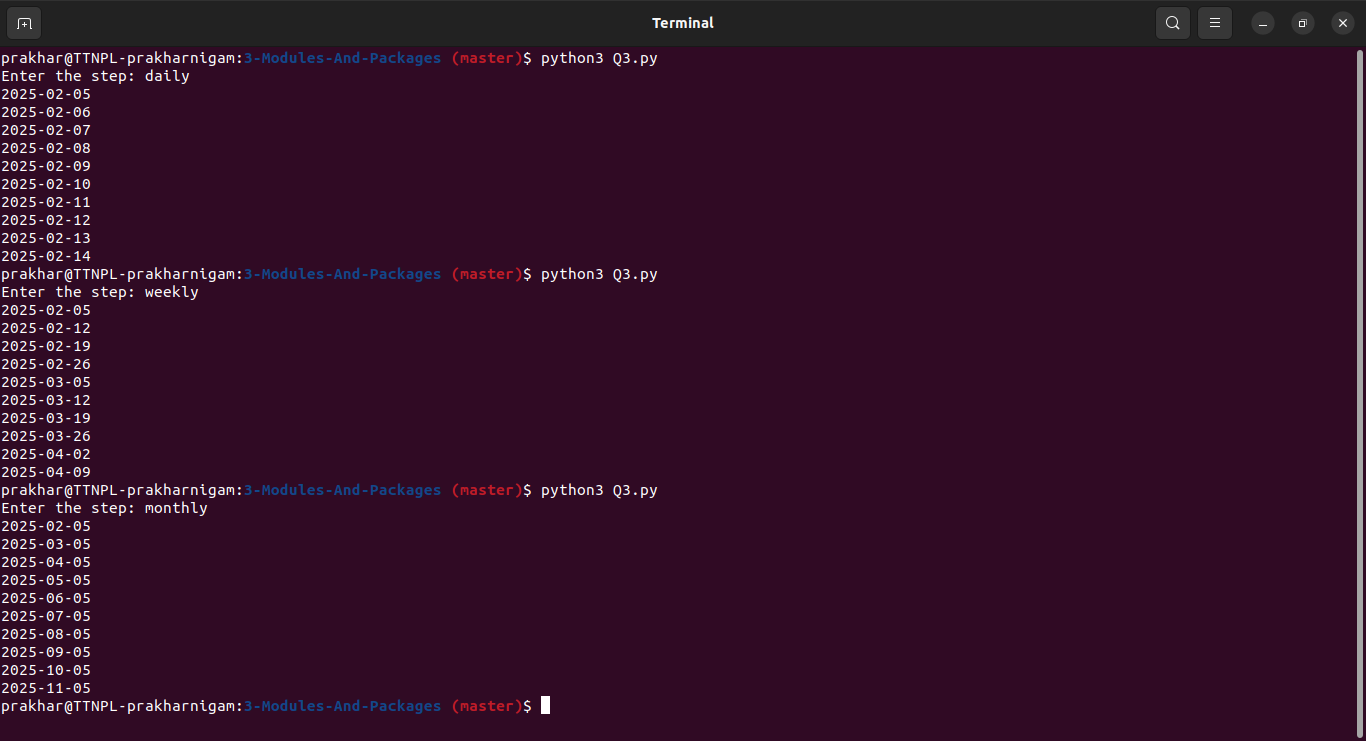
2025-03-21



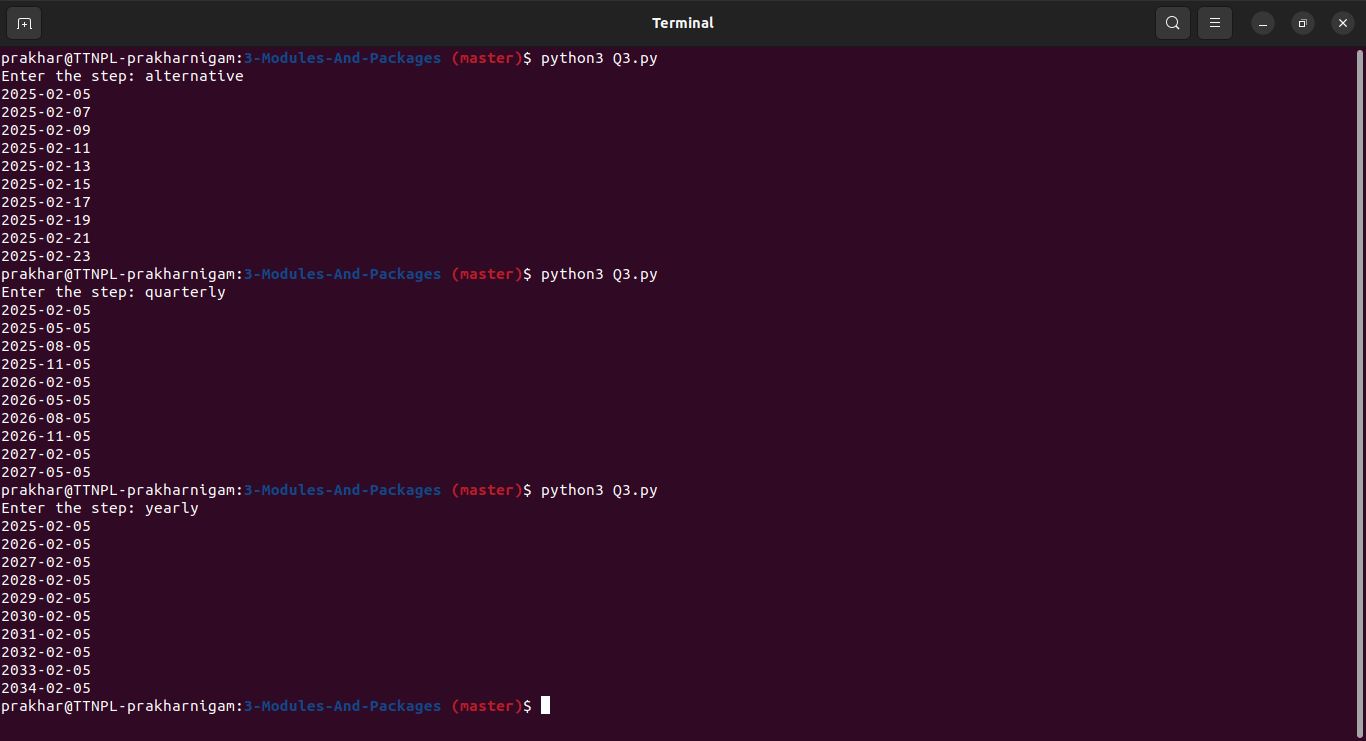
*code*



*Code*

**

*Daily, weekly, monthly*



*alternative, quarterly, yearly*

**Q4) Implement below options:**

`-name`

`-atime`

`-type`

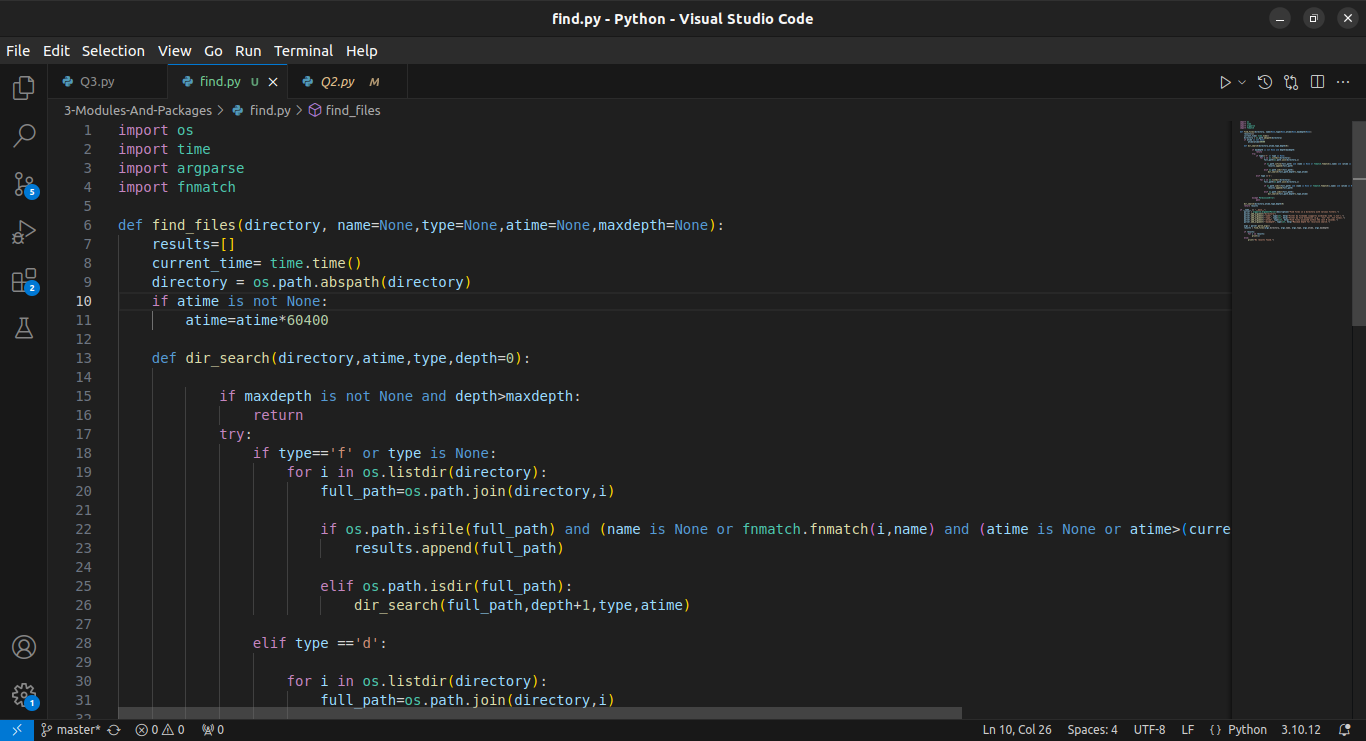
`-maxdepth`

**Example use:**

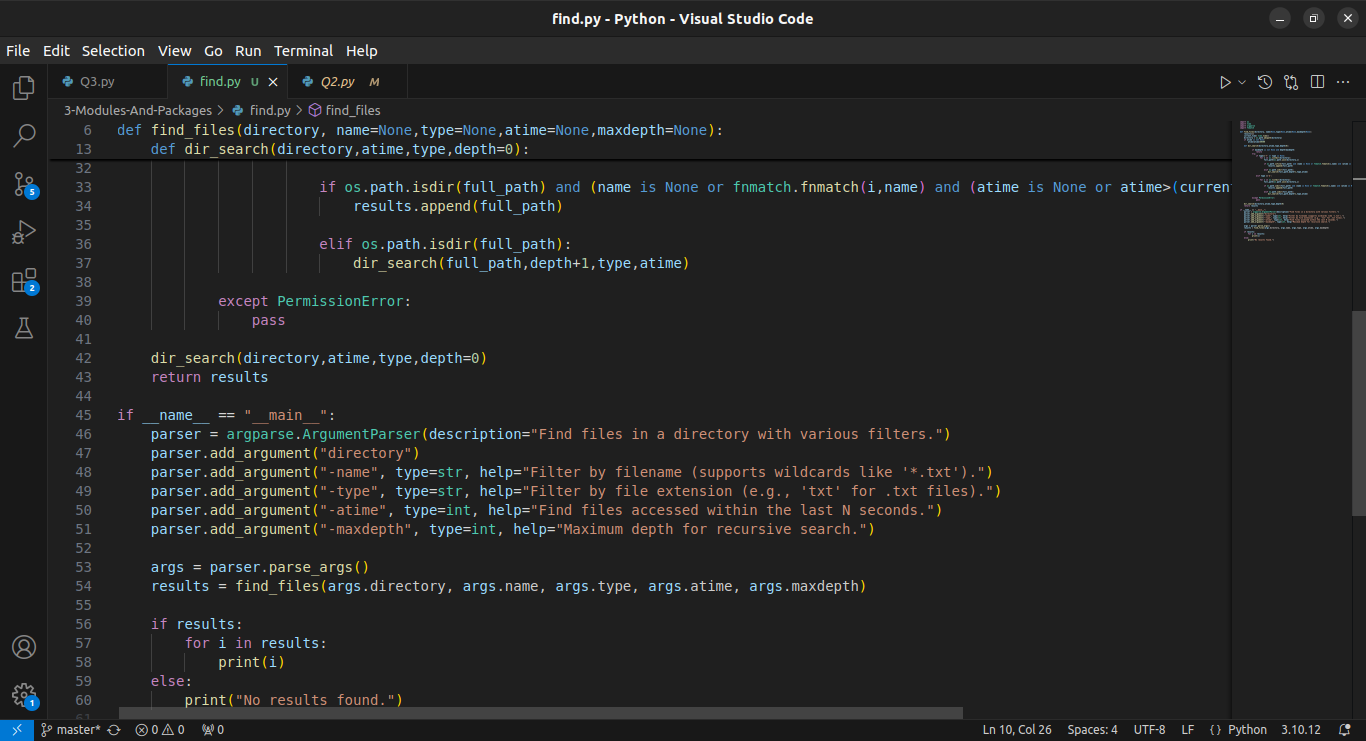
**To find all ".py" files (not folders) in home directory and 2 level sub-directories which were created recently in last 7 days write**

find.py ~/ -name "\*.py" -type f -atime -7

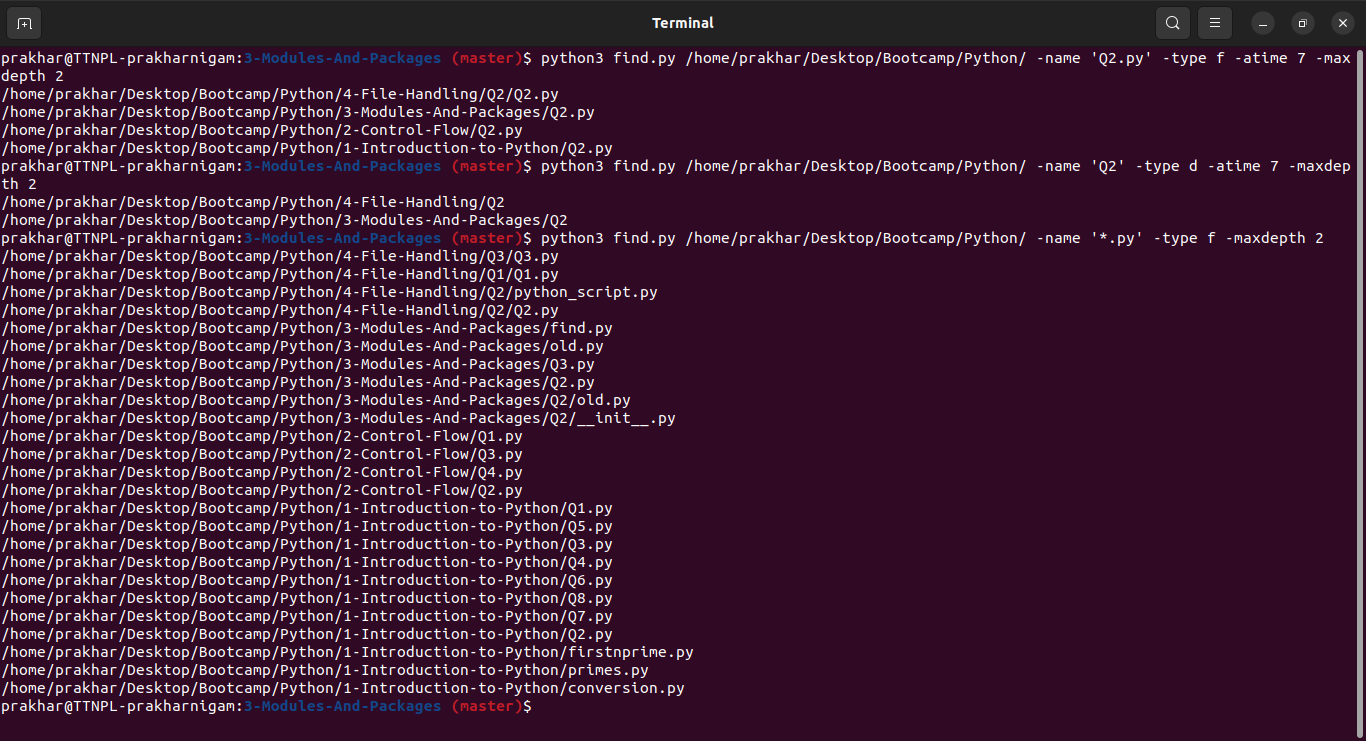
**Output**



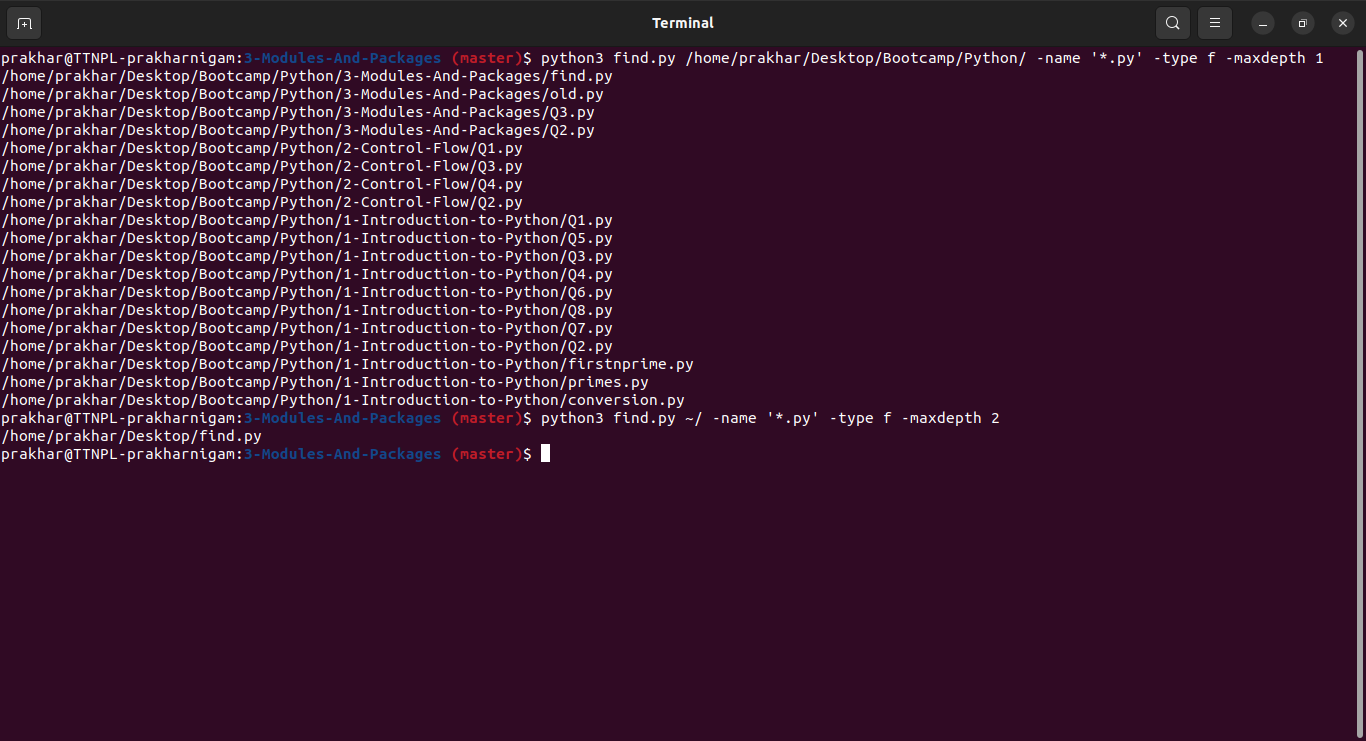
Code 1/2



Code 2/2



Output, tried variations, used different -type for files and directories, specified name



Tried various Maxdepths and tried the example