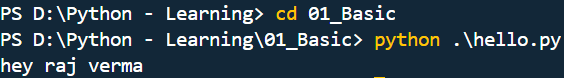
**Installation**

For local – Install Python from its official website

For Online – use google colab or any online compiler

**How to Run**

****

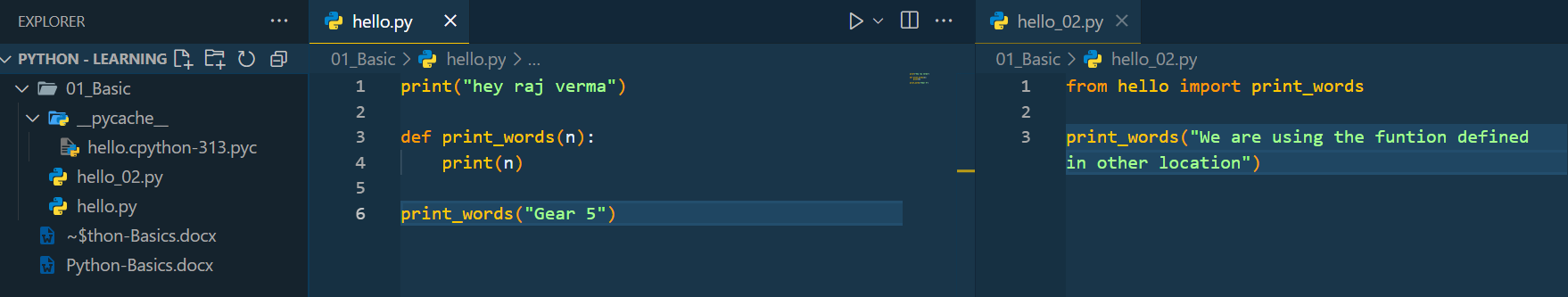
Go into the specific folder and use python .\filename.py to run the code

**Working**

Py converts the code into bytecode [ this can be run on any system ]  
Bytecode runs on Python Virtual Machine or PVM or Python interpretor  
i.e. when we installed python, its VM auto installed with it

.pyc -> compiled python [ frozen binaries ]

**Frozen binaries** are standalone executable files created by bundling a Python interpreter  
Tool used examples – PyInstaller [ package app into one .exe ]  
e.g. you created a python app and want your friend to use it, he don’t have any libraries and packages install so you will use frozen tool to share the app



We get to se a ‘\_\_pycache\_\_’ folder when we use importing of one file into another  
In example above we are calling for print\_words() function from hello\_02.py

The file inside of pycache tracks the changes of code, and re-compile only the changed code, i.e. similar to that of git-github

**Byte Code is not a machine code**

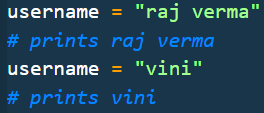
Machine code is a direct instruction to pc hardware, byte code is differ based on which language creating it

**Variant of Python**

Cpython [ default ], jython, Iron python, stackless, PyPy

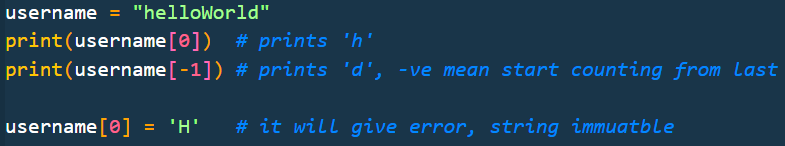
\*note – when we import a file, what happens our code import that specific state of code, and if we change the imported file, then there will be no change in the file which which we imported previously  
So to overcome that, like we want auto re-load the importing on change, we use library,  


**Mutable and Im-Mutable**

Garbage Collection  
Every variable in python is called as object  
So let say we created a string object  
now what happened when we replaced the raj with vini

Username first pointing to “raj” then it do not update the string but it start pointing towards “vini” “raj verma” is still existing

So, this situations calls for garbage collection which cleans all object which do not have any references **this is what called immuatable**

****

**[] – bracket, () – parenthesis, {} – braces/curly braces**

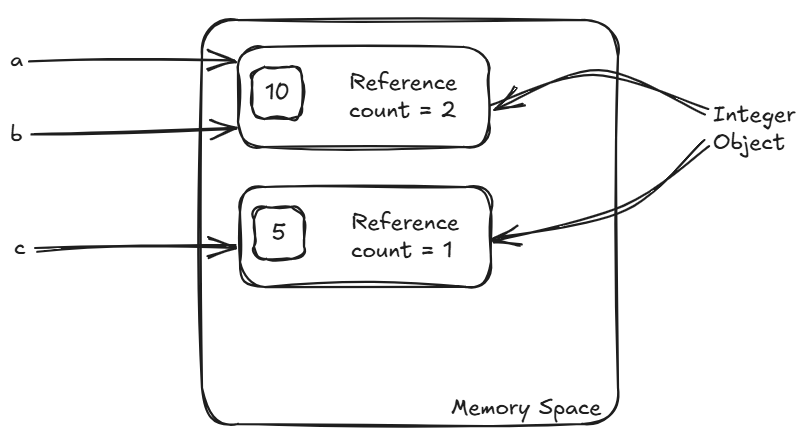
**Data Types / Object Types**

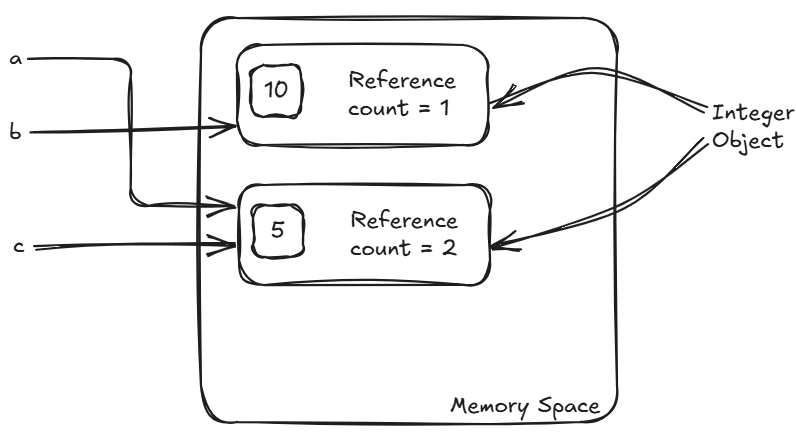
1. Number : 123, 3.14, 5+6j, 0blxll, fraction()
2. String : ‘aa’, “a’a”, b’a\x01c’
3. List : [1 , [2, ‘three’] , 4.5 ] , list(range(10))
4. Tuple : (1, ‘hey’, 4, ‘U’)
5. Dictionary : { 1 : “apple” , 2 : “kela” }
6. Set : { ‘a’, ‘b’, ‘c’ }
7. File : open(‘eggs.txt’), open(r’C:\ham.bin’, ‘wb’)
8. Boolean : true, false
9. Function, Module, Class
10. Advance : Decorators, Generators, Iterators, Metaprogramming

Python don’t have any Data-type, we don’t assign type to variable,  
Variable only stores reference of memory, and in memory stores specifically object know datatype of what it storing

Garbage collection of numbers and string happens with a little delay, as because they are frequently used things. So allocating/deallocating consumes less time

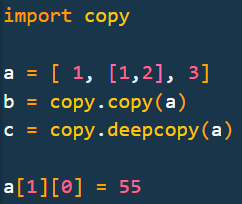
**Memory Referencing**

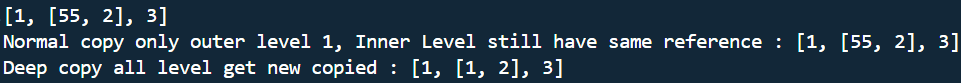
  
Now if we do : a = a – 5, then :

  
Here we can see that ‘a’ got new reference if we perform some operation

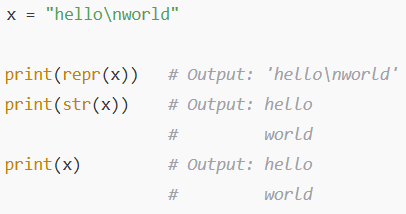
If the outcome is not present then a new object is created and then referenced  
e.g. a = a + 2 => a = 7 new object block is created for 7

**Shallow Copy vs Deep Copy**

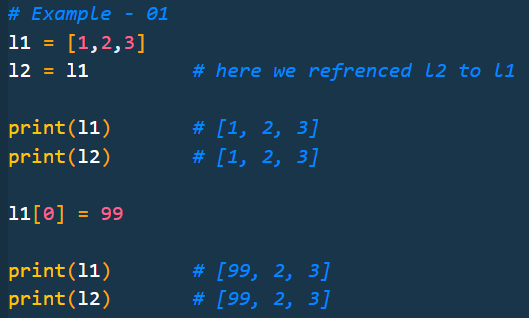
* ‘deep copy’ do copy all level of list, where normal ‘shallow copy’ copies only single level and use same reference for other levels

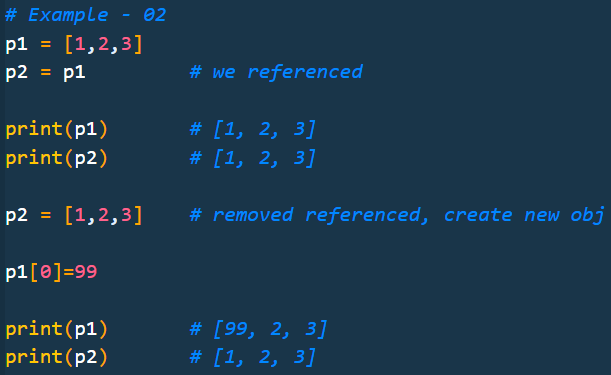
**Output:  
**

In normal case we use alternate approach to copy : **b = a[:]** , it is a shallow copy

**Print –** It calls repr(developer view) or str(user view) internally to display result

**How list behaves in memory references**





\****to install a library we use :* pip install library\_name**

**Open and Reading a file**

f = open(r'D:\Python - Learning\08\_Iteration\_tools\readablefile.py')

*#   -- OR --*

f = open('D:\\Python - Learning\\08\_Iteration\_tools\\readablefile.py')

*# LEARN TO READ FILE --------*

a = f.readline()

print(a)

*#   -- OR --*

a = f.\_\_next\_\_()

print(a)

*# LEARN TO READ MULTIPLE LINE IN ONE GO -----------*

for sentence in f:

    print(sentence)

f = open('D:\\Python - Learning\\08\_Iteration\_tools\\readablefile.py')  *# reset the file line to 1st*

while True:

    line = f.readline()

    if not line:

        break

    print(line, end = "") *# set end to anything if you want*

> When we use readline or next then we permanently moves to next line, i.e. when you use loop below the starting line will be 3rd

> Readline and next works same, the only difference is readline is more enhanced version of next and handles exception greatly

e.g readline return "" when file ends, next returns error "stop iteration" when file ends

**How iter() works**

l = [1,2,3,4]

I = iter(l)

print(I)                *# 0x000001D946D3AEF0*

print(I.\_\_next\_\_())     *# 1*

print(I)                *# 0x000001D946D3AEF0*

print(I.\_\_next\_\_())     *# 2*

print(I)                *# 0x000001D946D3AEF0*

print(I.\_\_next\_\_())     *# 3*

print(I)                *# 0x000001D946D3AEF0*

print(I.\_\_next\_\_())     *# 4*

print(I)                *# 0x000001D946D3AEF0*

print(I.\_\_next\_\_())     *# error - 'StopIteration'*

*# iter always point to the start location, but save the last point uptill it read*

*# internal pointer remember the last read point*

**How iter() handling differ in case of opening a file & traversing on list**

f = open(r'D:\Python - Learning\08\_Iteration\_tools\readablefile.py')

l = [1, 2, 3, 4]

print(f is iter(f)) *# True*

print(l is iter(l)) *# False*

f is itself a iterable object  
but l is list not a iterable object, we need to create a separate iter object

**Yield Keyword  
>** think of yield like a **vending machine**:

* Normally, a function gives you **everything at once** — like dumping the whole pack of snacks.
* A function with yield gives you **one item at a time** when you press the button — **only when you ask**.
* It **remembers where it left off**, so it doesn’t start over every time.

Perfect when you're working with a **huge amount of data**, but don’t want to carry it all at once — just take what you need, when you need it.

*# Yield used to return one value at a time, while remebering last return value*

*# Generally used in creating Generators, working with large datasets, streaming data, or pipelines*

def even\_generate(n):

    for num in range(2, n+1, 2):

        yield num

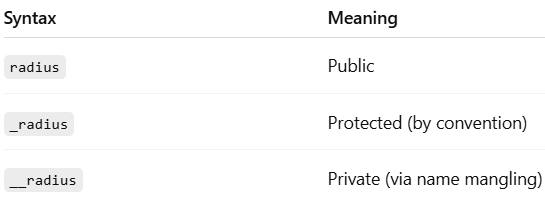
*# yield also work as return in terms of exiting function*

print(even\_generate(10))        *# 0x000002174D4202E0*

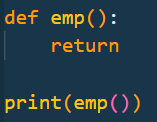
for nums in even\_generate(10):

    print(nums)

*# if we use return instead of yield then 'even\_generate' dont work as iterable object*

**Access Modifier  
**

**None is printed**

 If we try to print some empty function then print -> prints  
 'None'

If we go with only print() then it print blank line i.e. '\n'

**API Handling**

We install library first : pip install requests  
[ we installed numpy before, to check all installed packages : pip list ]

def fetch\_random\_users():

    url = 'https://api.freeapi.app/api/v1/public/randomusers'

    res = requests.get(url)     *# res is in text format*

    data = res.json()           *# converting it to json make data handling more efficient*

    if data['success']:

        print("Yes")

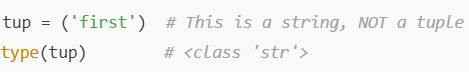
we can upgrade get request more by:

headers = {"accept": "application/json"}

params = {"page": str(page), "limit": str(limit)}

res = requests.get(url, headers=headers, params=params)

**How to define single element in a tupple ?**

**Python with MongoDB**

> first install : pip install pymongo

Then follow the ‘’***17\_Project***’’ file

**Python virtual env**

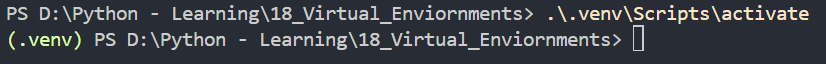
> they are used to know which packages our project is using and running on, generally when some new developer works on the project they get to know how to handle the project

install : pip install virtualenv

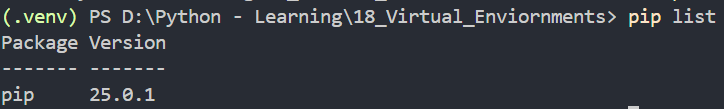
1. to create a virtual environment first go into the folder where you want to create it

2. then run the command into the folder : python -m venv <any\_file\_name>

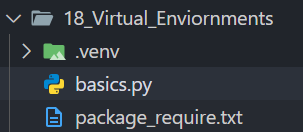
generally : <any\_file\_name> is taken as ‘.venv’ so that developer easily understand it

3. to enter the virtual env now do below:  


\* .venv is file name, if filename get changed that specific field also you have to change

4. now check the installed packages via:  


So the meaning of using it is “we don’t need to globally installed all the packages, just create a virtual env and install packages as per demand of your project”

Now install some other package e.g. numpy and then get all installed packages list in a text file via:  
  


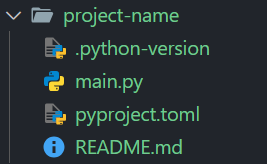
Now to all packages written in ‘package\_require.txt’ you can use :  


**Modern way of using virtual env**

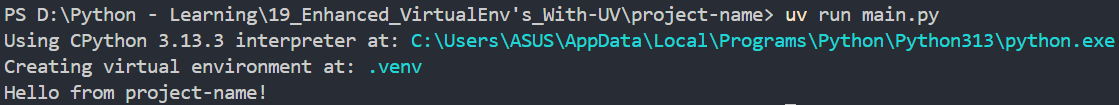
We will use -> ‘UV’ a modern package installer which work similar to npm/yarn  
It installs the packages in the local folder and creates .venv file on its own

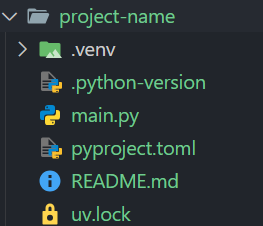
Install : pip install uv

Doing Below will create a new folder  


Folder structure  


pyproject.toml is similar to package.json file

Now run your main.py file, which will create a .venv on its first execution  


Updated folder  


uv.lock is same as package-json.lock

How to install a Package : 

How to build a distributable file : 