Lecture 10 – Advance Python Topics - 1

Topics to be covered:

- **Scope**
- **Modules and Packages**
- **PIP**
    - **Date Time and Math Module**
    - **Random and String Modules**
- **File Handling / CRUD**

SCOPE

What is Scope of variables?

Scope is like boundary of a variable within a program. It is like where you can use this variable. Not all variables can be accessed from anywhere in a program. The part of a program where a variable is accessible is called its **scope**.

Python follows **LEGB** rule for the scope. Where:

L = Local, E = Enclosing, G = Global, B = built-in

Local Variables

Local variables are the one that are defined within a function and so their scope is only within those functions. Those variables cannot be used outside the functions they were created for.

Example:

```
def function():
    a, b = 5, 10
    return (a+b)
# Now if we use the variables outside the function,
# it will throw NameError exception.
print(f"A is: {a}")
```

Enclosing Scope

If we define a function within a function (Nested functions), then the outer function (the main function) will have a larger scope then the inner function (sub function or nested function). So if we use values of outer function's variables in inner functions, we will get the result, but if we want to use the variables created in inner function in outer function, NameError exception will be thrown.

Example:

Consider the following example and analyze the error.

```
def outer():
    first_num = 1
    def inner():
```

```
        second_num = 2
        # Print statement 1 - Scope: Inner
        print("first_num from outer: ", first_num)
        # Print statement 2 - Scope: Inner
        print("second_num from inner: ", second_num)
    inner()
    # Print statement 3 - Scope: Outer
    print("second_num from inner: ", second_num)

outer()
```

Got an error? This is because you cannot access second_num from outer() (# Print statement 3). It is not defined within that function. However, you can access first_num from inner() (# Print statement 1), because the scope of first_num is larger, it is within outer().

This is an enclosing scope. Outer's variables have a larger scope and can be accessed from the enclosed function inner().

Global Variables

A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Global variables are available from within any scope, global and local.

Example:

```
name = 'Wasiq'

def greeting():
    print(f"Hello {name}")

greeting()
```


Here, the variable name is a global variable and it can be accessed within a function as well.

NOTE:

If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function):

Consider this:

```
x = 300
def myfunc():
  x = 200
  print(x)
myfunc()
print(x)
```

If you are within a function and still need to create a variable with the global scope, you can use the Global keyword.

Example:

```
def myfunc():
  global x
  x = 300

myfunc()

print(x)
```

In this example we have created a global variable within function and used it outside as well. Similarly, we can change the value of a global variable within a function using global keyword. Consider this:

```
x = 300

def myfunc():
  global x
  x = 200

myfunc()

print(x)
```

Now Can you guess the output of this program?

```
greeting = "Hello"

def change_greeting(new_greeting):
    global greeting
    greeting = new_greeting

def greeting_world():
    world = "World"
    print(greeting, world)

change_greeting("Hi")
greeting_world()
```

Built-in Scope:

All the special reserved keywords fall under this scope. We can call the keywords anywhere within our program without having to define them before use.

Extra: Search for **nonlocal** keyword in Python.

LEGB Rule:

LEGB (Local -> Enclosing -> Global -> Built-in) is the logic followed by a Python interpreter when it is executing your program.

Consider this program:

```
# Global Scope
x = 0

def outer():
    #Enclosed Scope
    x = 1
    def inner():
        #Local scope
        x = 2
        print(x)

    inner()
    print(x)

outer()
print(x)
```

Let's say you're calling print(x) within inner(), which is a function nested in outer(). Then Python will first look if "x" was defined locally within inner(). If not, the variable defined in outer() will be used. This is the enclosing function. If it also wasn't defined there, the Python interpreter will go up another level - to the global scope. Above that, you will only find the built-in scope, which contains special variables reserved for Python itself.

MODULES & PACKAGES

What is Module?

Module is a Python file containing a set of functions which we can use in our main / working python file.

There are built-in modules as well as modules that we can create (custom modules).

First Let's see how we can create a module.

We can create a module by creating a Python file and storing our functions within that file. Since module is a Python file it must end with .py extension. All the functions that we are creating must be inside this file.

Now let's suppose our module name is **custom.py** and it contains functions namely **custom_one** and **custom_two**. Two use a module, we have to import it first.

To use a module we have to import it first.

```
import custom
```
then we will use its functions in this format, **module.function(arguments)**.

```
custom.custom_one()

custom.custom_two()
```

NOTE: Not only functions, but the modules can contain variables of all types and they can be accessed too. Suppose our module contains a dictionary **user** which contains some information

```
user = {
  "name": "Wasiq",
  "age": 36,
  "country": "Pakistan"
}
```
So this can be accessed like this:

```
user_name = custom.user["age"]
```

Sometimes we can rename our module in short form for easy coding

```
import custom as cu

cu.custome_one()
```

Also, it is not necessary to import everything in a module, we may import only certain parts / functionalities of a module like this:

```
from custom import custome_two
```
or like this:

```
from custom import custom_two as cu2
```
if we want to rename it as well.

SOME NOTES ON IMPORTS

Let's say you import a module **abc** like so:

```
import abc
```

- The first thing Python will do is look up the name **abc** in **sys.modules.** This is a cache of all modules that have been previously imported.
- f the name isn't found in the module cache, Python will proceed to search through a list of built-in modules. These are modules that come pre-installed with Python and can be found in the **Python Standard Library**.
- If the name still isn't found in the built-in modules, Python then searches for it in a list of directories defined by **sys.path**. This list usually includes the current directory, which is searched first.

- When Python finds the module, it binds it to a name in the local scope. This means that **abc** is now defined and can be used in the current file without throwing a **NameError**.
- If the name is never found, you'll get a **ModuleNotFoundError**.
- NOTE: **PEP8** (https://pep8.org/), which is the official styling guide of python recommends using the import at the start of the program after any initial comment or doc string.

ABSOLUTE VS RELATIVE IMPORTS

1. **Absolute Imports**
    a. An absolute import specifies the resource to be imported using its full path from the project's root folder.
    b. Let's say you have the following directory structure:

```
└── project
    ├── package1
    │   ├── module1.py
    │   └── module2.py
    └── package2
        ├── __init__.py
        ├── module3.py
        ├── module4.py
        └── subpackage1
            └── module5.py
```

    c. There's a directory, project, which contains two sub-directories, package1 and package2. The package1 directory has two files, module1.py and module2.py.
    d. The package2 directory has three files: two modules, module3.py and module4.py, and an initialization file, __init__.py. It also contains a directory, subpackage, which in turn contains a file, module5.py.
    e. Let's assume the following:
        i. package1/module2.py contains a function, function1.
        ii. package2/__init__.py contains a class, class1.
        iii. package2/subpackage1/module5.py contains a function, function2.
    f. The following are practical examples of absolute imports:

```
from package1 import module1
from package1.module2 import function1
from package2 import class1
from package2.subpackage1.module5 import function2
```

    g. Note that you must give a detailed path for each package or file, from the top-level package folder. This is somewhat similar to its file path, but we use a dot (.) instead of a slash (/).
    h. Absolute imports are preferred because they are quite clear and straightforward. It is easy to tell exactly where the imported resource is, just by looking at the statement. Additionally, absolute imports remain valid even if the current location of the import statement changes. In fact, PEP 8 explicitly recommends absolute imports.
2. **Relative Imports**
    a. Sometimes, absolute imports can get quite verbose:

```
from package1.subpackage2.subpackage3.subpackage4.module5 import
function6
```

b. Obviously this is not the kind of import one would want. For this, we use relative imports.
c. A relative import specifies the resource to be imported relative to the current location—that is, the location where the import statement is.
d. The syntax of a relative import depends on the current location as well as the location of the module, package, or object to be imported.
e. Here are few examples:

```
from .some_module import some_class
from ..some_package import some_function
from . import some_class
```

f. You can see that there is at least one dot in each import statement above. Relative imports make use of dot notation to specify location.
g. A single dot means that the module or package referenced is in the same directory as the current location. Two dots mean that it is in the parent directory of the current location—that is, the directory above. Three dots mean that it is in the grandparent directory, and so on.
h. Let's have we have the same project structure as in the case of absolute imports.
i. If:
    i. package1/module2.py contains a function, function1.
    ii. package2/__init__.py contains a class, class1.
    iii. package2/subpackage1/module5.py contains a function, function2.
j. You can import function1 into the package1/module1.py file this way:

```
from .module2 import function1
```

k. You can import class1 and function2 into the package2/module3.py file this way:

```
from . import class1
from .subpackage1.module5 import function2
```

l. Unfortunately, relative imports can be messy, particularly for shared projects where directory structure is likely to change. Relative imports are also not as readable as absolute ones, and it's not easy to tell the location of the imported resources.

FINAL NOTES ON IMPORTING

1. We either import built-in modules or custom modules.
2. Built-in modules are already defined in Python's standard library and we can use them as we want. Like normal import, without any issue.
3. Custom modules on the other hands can be imported depends on their availability.
4. If Custom modules are our own modules, then:
    a. If they are in the same directory as the file calling them, we simply import them.
    b. If they are in different directory of the same project, we might use absolute or relative import techniques.
    c. If they are in different directories, then we might have to modify the sys.path or use relative imports.
    d. This is how to modify the path

```
import sys
sys.path.append('/path/to/your/module_directory')

import custom_module
```

5. If Custom modules are not our modules, then:
    a. If they are available on Package Managers, then we can use pip to install them.
        i. pip install package_name
    b. Once installed, they can be used.
    c. If they are unavailable on Package Managers, we can download / copy paste them manually in the project location and use them.
    d. If the module is hosted on a Git repository, you can use Git submodules to include it in your project.

What are Packages?

A Python package is a way of organizing related modules into a single directory hierarchy. It allows for a hierarchical file structure that is easy to navigate and manage. A package contains all the files you need for a module.

Example:

my_package/

|-- __init__.py

|-- module1.py

|-- module2.py

|-- subpackage/

| |-- __init__.py

| |-- module3.py

What are Package Managers?

Package managers are tools that automate the process of installing, updating, and managing libraries and dependencies in a programming language. For Python, the primary package manager is PIP.

NOTE:

You can search for Python packages on the Python Package Index (PyPI), which is a repository of software packages developed and maintained by the Python community. https://pypi.org/

PIP

PIP is Python's default package manager. This comes pre-installed after Python 3.4

We can check its version by going into CMD and typing **pip –version**. If it is installed it will return a version.

If it is not installed then it can be installed using this link: https://pypi.org/project/pip/

Once PIP is installed, we can use it to download different packages.

Installing a Package

Suppose we have a package name called **MyPackage**

We can install it by running the install command in pip, (run in some CLI or CMD)

```
pip install MyPackage
```

You would enter this command in your terminal or CMD and press Enter. This command tells the Python Package Installer (pip) to download and install the MyPackage package from the Python Package Index (PyPI).

NOTE: Sometimes we have to use virtual environments to prevent some package related issues, but we will get to this sometimes later.

To use a package, we have to import it like normal and use it.

MORE:

Uninstalling a package: pip uninstall MyPackage

List of all packages: pip list

We can also create our own package and add it in repository as well so others can also download it.

Now we will try to use some important Modules / Packages in Python. Like RegEx, Maths, Date and Time, Random, String.

DATE TIME MODULE

By default, Date is not a data type of python. So we have to import it first using **datetime** module.

```
import datetime
```

One imported we can use its many built-in functions to work with date and time

Here is a list of important datetime methods:

datetime() / date() classes:

These classes are the most common one to work with dates and time. It has some methods as well.

1. Displaying current date and time:
    a. now() method is used to create a datetime object containing the current date and time.

```
import datetime as dt

# get the current date and time
now = dt.datetime.now()

print(now)
```

   b. The output of this will be **2024-01-05 12:10:38.166397** (or whenever you check it).
   c. Note that it offers year, month, day of the month in date
   d. Similarly, hour, minute, second and microseconds accuracy as well.
   e. All of these properties can be accessed as well.

2. Displaying current date only.
   a. today() method is used to display current date only. But this is a method is date() class.

```
import datetime as dt
# get the current date only
date = dt.date.today()
print(date)
```

   b. Output will only be the date

3. Displaying only time
   a. We can use datetime() class with now() and time() methods to display only time.

```
time = dt.datetime.now().time()
```

   b. The output will only be the time.

4. Accessing Components of dates (year, month, hour)

```
from datetime import date

# date object of today's date
today = date.today()

print("Current year:", today.year)
print("Current month:", today.month)
print("Current day:", today.day)
```

5. Accessing Components of time (hours, minutes, seconds)

```
import datetime as dt
a = dt.datetime.now().time()
print("Hour =", a.hour)
print("Minute =", a.minute)
print("Second =", a.second)
print("Microsecond =", a.microsecond)
```

6. Creating Date and Time

```
from datetime import datetime
# datetime(year, month, day)
a = datetime(2022, 12, 28)
print(a)
# datetime(year, month, day, hour, minute, second, microsecond)
```

```
b = datetime(2022, 12, 28, 23, 55, 59, 342380)
print(b)
```

7. Calculating time span (durations)
    a. A timedelta object represents the difference between two dates or times.

```
from datetime import datetime, date

# using date()
t1 = date(year = 2018, month = 7, day = 12)
t2 = date(year = 2017, month = 12, day = 23)

t3 = t1 - t2

print("t3 =", t3)

# using datetime()
t4 = datetime(year = 2018, month = 7, day = 12, hour = 7, minute = 9, second
= 33)
t5 = datetime(year = 2019, month = 6, day = 10, hour = 5, minute = 55, second
= 13)
t6 = t4 - t5
print("t6 =", t6)
```

    b. It is worth noticing that datetime or date classes take named or positional arguments both.
    c. Check this example:

```
from datetime import datetime

# Using positional arguments
date1 = datetime(2023, 5, 10, 15, 30, 0)
print("Date 1:", date1)

# Using named arguments
date2 = datetime(year=2023, month=5, day=10, hour=15, minute=30, second=0)
print("Date 2:", date2)
```

8. We can also have difference between two **timedelta** objects

```
from datetime import timedelta
t1 = timedelta(weeks = 2, days = 5, hours = 1, seconds = 33)
t2 = timedelta(days = 4, hours = 11, minutes = 4, seconds = 54)
t3 = t1 - t2
print("t3 =", t3)
```

9. Formatting date and time:

a. The way date and time is represented may be different in different places, organizations etc. It's more common to use mm/dd/yyyy in the US, whereas dd/mm/yyyy is more common in the UK.
b. For this reason we use strftime() and strptime()

10. Strftime() is most commonly used. You can learn more about it at this URL:
    a. https://www.programiz.com/python-programming/datetime/strftime

MATHS MODULE

Python has also a built-in module called math, which extends the list of mathematical functions.

```
import math
```

Once imported, you can use its different methods.

To see a list of available maths methods, you can do this:

```
print(dir(math))
```

dir() function is used to access the entire directory / list of available functions (methods) in a module.

Example: Usage of math module's square root function.

```
print(f"The square root of 64 is: {math.sqrt(64)}")
```

For a complete list of important math functions and constants, check this link:
https://www.w3schools.com/python/module_math.asp

RANDOM MODULE

Python has a built-in module that you can use to make random numbers.

```
import random
```

This module also has many amazing methods for random number generations.

```
a = random.randrange(1,10)
print(a)
```

This program generates a random number in the given range. (Excluding 10)

You can see many of such random methods at this link:

https://www.w3schools.com/python/module_random.asp

STRING MODULE

The string module in Python provides a collection of string constants and some utility functions. It does not include string manipulation functions but rather contains predefined constants such as ASCII letters, digits, and punctuation characters.

```
import string
```

More Examples

```
print(string.ascii_letters)
print(string.ascii_lowercase)
print(string.ascii_uppercase)
print(string.digits)
print(string.punctuation)
```

Output:

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

Checkout the official documentation for more details: https://docs.python.org/3/library/string.html

STRONG PASSWORD GENERATOR EXAMPLE:

```
import random
import string

length = 12
characters = string.ascii_letters + string.digits + string.punctuation

password = ''.join(random.choice(characters) for _ in range(length))

print(password)
```

TASK: Can you make a password strength checker? Try!

FILE HANDLING

File handling in Python is a powerful and versatile tool that can be used to perform a wide range of operations. Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, like other concepts of Python, this concept here is also easy and short.

Here is a complete breakdown and examples from simple to complex files operations.

NOTE: CRUD is often a term associated with create, read, update and delete. Although this is mainly associated with Databases, but we can also use it for file handling as well.

Reading Files

First of all, note that python can treat two types of files

1. Text
2. Binary (like images)

Now suppose we have a file name check.txt (in our project directory) with lines of text.

Before performing any operation on the file like reading or writing, first, we have to open that file. For this, we should use Python's inbuilt function open() but at the time of opening, we have to specify the mode, which represents the purpose of the opening file.

Syntax: `f = open(filename, mode)`
These are the different modes available:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

Additionally, we can also specify the type of file as well.

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

NOTE: "r" and "t" are default values. So the below lines of code are the same.

f = open("check.txt") or f = open("check.txt", "rt")

The open() function returns a file object, which has a read() method for reading the content of the file:

```
f = open("check.txt", "r")
print(f.read())
```

Output:

```
My name is Sheela
Sheela ki jawani
```

NOTE: If the file is located in a different location, you will have to specify the file path, like this:

```
f = open("C:\Users\HP\Desktop\Code
Camp\Alpha\Lectures\Python\programs\check.txt", "r")
print(f.read())
```

The read method also takes an optional argument as number which tells how many characters to read in the opened file.

We can also read lines in the file using readlines().

Every time we call readline(), a line is returned. If we call it multiple times, it will move to read the next line in each attempt.

```
f = open("check.txt", "r")
print(f.readline())
print(f.readline())
```

readlines() is another method similar to read, but instead of returning one single string of the content of the file, it returns a list of strings where each member of the list is a string representing a line.

For example, the output of the following code

```
f = open("check.txt", "r")
print(f.readlines())
```

will be **['My name is Sheela\n', 'Sheela ki jawani']**

There is another and more frequently used method of reading file which is the **with** statement. Consider the example:

```
with open("check.txt") as file:
    data = file.read()

print(data)
```

NOTE: If we loop through the file object, then we can also print line by line like this:

```
f = open("check.txt", "r")
for each_line in f:
  print(each_line)
```

Closing a file.

Once done, it is a good and recommended practice to close a file as well.

```
mefile = open("check.txt", "r")
print(mefile.readlines())
mefile.close()
```

NOTE: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

Updating Files

To write to an existing file, you must add a parameter to the open() function:

“a”: to append, this will add content at the end of the file.

“w”: to write, this will over – rite any existing content with the new content.

Example:

```
file = open("check.txt", "a")
file.write("\nI am 1/cos(C) 4u")
file.write("\nMe 13 hands not coming")
file.close

# Now lets open it again and read
file = open("check.txt", "r")
print(file.read())
```

Output:

```
My name is Sheela
Sheela ki jawani

I am 1/cos(C) 4u
Me 13 hands not coming
```

We can also erase content of a file and write new content.

```
file = open("check.txt", "w")
file.write("Bad song! Deleted ... ")
file.close

# Now lets open it again and read
file = open("check.txt", "r")
print(file.read())
```

Output:

```
Bad song! Deleted ...
```

Creating a File

The open() command can also be used to create a file. Both “w” and “a” parameters will create a file for writing / appending if they don’t exist.

Let’s check:

```
file = open("check1.txt", "w")
file.write("Hello Frands chaye pi lo ... ")
file.close

# Now lets open it again and read
file = open("check1.txt", "r")
```

```
print(file.read())
```
Output:

The file check1.txt is created and content is also written on it.

Open() command also takes optional argument “x” to create a file. But if the file already exists, it will through errors.

```
f = open("myfile.txt", "x")
```

Deleting Files

Deleting file requires os module.

```
import os
os.remove("check1.txt")
```
The file will be deleted. If they file does not exist, it will through error. One way to avoid is to check if the file is there (before deleting) then it should delete.

```
import os
if os.path.exists("check.txt"):
    os.remove("check.txt")
else:
    print("The file does not exist")
```

NOTE: OS module can also be used to rename a file.

```
import os
os.rename("check.txt", "renamed.txt")
print("Renamed file!")
```