PYTHON

Contents

Chapter: 1 Basics of Python

Objectives

To discuss the basic of Python by exploring the

- Meaning and application of Python
- Discussing the outlook of lanaguage by mentioning about its interpreter and code editor
- Explaining the basic terminologies and elements of Python by mentioning about procedure of working with Python and the keywords used.

0. Introduction

Python is simply an interactive, general-purpose interpreted, high-level, and object-oriented programming language which was released in 1991 and created by Guido van Rossum. Being a continuously evolving and easy-to-learn language, python is consistently recognized as the most popular language in the world. This chapter will focus on digging deep into the concept of Python by assessing the application of Python, its interpreter, code editors, and writing in the introductory program. Further, basic terminologies and working in Python will be discussed in detail to provide an overview of how a simple program could be returned and what different elements are included while writing the program.

1. Application of python

We live in a world where the usage of software is a major requirement for all types of industries like the military, research, banking, or healthcare. There is a huge list of programming languages available for supporting the requirement but among them, Python is the most exciting and lucrative programming language. Python is a language that is simple to understand and use due to its syntax similar to the English language, free and open source, has the presence of extensive libraries for supporting every imaginable function, and is an interpreted language which can be read line by line and could be dynamically typed. These features made Python among the most popular programming languages and resulted in having its demand by all top companies. In the real world, python could be used for supporting software components, and website development, or to work with machine learning (ML), artificial intelligence (AI), and data science technologies. The top applications of Python in the real world are discussed below.

1.1. Desktop GUI

GUI (Graphical user interface) is the interface that helps in building user interaction with the electronics using audio indicators or graphical icons instead of having text-based details like text navigation of typed command use. As Python is an interactive language, it helps in building the GUI for the user easily and quickly. There are many inbuilt tools in Python such as wxWidgets, kivy, or PyQT library which help in developing functional GUI efficiently and securely.

1.2. Console Based

The console-based application could be stated as the command program which can be used for execution of the program. In old computer generations, console-based applications are very famous. Python has Read Eval Print Loop (REPL) which makes the language suitable for having command-line applications. There is the existence of many IO libraries with Python which contribute towards building command line apps.

1.3. Software development

Features like high compatibility, enhanced code readability and reusability, inbuilt frameworks or libraries, and platform independence contribute towards making Python suitable for software development. Technologies like AI and ML can be integrated with Python for software development. Some of the common applications that are using Python are Reddit, Google, and Netflix.

1.4. Scientific and numeric

Python is identified as the means of simplifying all scientific computing due to libraries like NumPy (for addressing complex numeric calculations), IPython (for recording, editing work

sessions, supporting parallel computing and visualizations), Pandas (for data modelling and analysis), and SciPy (package for supporting engineering, science and mathematics work).

1.5. Business

Python helps in providing scalability and security features which helps in the development of high-performing business applications. There are libraries like Tryton which is an open-source business software with functions like purchasing, financial accounting, shipping, or sales thus making the language suitable for business applications.

1.6. Audio or video-based

The applications for audio and video are the most interesting feature of Python wherein there are libraries like Pyo, SciPy, Mingus, Dejavu, and OpenCV which help in completing the task flawlessly. Some of the applications which are formulated using Python are YouTube, Spotify, and Netflix.

1.7. CAD

CAD (Computer aided design) is defined as the procedure which is used for the creation of 3D or 2D models digitally. The CAD applications majorly are used by construction managers, product designers, or architects for maintaining consistency and designing products. Python has tools like Open Cascade, FreeCAD, or Blender which provide features of dynamic system development, technical drawing, file exporting or importing, and recordings. These efforts help in designing the products efficiently.

1.8. Enterprise

Within an organization or an enterprise too, the Python language could be to support operating needs. Herein, the real-time usage of Python for enterprise applications is like Picalo, Tryton, or OpenERP.

1.9. Image processing

Python is identified as an important tool for image analysis and manipulation. With the presence of open-source dependencies that can be installed using the command line, python helps in the assessment of the images using tools like Pillow, OpenCV, or SimpleITK.

1.10. Web

The web applications for Python focus on the development of webpages using Python tools. There are many protocols with Python like JSON, XML, HTML, BeautifulSoup, Request, or email processing which help in web development. There is the presence of frameworks like Bootle, Flask, or Django which are used by developers for making the process completely effortless. Further, the usage of Python also contributes to having a fast development process, enhancement in security, adding convenience in development, and supporting better visualization.

Python in the real world has bought many changes to industries. the presence of many powerful libraries helps in fulfilling the development requirements of industries, thus, usage of Python is essential for boosting productivity.

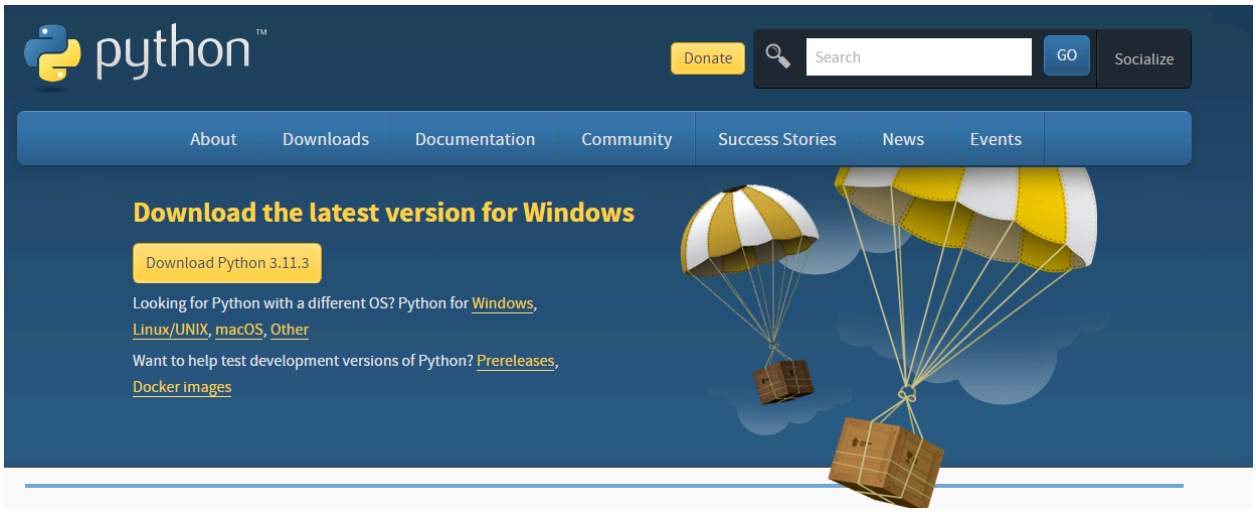
2. Using Python Interpreter

Python interpreter is defined as a computer program that helps in converting a high-end program into machine language. The interpreter is the tool used for translating the program that is typed into the language which the computer can understand. Machine language is represented by bits strings i.e. 1s or 0s and reading this language is difficult for humans. To bridge this gap and make the codes readable for humans and understandable for computers, interpreters are used. The interpreter is different from the compilers. A compiler also contributes to translating, but the functioning is different. With an interpreter, line to line translation is done while with a compiler batch translation is practiced. This saves time while using the interpreter. Usually, the Python interpreter is stated as:

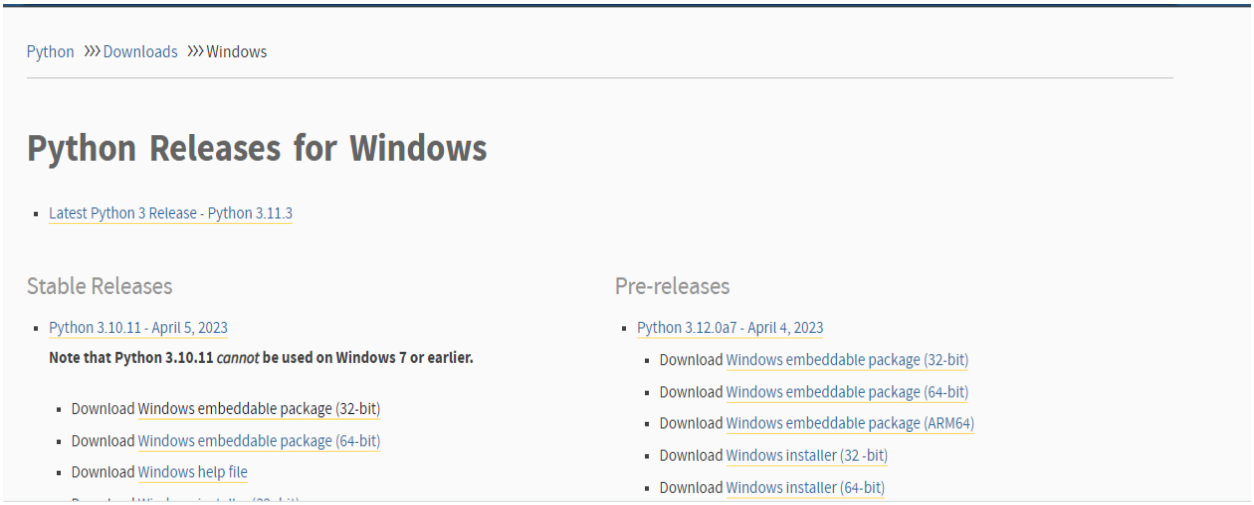
/usr/local/bin/python3.11

Although the interpreter installation location is optional, the stated path is the most used one.

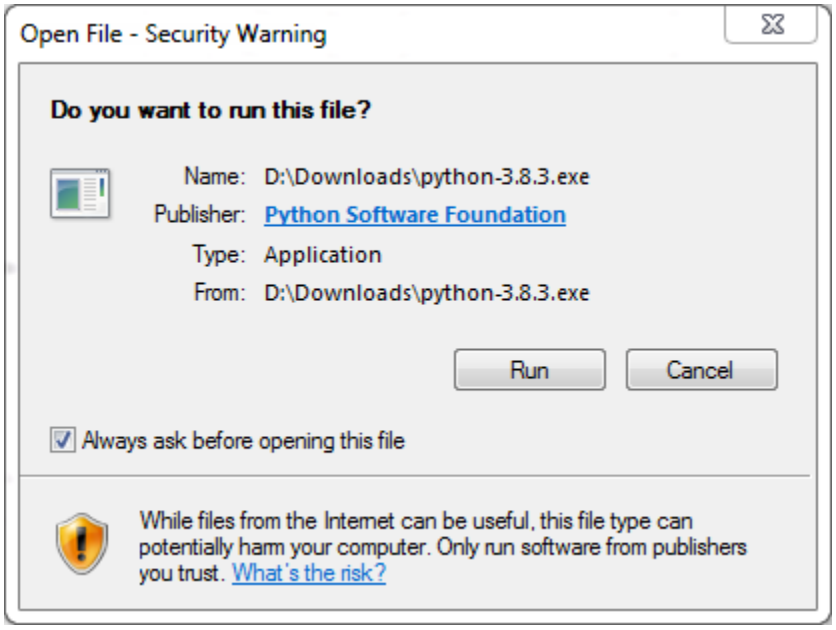
For installing in Python, visit https://www.python.org/downloads/ and download the latest version of Python for your system. Currently, the latest version available for Windows is 3.11.3. Now select the system type i.e. Windows, macOS, Linus/UNIX, or any other.



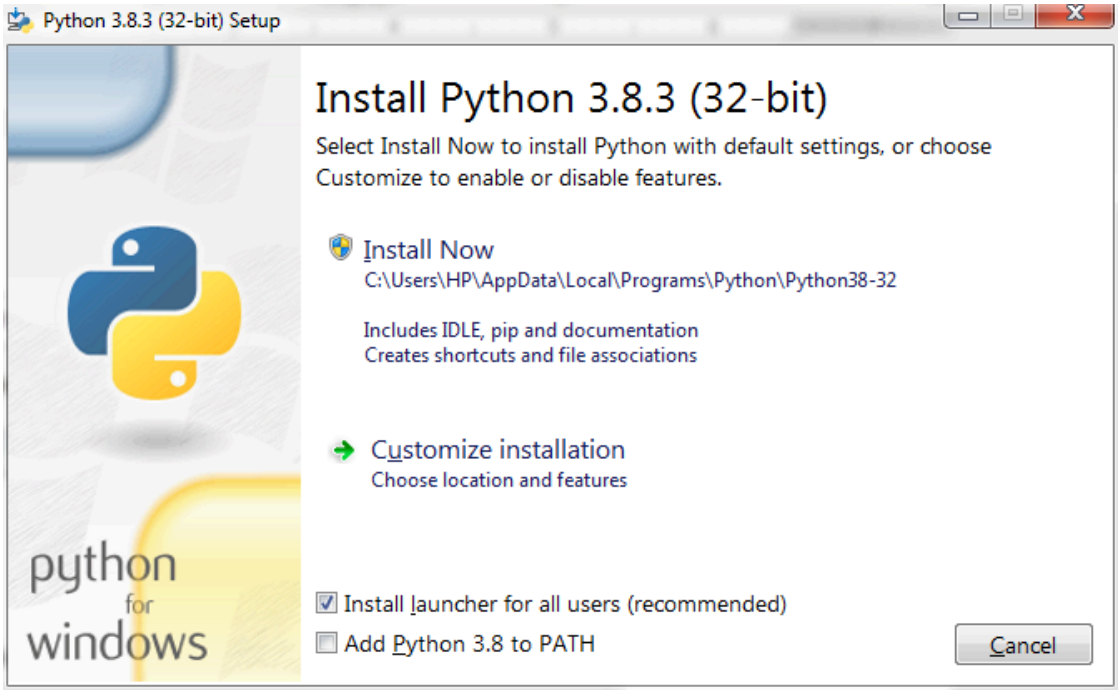
Select the Python version you want to download,



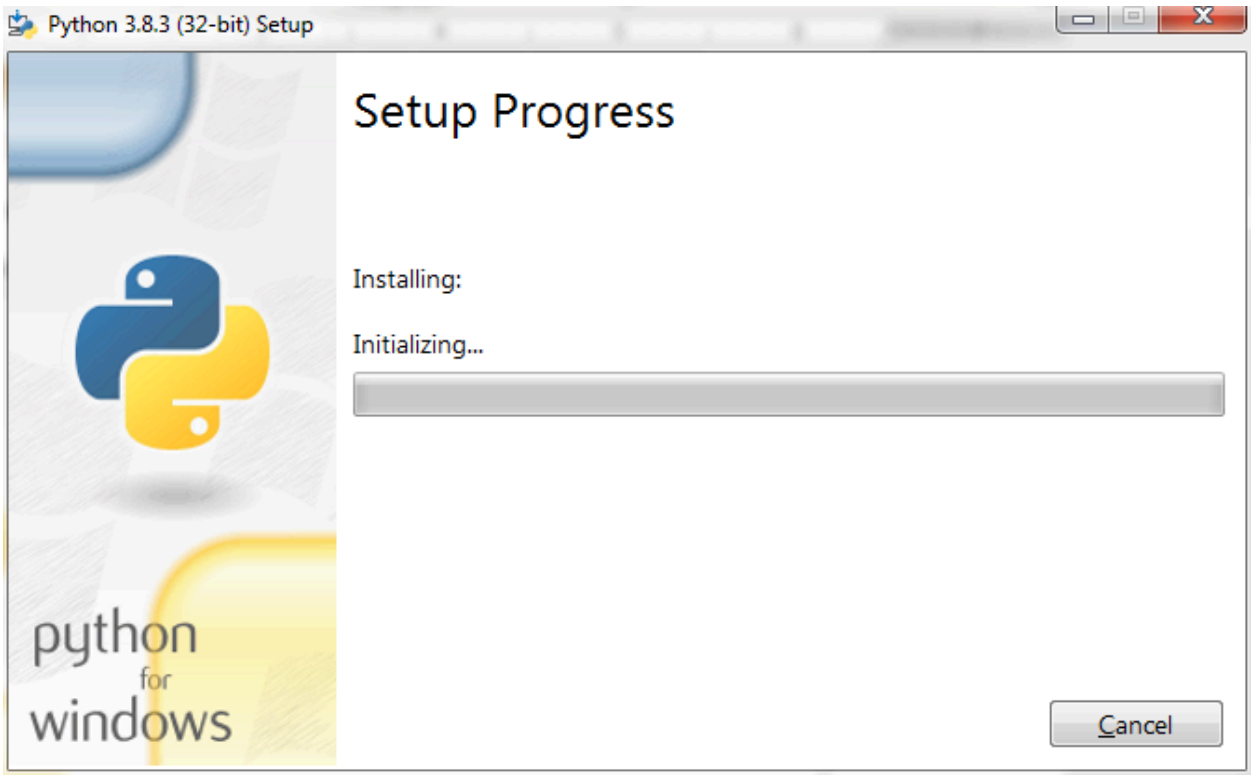
click on the exe file and download it.



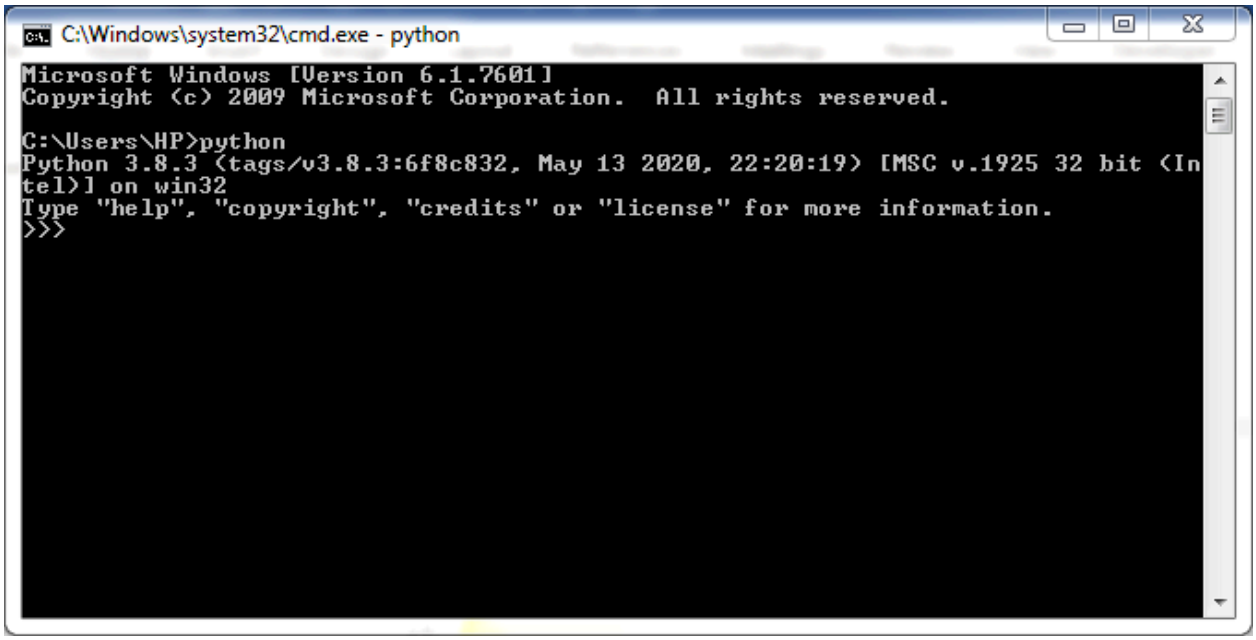
Once the file is downloaded open the file and run it for installation.



By clicking on 'Install Now', the Python interface will be installed for Windows. With this below window will appear.



With this, a Python interpreter would be installed which you can check by opening the command prompt and typing ‘python’.



The Python command represents the version that is present in the system and some of the built-in functions which can be used for more information about the downloaded version of Python.

There are two methods for using in the Python interpreter i.e.

- Argument passing – This method gives the argument based on which the execution is required. Herein, the additional arguments and string names are converted into lists. For example, by importing the sys module, the length of the module is at least 1 with no arguments and strings present. Now if the -m module is used then the sys. argv[0] is set to be the located module's full name.
- Interactive mode – The interactive mode involves command reading using tty. Herein, the next command is based on the primary prompt thus, >>> signs are used.

3. Code editors in Python

A Python code editor is defined as an editor which is used specifically for writing or editing a program. Often the program written is very lengthy and complex which makes Python command line difficult. Thus, code editors are used to keep a track of the entire code and to easily modify and write. There are many code editors which are popularly used i.e. IDLE, Sublime Text,

PyCharm, Visual studio code, Atom, Jupyter, Spyder, PyDev, thonny, or Wing. Among the stated code editors, IDLE is among the most popular code editors as it is suitable for beginners, is free, and is a very capable debugger. Simplerly PyCharm, Sublime Tex, and Visual Studio codes are also free and have features of good functioning, thus, they are also among the popular editors used for Python.

4. Introductory program of Python

Let's start working with Python by writing a very simple program of “Hello World!”. Herein Sublime Text is used as the script for the program. Since the purpose here is only to print the statement, the code is designed by using the print function i.e.

```
print("Hello World!")
```

The code is written on the code editor. Therefore, for running the command and deriving results, the python command line will be used. To execute the Python file, open the command prompt and navigate to the directory where the text file is saved using the "cd" command. Use backslashes to specify sub-folders to define the file path and run the Python script properly. For this, type the file name followed by the file type and click on enter i.e.

```
C:\Users\HP\Google Drive\Riya\2023\May\IAQS>sample.py
Hello World!
```

The output in the Python command line is “Hello World!”.

5. Getting Started with Python

To install Python, use the versions published on the official website of Python https://www.python.org/, to download.

5.1. Python script

Python is the interpreted programming language which uses.py files in text editors to write the codes. These files are then placed into a Python interpreter for execution. For running in a Python file developed in a text editor and named sample.py i.e.

```
print("Hello everyone!")
```

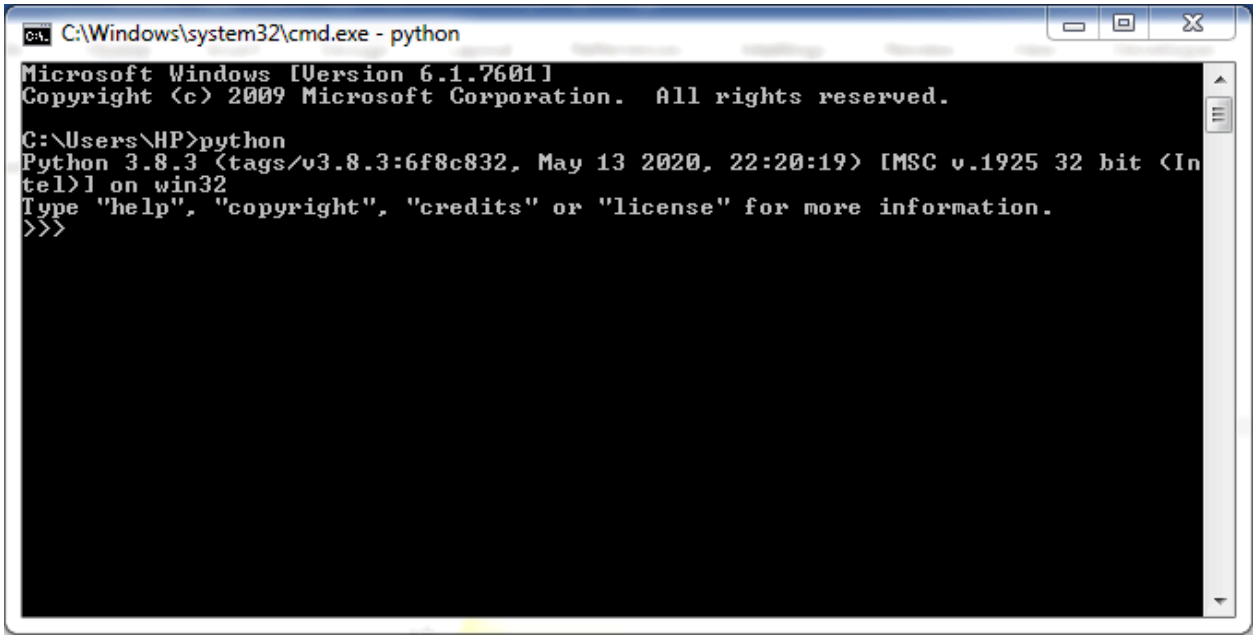
we give the command in the command prompt

```
C:\Users\HP\Google Drive\Riya\2023\May\IAQS>sample.py
Hello everyone!
```

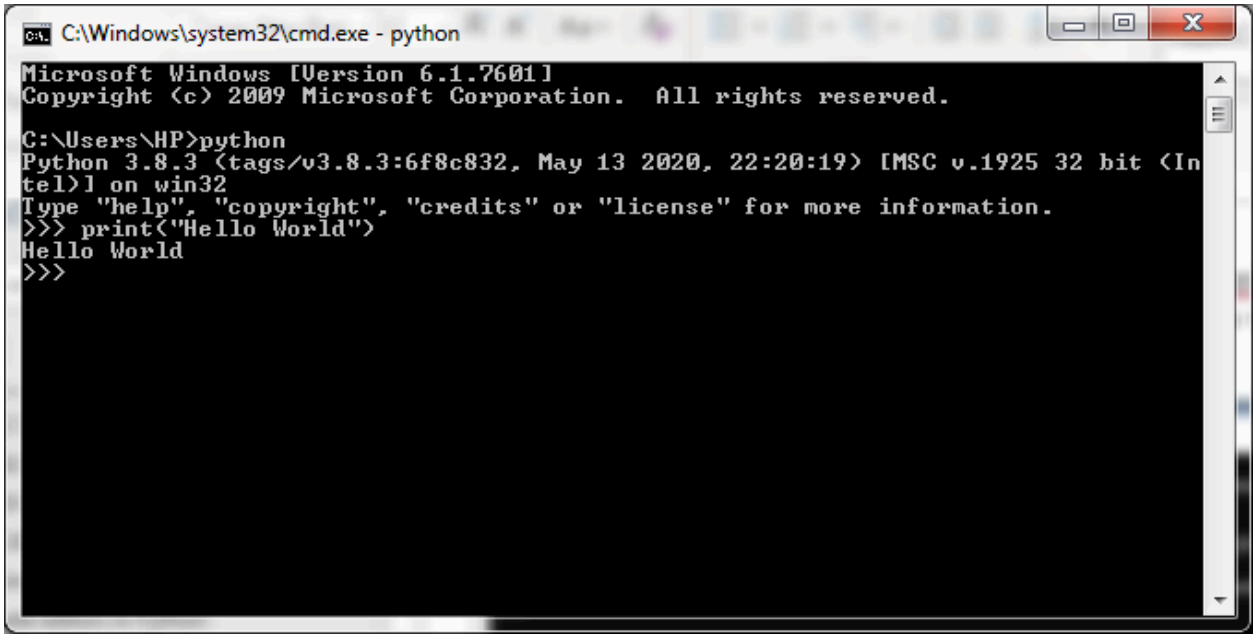
The file which contains all the commands which need to be processed and are structured in the form of a program is known as Python script. Thus, the Python file saved on the text editor is simply called Python script. Python scripts are essential because they simplify complex programs.

5.2. Python command line

Sometimes, instead of running a Python script, the code could be directly written on the Python command line. This could be as with command prompt, python could itself be run as a command line like



Herein, we can write a simple introductory program like printing Hello World.



For exiting in this command line the exit() function can be used.

5.3. Print function

The print function defined by print() is used in Python for printing the specified message. This message could be any object, string, or any other thing written in the code. The syntax for the print function is

$$print(object(s),\ sep = separator,\ end = end, file = file, flush = flush)$$

Wherein the object defines anything which needs to be printed, the separator is the statement of how to separate the objects, the end mentions the point wherein the print will end, the file defines the object with the write method, and flush is a boolean used for mentioning that whether the stated object is True or False.

```
print("I am","", "learning Python")
```

The output for the command would be

```
C:\Users\HP\Google Drive\Riya\2023\May\IAQS>sample.py
I am  learning Python
```

Herein, space is added as the separator.

5.4. Python syntax

For every programming language, there is a set of rules defined for writing the program. These rules are known as syntax. As initially the code written with Python is read by a parser which

would only be able to understand the code if it's written as per the rules. It is therefore essential to understand language syntax. Python syntax consists of:

- Line structure wherein logical lines are added and all blank lines are ignored. New lines mean new statements with python
- The multiline statement defines the method wherein more than one line is used for single statement representation. Herein backward slash or triple quotes could be used.

```
print("Hello\
            World!")
print("""I am
            good""")
```

The output for the above code is

```
C:\Users\HP\Google Drive\Riya\2023\May\IAQS>sample.py
Hello           World!
I am
                good
```

- Multiple statements in one line could be stated by separating the statement using semicolons

```
a=1;print(a)
```

The output for the above statement is

```
C:\Users\HP\Google Drive\Riya\2023\May\IAQS>sample.py
1
```

- For writing string, the single or double quote could be used but just keep the format the same i.e. if started with a single string then end also with a single string.

```
a = 'Finance'
print(a)
```

The output is

```
C:\Users\HP\Google Drive\Riya\2023\May\IAQS>sample.py
Finance
```

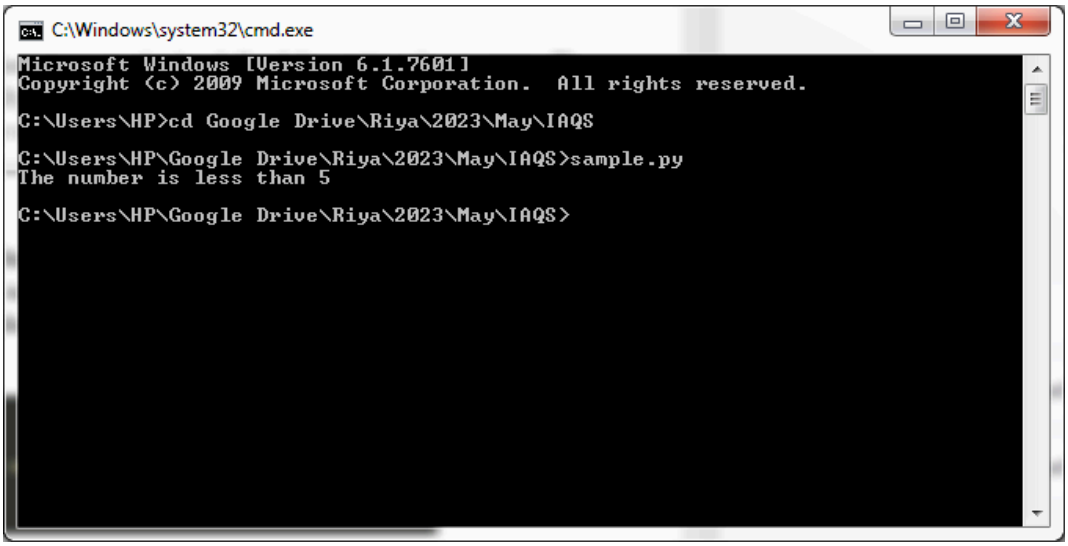
- All the blank lines or whitespaces are ignored by the interpreter

5.5. Python indentation

Indentation is defined as the space which is added at the beginning of the code. In Python, indentation is very important as it defines the block for a function or method i.e. indentation in Python is used for block creation.

```
a = 0
if a<5:
    print("The number is less than 5")
    a = a+1
```

With this indentation, the result would be



In case indentation is not properly placed like in the if' statement no indent is added, then a syntax error is derived.

```
a = 0
if a<5:
print("The number is less than 5")
a = a+1
```

The output would be

```
C:\Users\HP\Google Drive\Riya\2023\May\IAQS>sample.py
  File "C:\Users\HP\Google Drive\Riya\2023\May\IAQS\sample.py", line 3
    print("The number is less than 5")
    ^
IndentationError: expected an indented block
```

Herein, an indentation error is derived representing that the syntax of the code is not properly indented. The number of spaces should remain consistent throughout the block.

5.6. Creating comments in Python

Python gives the option of inserting comments in the code. Comments are simply statements used for explaining the code and are not included as part of the code. The comments help in making the entire code more readable and prevent time wastage in understanding the code while executing or re-checking in case of error or after a long time. The comments in Python are created using the ”#” command and Python will ignore the statement written after it. For multiple line strings the double quotes i.e. ”” could also be used to add comments.

```
a = 0
#the loop is created for running the command until the value is less than 5
while a < 5:
    print("The number is okay")
    a = a + 1
```

Herein, the comment is used to explain what would be done in the while loop.

The output for the code is

```
C:\Users\HP\Google Drive\Riya\2023\May\IAQS>sample.py
The number is okay
The number is okay
The number is okay
The number is okay
The number is okay
```

The while command stated that the operation will continue till the value of a is less than 5 so once the output will be printed for a =0 i.e. “The number is okay”. Further, due to the expression a = a+1, the value of “a” will rise to 1, 2, 3, and 4 leading to “The number is okay” printed 4 more times. Hence, in total 5 times the output is printed as by then the value of a remains less than 5.

6. Basic terminologies with Python

Although codes in Python vary as per the requirement, some of the terminologies used in program building are common. Some of these basic terms are:

- Variables – Python variables are containers used for storing the data values. There is no command present in Python for declaring a variable. The variable is developed with the assignment of values to it.

```
name = "Ranjan"
age = 20
print(name,age)
```

  In the above example, the name is the variable created to store a string value Ranjan while age is the variable developed for storing integer value i.e. 20. Thus, there is no requirement of declaring a variable for any particular type only. It can be changed once a value is set.

- Function – A function in Python is defined as a block of code that only can run when it's called. It is a form of code that can be re-used again for performing a particular action. Instead of copy-pasting the same line of code multiple times, the complex code could be broken down into more maintainable and readable forms. Herein the data (which is known as parameters) could be parsed into a function and the data could be derived from the function as a result.

```
def feature():
    print("I am learning Python")
feature()
```

- Classes – Python classes are defined as the prototype or blueprint in which the objects are created. A "class" is the means of enabling functionality and data bundling. It is useful for creating a user-defined data structure that includes its own data members and functions. The class is developed using keyword class and herein variables developed inside the class are the attributes. For example, a class needs to be developed for students wherein multiple elements are present like course, age, semester, and department. Now, for recording the BBA course we can develop a class student with a course as "BBA" i.e.

```
class student:
    course = "BBA"
```

- Object – A Python object is an instance of the class. Herein "class" is like a blueprint while the "object" is a class copy but with the inclusion of actual values. Objects are not any idea stated but the specific details mentioned under a class. The object would include a state (object attribute or the object properties), behaviour (object methods or one object

response to another), and identity (the unique name for any object used for interacting with other objects).

```
class student:
    course = "BBA"
    age = "20"
    def value (self):
        print("My course is",self.course)
        print("I am", self.age)

Neha = student()
Neha.value()
```

For the given code, the output would be:

```
My course is BBA
I am 20
```

Herein a class is created named "student" having 2 attributes known as class variables i.e. "course" and "age". A method named "value" is developed which helps in printing the values. Now an object named "Neha" is created from the "student" class and using neha.value, we are using "value" method for calling the values stored in "student" class to print.

- Lambda – Python lambda is the keyword that helps create the lambda functions. These are the anonymous functions that can be used for simple calculations. For example – With the inclusion of the x/y function, a lambda function is developed:

```
a = lambda x, y : x/y
print(a(6,2))
```

The output for the code is

```
3.0
```

Herein, the value of a variable is derived to be 3.

- Array – Python array is defined as the specific variable used for storing more than one value at a time. As more than 1 value is recorded, their index values could be used for referring to the number. For example – There are different types of insurances provided by an insurance company such as life insurance, health insurance, general insurance, home insurance, and property insurance. Then the code would be:

```
insurance = ["home","health","life","general","property"]
print(insurance[1])
```

Here, “insurance” is the array with all types of insurance mentioned. For assessing the insurance at an index value of 1, we print the value and the output is “health”.

```
health
```

7. Python Keywords

Similar to any other programming language, python also includes some specified words which have restrictions on where those words could be used. These keywords serve as the building block for the program and thus a programmer needs to be aware of the keywords for creating a program.

7.1. What are keywords

The keywords are defined as the reserve words which have some specific meaning and use in the programming language and these words could not be used for any other thing. Python has the presence of these keywords and there is no requirement to import them. In Python there are many built-in-names too but their application is not reserved and the meaning of some built-in functions (NameError, or NotImplemented) could change, making them less restrictive compared to the keywords and are also different. For Python 3.8, there are 35 keywords such as “False”, “and”, “or”, “None”, “finally”, “yield”, “else”, “as”, “pass”, or “in”.

7.2. How to identify Python Keywords

The Python keywords can be identified using three means: IDE, code in REPL, and syntax error.

- With IDE, all the keywords are highlighted and it would help to differentiate them from other words.
- Using REPL, the keywords could be used by help(). This would provide a list of all keywords. However, for specific keyword information, the word could be passed through a help function like help(“from”). Moreover, a keyword module exists from which the details about keywords could be derived. For exampl, “kwlist” and “iskeyword()” for knowing whether a particular word is a keyword or not.
- Lastly, the “syntax error” presence when a value is assigned to the keywords shows that keyword is used incorrectly. This also exists keywords are used as function name.

7.3. Most common Python Keywords

There are multiple keywords present in Python and they are grouped on the basis of their nature and usage. These groups are:

- Value keywords – The ones consisting of singleton (only one instance of itself) values and can be used multiple times. The keywords are True, False, and None.
    - *True* is the Boolean true value
    - *False* is the Boolean false value.
    - *None* represents the existence of no value and is the default value if no return statement is present.
- Operator keywords – Many operators in Python could be used as keywords like AND ∧ as "and", Not or ¬ as "not", OR or ∨ as "or", IDENTITY as "is", and CONTAINS or ∈ as "in".
    - *and* is used to know whether both right and left statements are falsy or truthy. If both are true then they are truthy else a falsy
    - *or* is used to check the truthiness of at least one of the statements i.e. if one of the statements (right or left) is truthy then the result is truthy.
    - *not* is used for defining the opposite boolean value i.e. conditional statement used for flipping the meaning.
    - *in* is membership operator or containment check keyword used to know whether a particular element is present in the specific variable or set or not.
    - *is* means having an identity check i.e. to know whether two objects are exactly the same or not.
- Control flow keywords – They are used for controlling the flow.
    - *if* is the conditional statement used for building a code block and which operates only when the stated condition is truthy.
    - *else* is the Python keyword consisting of its code block which is used only when the *elif* or *if* condition block is not truthy.
    - *elif* is used for adding multiple conditions i.e. can be used only after an *if* or *elif* statement. There are no restrictions on the number of *elif* usage.
- Iteration keywords – These keywords are used for repeatedly having execution of a statement or block of statements.
    - *for* is combined with *in* keyword and used for specifying the condition until when the loop should continue.
    - *while* is used to continue the iteration of the loop until the while keyword condition is falsy.

- *break* helps in exiting the loop earlier than the normal exiting
- *continue* is used for skipping the iteration of the next loop
- *else* with loops are used to define loops which will continue to run when the loop normally exists i.e. when the break is not called.

- Structure keywords – Used for defining classes or functions in context manager, the structure keywords could be used.
  - *def* is used for defining a method or function of a class.
  - *class* is the one used for defining the class
  - *with* is used for defining the code which needs to be executed under the scope of context manager
  - *as* is used with *with* keyword for accessing the values derived by using *as* keyword as an alias (denoting coded name for a variable).
  - *pass* is used to state that a block is intentionally kept blank. It's like no operation.
  - *lambda* is used for defining the function which has no name but just one statement and based on it the results are derived. A function formulated with the *lambda* keyword is called a *lambda* function.
- Returning keywords –These are used for defining the values which should be returned from the methods or function.
  - *return* is a keyword defined with def and is used for exiting the function and returning the value or result specified after the return keyword.
  - *yield* is similar to the return keyword which mentions what needs to be returned from the function. However, the value derived from the yield keyword is a generator and could be parsed with a built-in-next() function for generating the next value from the function.
- Import keywords – There are many libraries or modules which are not in-built and hence need to be imported into the program. The keywords used for importing are:
  - *import* is a keyword used for importing or including a module from Python
  - *from* is used for importing something specific from a particular module
  - *as* is an alias used for importing a tool or module. The keyword is used along with *from* and *import* keywords. For lengthy and complex names, the keyword helps in naming the keyword as something else which is simpler.
- Exception handling keywords –These are used for catching and raising exceptions. Some of these keywords are:
  - *try* used for raising exceptions and defining what is done if an exception is raised.

    - *except* is used along with *try* to define what can be done if some specific exemptions are raised. With a single try, more than one except blocks could be created.
    - *raise* is used for exception raising
    - *finally* is the specification of code that needs to be compulsorily performed irrespective of the results in *else*, *except*, and *try* block.
    - *else* used along with *try* and *except* means that the *else* keyword could only be used if an exception is not raised with a *try* block.
    - *assert* is used for defining an assertive statement. Herein nothing would be derived if the expression is truthy while an assertion error is drawn when the expression is falsy.
- Variable handling keywords -Are used for working with variables. Some common keywords are-
    - *'del'* used for unsetting a name or variable and commonly used for list or dictionary index removal.
    - *'global'* used for defining a variable in global scope i.e. a variable that can be pulled at any level in the code.
    - *'nonlocal'* is like a *global* keyword used for variable modification from the global scope. Herein nonlocal keyword helps in pulling the variable from the parent scope.

8. Summary

Python is simply an interactive, general-purpose interpreted, high-level, and object-oriented programming language.

In the real world, python could be used for web applications, image processing, enterprise applications, CAD, audio or video-based applications, business applications, scientific and numeric, software development, console-based and desktop GUI.

Many code editors are present for Python, but among them, IDLE, PyCharm, Sublime Tex, and Visual Studio codes are the most popular.

For writing a program in Python, the requirement is to be aware of some basic terms like class, variable, object, array, and lambda.

The keywords are defined as the reserve words which have some specific meaning and use in the programming language and these words could not be used for any other thing. Some common keywords are False, and, or, None, finally, yield, else, as, pass, or in.

Chapter: 2 Anaconda & Pip

Objectives

To set up anaconda environment

- Download and install anaconda
- Management of package and environment with Pip

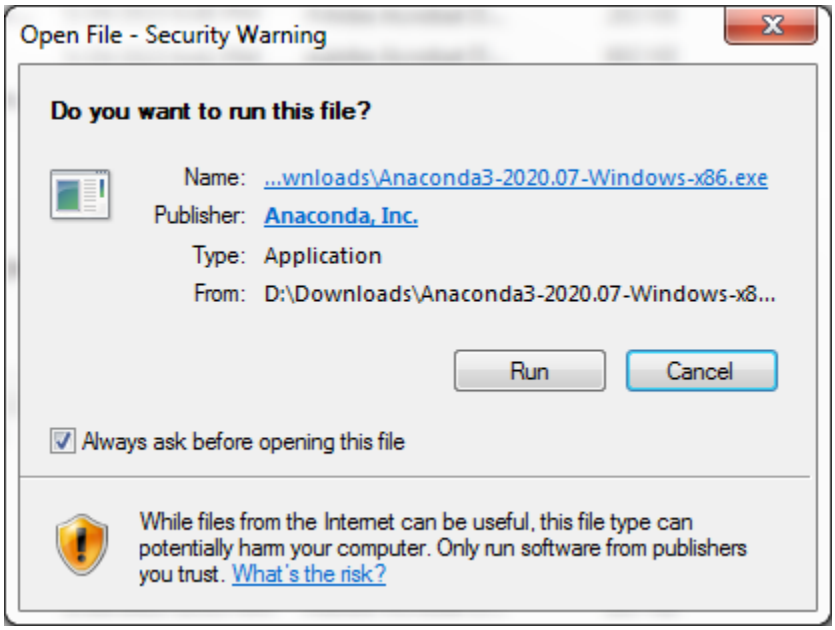
0. Introduction

When starting to learn Python, installation of Python's latest version along with the packages needed for performing different tasks is preferred. However once the needs of the project become more versatile, structured, or complex, a virtual environment is required for organizing different codebases. The virtual environment is simply a networked application that helps the user interact with other users' work and computing environment simultaneously. A virtual environment enables control of as many packages as needed, thus, a more reproducible, portable, and stable environment is available for coding. Some of the popular packages are Virtualenv, Anaconda (popularly known as Conda), and Pipenv. Among them, Anaconda is the most popular open-source virtual environment which consists of Spyder, Jupyter, or other notebooks which are required for scientific computing, data analytics, or large data processing. This chapter explains the conda installation process along with discussing the packages used for environmental management to ensure the availability of a base for working with complex data.

1. Installation of Anaconda

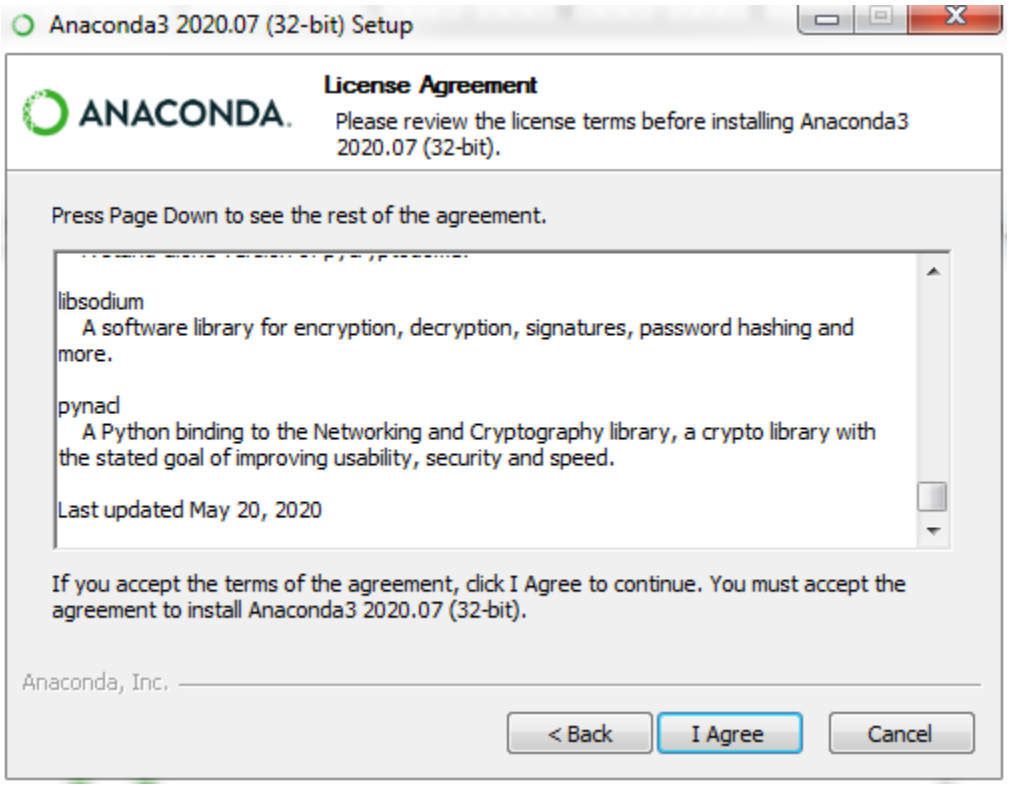
To start working with Python, install Anaconda's latest version, i.e., at least version 3.7. The steps for installing the Anaconda software are as follows.

- Download Anaconda installer for Windows or Mac as per the requirement
- In the downloads folder, double-click on the downloaded exe file for Anaconda and click on it to launch.

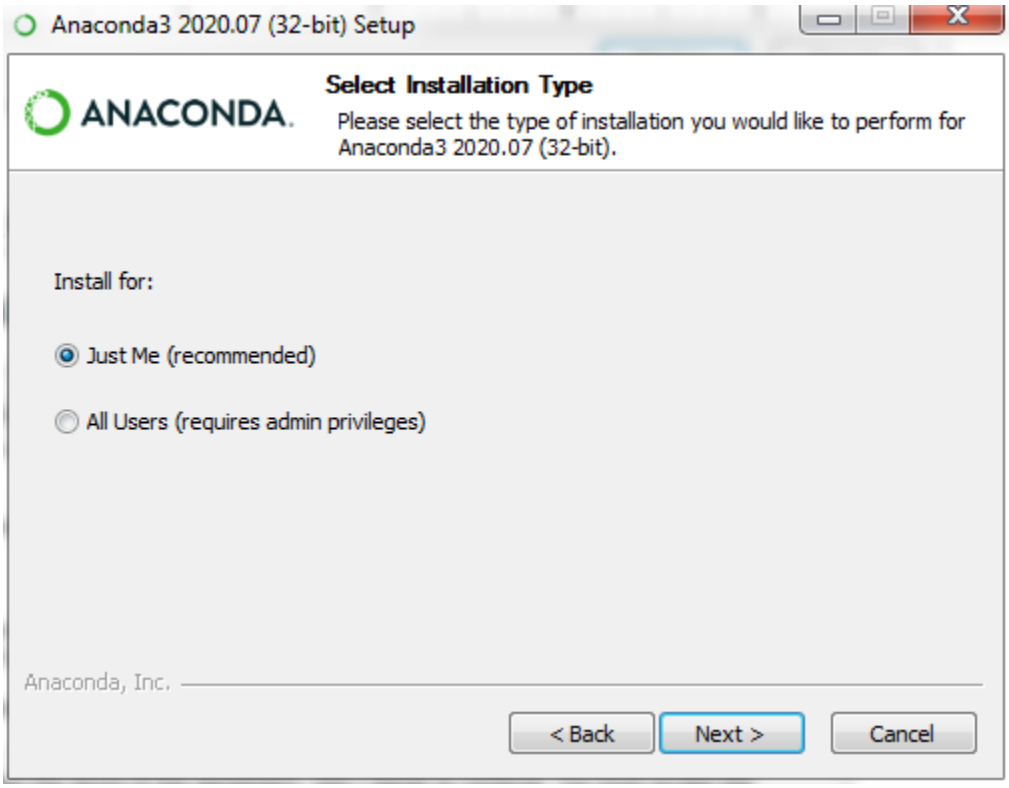


- The setup wizard window will appear. Click on 'Next'.

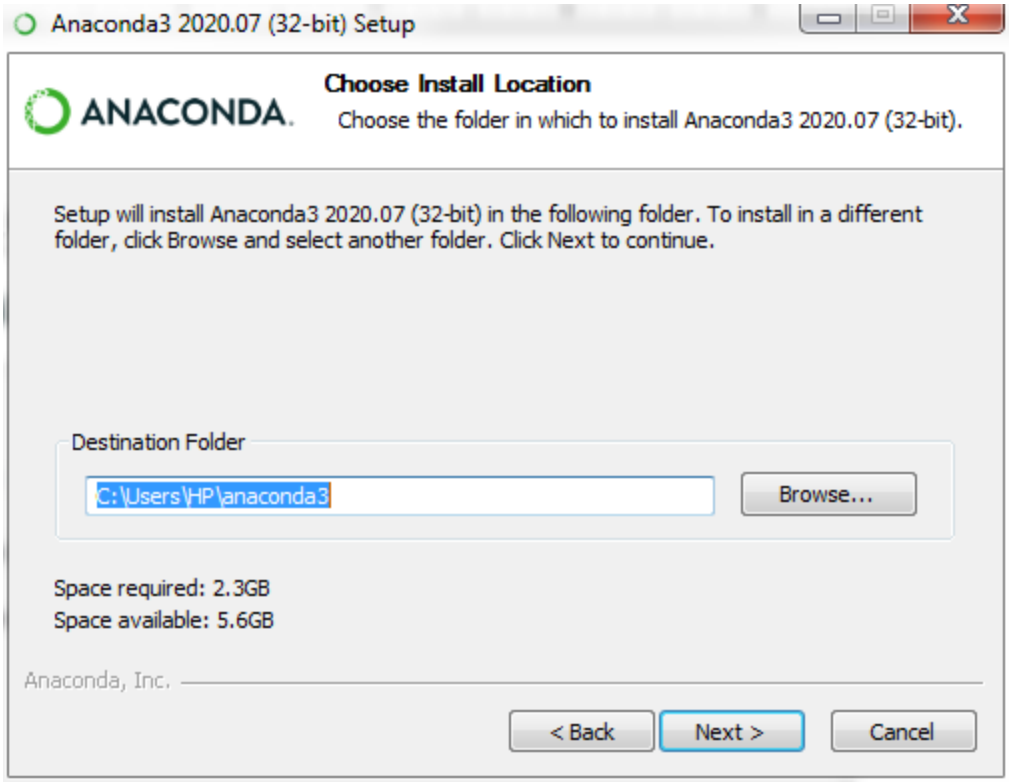
- The license agreement window will appear. Read all the licensing agreement terms and then click on 'I agree'.



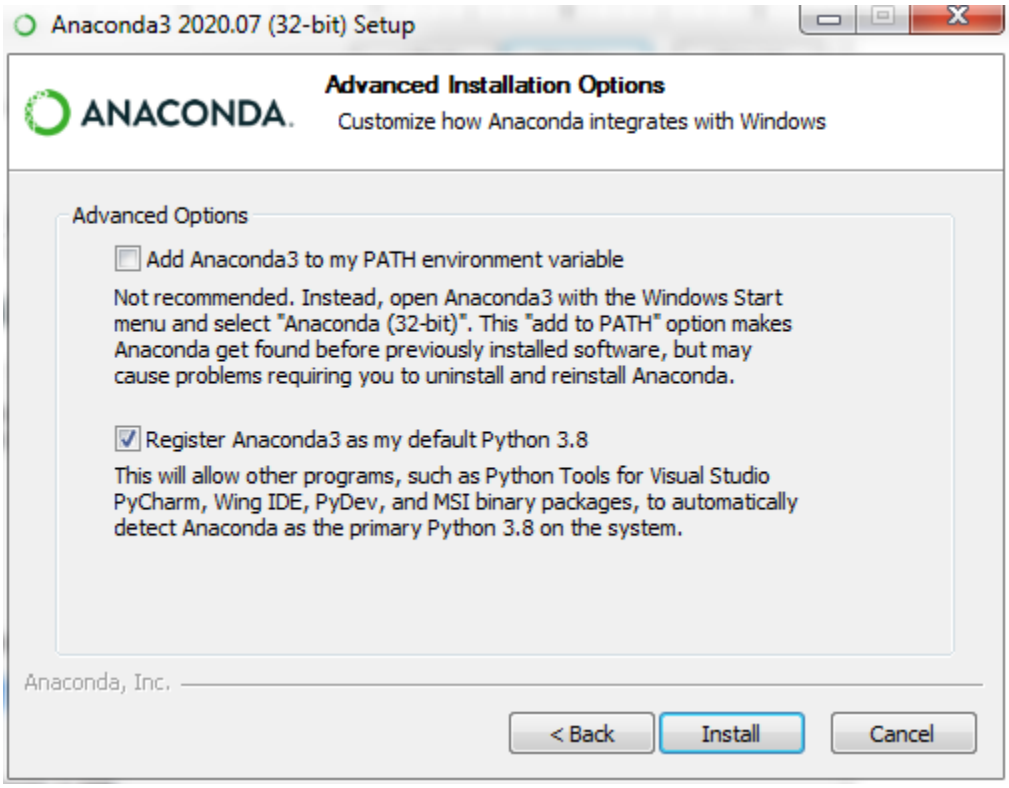
- Once the licensing requirement is completed, select the installation type you need i.e. for only yourself or for others too. It's recommended to install it for yourself. Select all users if you want to install it for all users on the computer. After this selection click on ‘Next’.



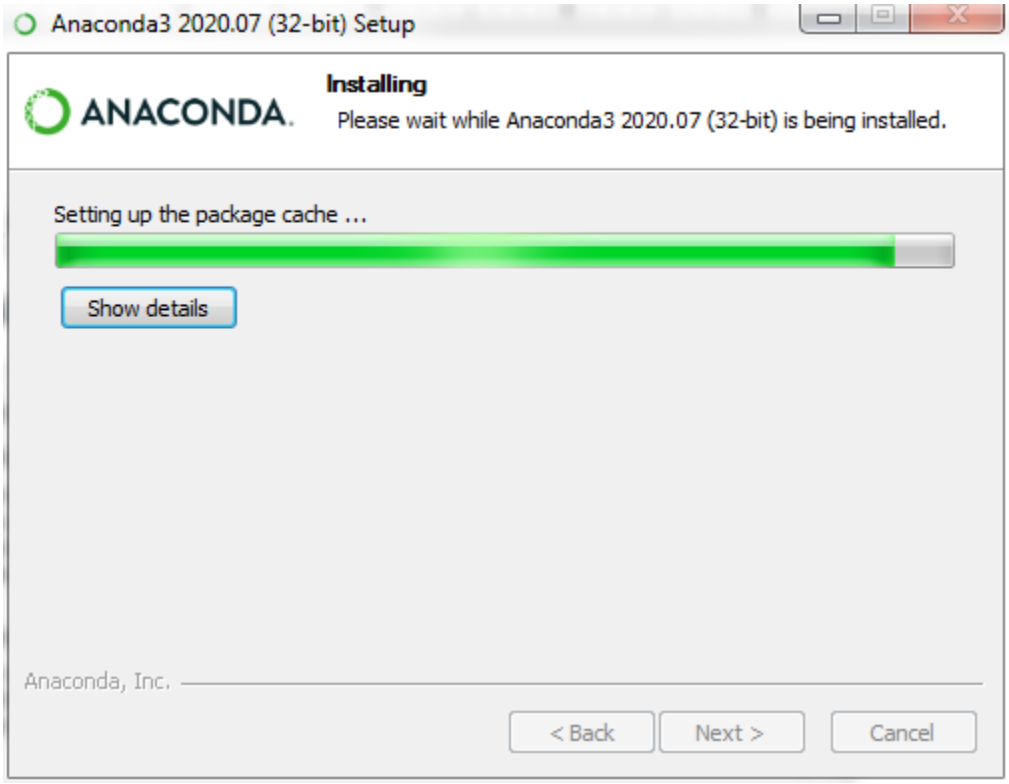
- Select the destination folder for the installation and then click on ‘Next’.



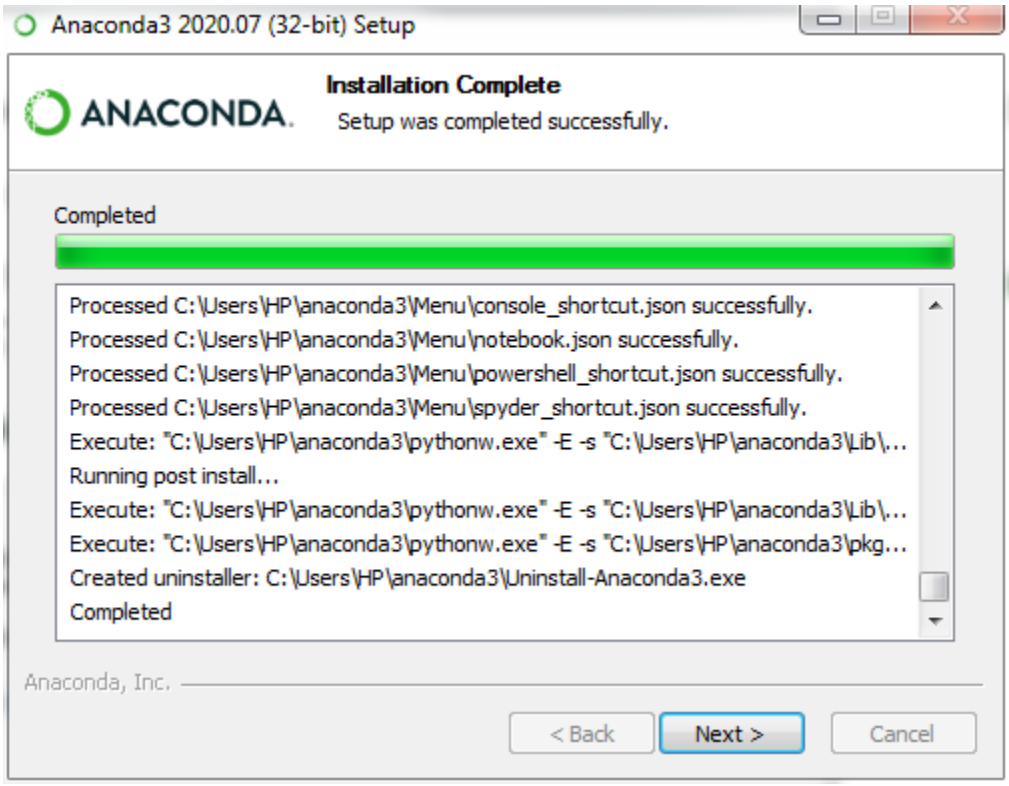
- There is an option to install Anaconda2 in the path environment, however, it's not recommended. So, just click on ‘Install’.



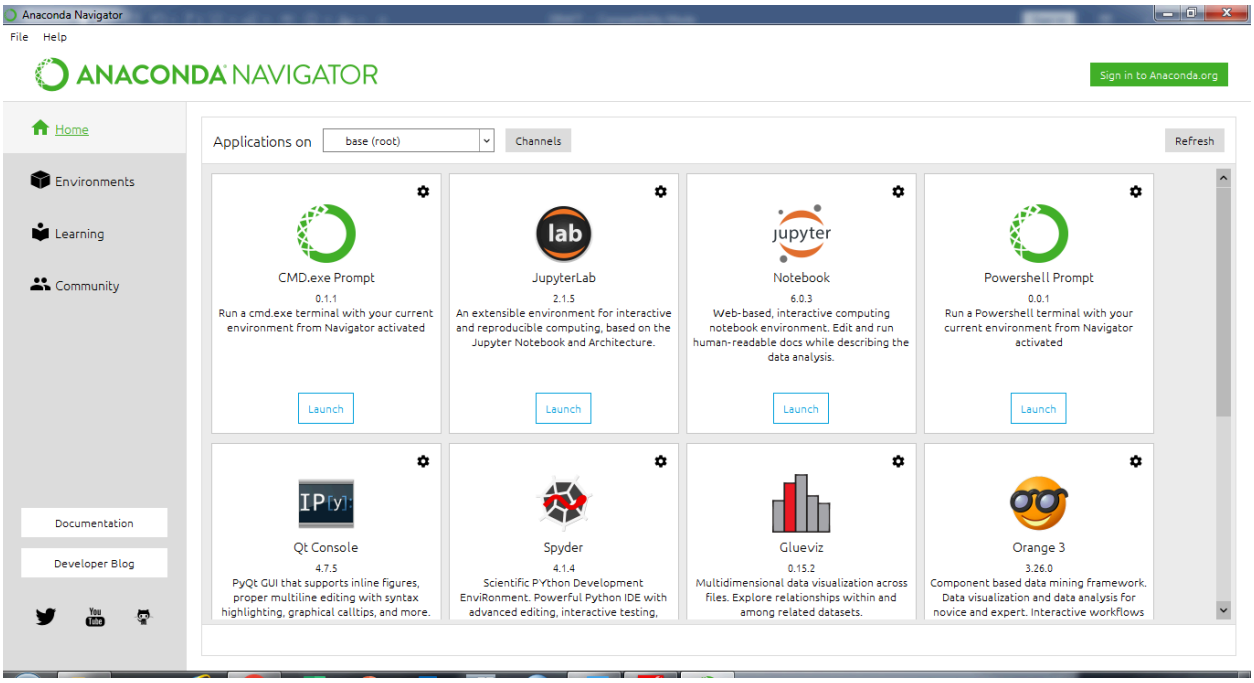
The installation will begin.



- Once the installation is completed, the window will appear. Click on ‘Next’.



- Completing these steps the final windows for installation will open the windows of Anaconda completion and there click on ‘Finish’.
- Open the Anaconda navigator. The virtual environment with the anaconda could be seen as shown below.



Following these steps, the Anaconda environment is installed on Windows.

2. Packages

With programming, it can be difficult to decide which package to use for multiple works. There are multiple packages available like Anaconda. Conda is a powerful package and environment manager which uses the Anaconda prompt command line in Windows or a terminal window in Linus or macOS. In Python, pip is a standard package manager which can be used outside and inside of Anaconda for installing and managing additional packages which are not part of the Python package index. Though conda and pip seem to be identical package management tools, they are designed for different purposes. Pip is an official and recommended tool for installing packages from the Python package index (PyPI) and GitHub, while conda is used for installing packages from the Anaconda cloud and Anaconda repository. Conda cannot be used for installing packages from GitHub but we can use Conda for pip installation. There are 150000 packages with PyPI while only 1500 with the Anaconda repository. In case a package is needed

and not available on the Anaconda repository then it can be installed with pip. Moreover Pip installs Python packages while conda installs software written in any language. Thus, before using pip, a Python interpreter needs to be installed.

For example – The available libraries in the anaconda environment could be checked using conda list –explicit.

```
conda list --explicit
```

```
https://repo.anaconda.com/pkgs/main/win-32/libtiff-4.1.0-h56a325e_1.conda
https://repo.anaconda.com/pkgs/main/win-32/libxslt-1.1.34-he774522_0.conda
https://repo.anaconda.com/pkgs/msys2/win-32/m2w64-gcc-libs-5.3.0-7.tar.bz2
https://repo.anaconda.com/pkgs/main/win-32/python-3.8.3-he1778fa_3.conda
https://repo.anaconda.com/pkgs/main/win-32/qt-5.9.7-vc14h73c81de_0.conda
https://repo.anaconda.com/pkgs/main/win-32/curl-7.71.1-h2a8f88b_1.conda
https://repo.anaconda.com/pkgs/main/win-32/pywin32-227-py38he774522_1.conda
https://repo.anaconda.com/pkgs/main/win-32/menuinst-1.4.16-py38he774522_1.conda
https://repo.anaconda.com/pkgs/main/win-32/argh-0.26.2-py38_0.conda
https://repo.anaconda.com/pkgs/main/win-32/asn1crypto-1.3.0-py38_0.conda
https://repo.anaconda.com/pkgs/main/win-32/bitarray-1.4.0-py38he774522_0.conda
https://repo.anaconda.com/pkgs/main/win-32/boto-2.49.0-py38_0.conda
https://repo.anaconda.com/pkgs/main/win-32/certifi-2020.6.20-py38_0.conda
https://repo.anaconda.com/pkgs/main/win-32/chardet-3.0.4-py38_1003.conda
https://repo.anaconda.com/pkgs/main/win-32/comtypes-1.1.7-py38_1001.conda
https://repo.anaconda.com/pkgs/main/win-32/console_shortcut-0.1.1-4.conda
https://repo.anaconda.com/pkgs/main/win-32/docutils-0.16-py38_1.conda
https://repo.anaconda.com/pkgs/main/win-32/entrypoints-0.3-py38_0.conda
https://repo.anaconda.com/pkgs/main/win-32/fastcache-1.1.0-py38he774522_0.conda
```

Once the list is checked, install pip and other pip packages like herein for NumPy:

```
conda install pip
```

```
Collecting package metadata (current_repodata.json): ...working... done
Solving environment: ...working... done

# All requested packages already installed.


Note: you may need to restart the kernel to use updated packages.
```

```
conda install NumPy
```

```
Collecting package metadata (current_repodata.json): ...working... done
Note: you may need to restart the kernel to use updated packages.

Solving environment: ...working... done

# All requested packages already installed.
```

Thus, it is beneficial to use a virtual environment as it helps in managing dependencies but ensures that the package remains restricted to the environment and doesn't affect other scripts.

There is a need to install both pip and conda requirements so that packages that can’t be installed with conda can be used.

3. Summary

The virtual environment is simply a networked application that helps the user interact with other users' work and computing environment simultaneously.

Some of the popular packages are Virtualenv, Anaconda (popularly known as Conda), and Pipenv.

Anaconda is the most popular open-source virtual environment which consists of Spyder, Jupyter, or other notebooks which are required for scientific computing, data analytics, or large data processing.

Conda and pip seem to be identical package management tools but they are designed for different purposes.

Chapter: 3 Creating Python Variables

Objectives

To discuss the concept of Python variables

- Explore the Python variables meaning along with discussing the means to create it, types, or the technique of assigning name
- Discuss the types of data present along with mentioning the means of deciding data for the variable

0. Introduction

One of the important aspects of working with any programming language is to understand how to store and manipulate data. This chapter discusses this feature of Python by examining Python variables, their types, and the procedure of naming variables along with assessing different built-in data types present in Python.

1. Python Variables

Python variables are defined as the containers used for storing the values. In simple terms, it's the space that you keep in memory. There is no explicit command which needs to be entered for the creation of a variable. The sign of equal to (=) is used for the value assignment. For example – Two variables are created i.e. name and age:

```
name = "Dinesh"
age = 20
```

Herein, 'name' and 'age' arde the variable name while 'Dinesh' and '20' are the variable values.

Python also allows the multiple assignments of a single value for different variables by using equal signs simultaneously. This could be written as:

```
x = y = z = 10
```

Python also provides the facility of deleting a created variable by using 'del'. For the above code, in case deletion of the variable 'message':

```
message = "I am learning Python"
print(message)
del message
print(message)
```

The output would be:

```
C:\Users\HP\Google Drive\Riya\2023\May\IAQS>sample.py
I am learning Python
Traceback (most recent call last):
  File "C:\Users\HP\Google Drive\Riya\2023\May\IAQS\sample.py", line 4, in <mod
le>
    print(message)
NameError: name 'message' is not defined
```

1.1. Types of variables

As Python is completely an object-oriented language and not a typed one, there is no need to declare a variable or its type. Python is an inferred language, so it can derive the type of variable by examining the stored data.

While working with the variables, different types of variables are discussed i.e.

- Numbers – This variable type is used for storing numbers. There are two types of numbers; floating point numbers which consist of decimals and integers which are whole numbers. 'Height' is a floating point while 'age' is an integer.

```
height = 133.5
age = 20
```

- String – The variable used for storing text is called a string variable. It could be defined by using a single or double quote. In the below example 'name' and 'education' are two string variables defined using double and single quotes respectively.

```
Name = "Ankit"
Education = 'Graduate'
```

The variable types can also be classified based on the location for which the variable is created. The variable which is defined inside a function is called a 'local variable' that cannot be accessed outside the function. For example, if a code is written as below wherein the product of two variables i.e. 'a' and 'b' needs to be derived.

```
def product(a,b):
    product=a*b
    return product
print(product(3,4))
```

In the above example the value of 'a' and 'b' is not defined outside the function and the value is called using function only. Therefore, the variables 'a' and 'b' are local variables that can only be used concerning function product.

However, in case when a variable can be accessed within any function without any restriction then the variable is said to be global. The case of a local variable can be rewritten in a global variable as:

```
a = 3
b = 4
def product():
    product=a*b
    return product
print(product())
```

Here, as the variables 'a' and 'b' are defined outside the function product, they are global variables and can be used anywhere in the code without having any boundary.

1.2. Variable name

A variable name is simply defined as the identifier for the location wherein data is stored. In Python, every variable needs to be unique and thus, there is a set of rules which need to be followed for naming the variables like:

- The name of the variable should not be the same as the keywords of the language.
- The variable name is case-sensitive thus, 'name' and 'Name' are two different variables.
- The starting of a variable name should be from an underscore character or a letter.
- The variable name must only consist of the underscores or alpha-numeric characters (0-9, A-Z, a-z, and _).
- No special character or number can be used as the variable name starting i.e. a variable name cannot be '1age' or '_email'.

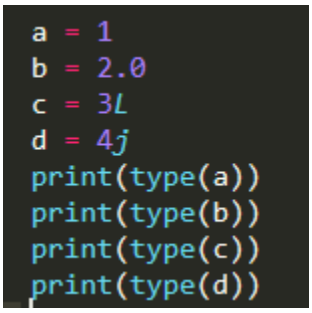
2. Python Data types

The data types in Python are defined as the means of measuring the variable type. It generally explains the type of data stored within a variable. For example – the address stored in a variable would define alphanumeric characters while the experience of an employee is the numeric variable.

2.1. Built-in data types

Python has the presence of many built-in data types. These built-in data types could be used to store different types of values. Therefore, the knowledge of each data type is relevant. The built-in data types are as follows.

- Numeric –the one wherein numeric values are stored. There are generally four types of numeric values i.e. integers, complex, float, and long (the values exist in Python 2.x but are depreciated in Python 3.x). The sample for each type is



For the given values, the type could be derived which represents each numeric type. As with Python 3.x *long* data type doesn't work, its command can be marked as comment and the output for remaining code will be derived as

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

- String – The string data type consists of the character's contiguous set which is represented using quotes. Python allows the use of double or single quotes. The sample code for each of the functions is presented below.

```
a = "Learning"
b = "Python"
print(type(a))
print(type(b))
print(a+b)
print(a*2)
print(b[2])
print(b[2:4])
```

The output for the code is given below.

```
<class 'str'>
<class 'str'>
LearningPython
LearningLearning
t
th
```

The code shows the type of a and b, concatenates both the variables, repeats the variable a, provides the value of b at index 2, and also slices the b variable from index value 2 to 4.

- Sequence - The sequence data types could be of three types i.e. list, tuple, and range. The list data type is the most versatile one wherein items are separated by a comma and written in square brackets []. Tuple data type is another sequence wherein enclosed with parenthesis (), the values are separated by a comma only. Lastly, the range is the in-built function that returns the number sequence starting from 0 and increasing by 1 until the desired point is derived. The code for each step is given below.

```
a = [1, 2]
b = [3]
c = (4, 5)
d = (6)
print(a)
print(a[1])
print(a+b)
print(b*2)
print(c)
print(c[1])
print(d*2)
for i in range(1,10,2):
    print(i)
```

In the above code, 'a' and 'b' are lists while 'c' and 'd' are tuples. As tuples can't be updated, concatenation for them cannot be done. Herein, the value from the range is computed with a starting value of 1 while the ending point as 10, and an increment level of 2. The output for the functions is given below.

```
[1, 2]
2
[1, 2, 3]
[3, 3]
(4, 5)
5
12
1
3
5
7
9
```

- Binary – The binary built-in data type consists of three things i.e. bytes, bytearray, and memoryview. The bytes and bytearray are the means of manipulating binary data while memoryview is used as the buffering protocol for accessing the other binary objects' memory without making any copy. The bytes are similar to strings just that with bytes prefix 'b' is added. Bytearray is created by the bytearray() constructor and they are mutable objects.
- Mapping – The Python dictionary is the one used for representing mapping data type. It is a hash table type that works like associative arrays which consist of key and value pairs. Enclosed using the curly braces i.e. {}, with dictionary the values could be accessed and assigned using square brackets []. The sample for the mapping is

```
Map = {1:'a', 2:'b', 3:'c', 4:'d'}
print (Map)
print(Map[1])
print (Map.keys())
print (Map.values())
```

Herein, the output of the code prints the dictionary, the value of key 1, all keys, and all the values.

```
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
a
dict_keys([1, 2, 3, 4])
dict_values(['a', 'b', 'c', 'd'])
```

- Boolean – The boolean data type is the built-in data type that can be used for the representation of the variable by two values i.e. True or False. The bool() function is the one that helps in the evaluation of the variable and has the results in the form of True or False.
- Set – The set in Python is defined as an unordered collection of the items which are enclosed using the curly braces {}. The Python set elements are mutable but the elements cannot be duplicated. Another form of set built-in data type is a frozenset which is defined as the immutable form of the set wherein no removal or addition could be done. The sample code for the types is

```
elements= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
print(elements)
print(type(elements))
for i in elements:
    print(i)
elements.remove(0)
print(elements)
frozen = frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
frozen.remove(0)
```

Herein, elements are the set which is initially printed along with its type mentioning, and looping through each element to print each value of the set. Further, as the set is mutable thus a value is removed, and then again set is printed. Now for another frozenset the feature of remove is tried but as it's not mutable thus the error is derived i.e.

```
    frozen.remove(0)
AttributeError: 'frozenset' object has no attribute 'remove'
```

Now, remove the frozen.remove(0) command from the code we can run the code

```
elements= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
print(elements)
print(type(elements))
for i in elements:
    print(i)
elements.remove(0)
print(elements)
frozen = frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
print(frozen)
```

and the results would be

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
<class 'set'>
0
1
2
3
4
5
6
7
8
9
{1, 2, 3, 4, 5, 6, 7, 8, 9}
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

- None – None is the data type that is used to show that an object consists of no data. Though a variable None could be assigned but there is also the presence of methods that are used for calling the None type.

2.2. Working with data types

The specification of the built-in data types mentions that there is the presence of data of various natures. Sometimes it's known to the coder while sometimes not known. Thus, there is the presence of a function to check the type of data. The type() function is the one which can be used for determining which type of data is stored in a variable. The example code for using the function is

```
data = "numeric"
number = 10
value = 15.0
print(type(data))
print(type(number))
print(type(value))
```

Herein, the type for each of the variables could be derived using the type() function i.e.

```
<class 'str'>
<class 'int'>
<class 'float'>
```

Once the type is derived, often while working the requirement could be to convert the data type from one form to another. This process of conversion is known as data type conversion. These conversions could be of different types as shown in the below example

```
a = 10
b = 11.0
c = "4"
print(int(c))
print(int(b))
print(float(a))
print(float(c))
print(str(a))
print(str(b))
```

The output for the above code is

```
4
11
10.0
4.0
10
11.0
```

- Conversion to int – Herein, b is float and c is a string that is converted into an integer using int().
- Conversion to float –The example has a as an integer and c as a string which is converted into float using float().
- Conversion to string – The integer value of a and the floating value of b are converted into a string using str().

There is even the presence of many functions like int(), long(), float(), complex(), str(), tuple(), list(), set(), dict(), frozenset(), oct(), and hex(); which can be used for setting the data type. Thus, Python allows the creation of variables with the inclusion of the variable type needed by the user.

3. Summary

Python variables are simply defined as the containers used for storing the values.

There are different types of variables like numeric and string which help in storing various types of data

The numeric variable is used for storing integer or floating values while the string is for textual data.

Based on the location of the storing variable, the Python variables are classified as local and global.

Local variables can’t be accessed outside functions while global can be.

The variable name is simply defined as the identifier for the location wherein data is stored, still, every variable needs to be unique in Python.

A set of rules is specified for defining variable names.

A mnemonic naming procedure is recommended while naming variables

The data types in Python are defined as the means of measuring the variable type.

The built-in data types are numeric, string, binary, sequence, mapping, none, set, and Boolean.

The built-in data types provide the feature of converting the data type, setting the type of a value, and also determining the type of a variable.

Chapter: 4 Numeric, string and logical operations

Objectives

To discuss the operations with Python

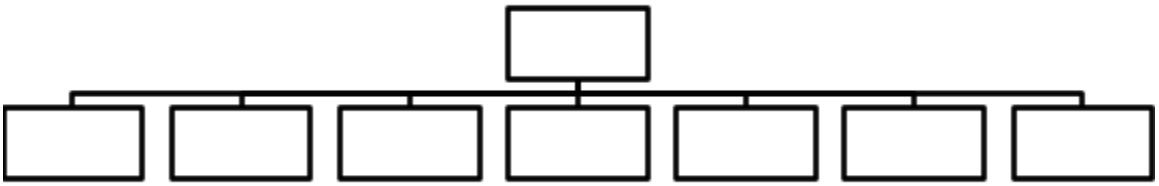
- Examine the numeric operations by discussing its operators and functions
- Explore the string operations and the methods used for functioning
- Describe logical operations used for looping, comparison, and performing logical actions for sequences and sets

0. Introduction

In Python, operators are used for performing different operations for variables and values. Some specific symbols are used for defining the purpose of the mentioned arithmetic or logical operations. In this chapter, a discussion of the types of operations along with mentioning the functions and methods is provided.

1. Operations

The symbols used in Python for describing the computation are referred to as the 'operator'. The values derived from the operator are recognized as operands. The sequence of operators and operands simply is defined as the expression. Python language supports the usage of different operators for combining many data objects into expression form. There are generally 7 operators as shown below.



Among the given operators,

- Arithmetic operators are used with the numeric values for performing operations,
- The assignment operator is used to assign values,
- The comparison operator helps in comparing values,
- Logical operators support by adding conditional statements,
- The identity operator enables object comparison in case the object is the same and from same memory location,
- Membership operators test whether the sequence is present for an object, and
- Bitwise operators are used for binary values comparison.

Another form of categorization for the operators is as per their nature i.e. numeric, string, or logical. The detailed discussion of each of the operators under the mentioned data nature is explained below.

1.1. Numeric

Python has full arithmetic operators set which helps in working with numeric values. The processing with numeric data is based on using arithmetic, assignment, and comparison operators. Arithmetic operators are used for performing arithmetic operations. The operators included for working with numeric values are discussed in the below table.

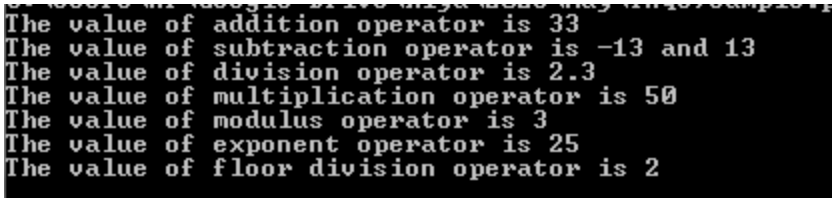
Name of operator	**Symbol**	**Description of the operator**

Addition	+	The operator is used for adding two operands
Subtraction	-	The operator enables subtraction of the 2nd operand from the 1st operand. In case 1st operand is less than 2nd operand, the result would be negative.
Multiplication	*	The operator helps in multiplying one operand with another
Division	/	The operator provides the quotient after dividing 1st operand with the 2nd one.
Remainder of Modulus	%	The operator provides the remainder value after having a division of 1st operand with 2nd
Exponent	**	The operator defines the 1st operand power to the 2nd operand
Floor division	//	The operator provides the quotient when 1st operand is divided by 2nd one by rounding the quotient value to the smallest whole number

Using each of the stated numeric operators, the code is developed with majorly 4 variables i.e. a, b, c, and d. The code for the variables is stated below

```
a = 10
b = 23
c = 5
d = 2
e = a+b
f = a - b
g = b - a
h = b/a
i = a * c
j = b%a
k = c**d
l = b//a
print("The value of addition operator is", e)
print("The value of subtraction operator is", f, "and", g)
print("The value of division operator is", h)
print("The value of multiplication operator is", i)
print("The value of modulus operator is", j)
print("The value of exponent operator is", k)
print("The value of floor division operator is", l)
```

The output for the code is presented below.



The numeric operators helped in simple arithmetic calculations. The division operator had a value of 2.3 but floor division enables rounding the value to the smallest whole number and therefore, the result is 2. For the subtraction operator too, the value of 'b' is initially more than 'a', therefore the operand is negative while positive for the second case when 'a' is subtracted from 'b'.

Apart from including arithmetic operators, the numeric value also requires assignment operators. With the usage of assignment operators, the right side value to the operator is stated as an expression while the left side operand is variable. The expression with assignment operator use is assigned to the variable. The different assignment operators which can be used in Python are:

Name	**Symbol**	**Description**
Assignment operator	=	Assignment of right expression to left operand
Addition assignment	+=	Adding the right operand value to the left operand, the left operand gets a new value
Subtraction assignment	-=	Reducing the left operand value by the right operand value, the left operand value is modified
Multiplication assignment	*=	Multiplying the left operand value with the right operand, the value of the left operand is changed
Division assignment	/=	Dividing the left operand value by the right operand, the quotient becomes the new value of the left operand
Remainder assignment	%=	Dividing the left operand by the right operand, the remainder is accepted as the new value of the left operand
Exponent assignment	**=	Computing the value of left operand power to the right operand, the newly derived value is of the left operand
Floor division assignment	//=	The rounding to the smallest whole number of the quotient of the left operand division by the right operand is assigned as a new value to the left operand

For the same values as arithmetic operators, the assignment operators are applied and the modified code is designed:

```
a = 10
b = 23
c = 5
d = 2
a+=b
print("The value with addition assignment is ", a)
a-=b
b-=a
print("The value with subtraction assignment are", a ,"and", b)
c*=d
print("The value for multiplication assignment is", c)
b/=d
print("The value for division assignment is", b)
a%=b
print("The value for remainder assignment is", a)
d**=c
print("The value for exponent assignment is", d)
b//=d
print("The value for floor division assignment is", b)
```

The results for the code is:

```
The value with addition assignment is  33
The value with subtraction assignment are 10 and 13
The value for multiplication assignment is 10
The value for division assignment is 6.5
The value for remainder assignment is 3.5
The value for exponent assignment is 1024
The value for floor division assignment is 0.0
```

Here, the value of 'a' initially has been 10 but after the additional assignment, the new value of 'a' includes the addition of 'a' and 'b'. Therefore it is now 33. So, when applying subtraction assignment, the value of 'a' again is modified by subtracting 'b' from 'a', making the value of 'a' 10 while that of 'b' becomes13 due to te subtraction. With the usage of assignment operators, the value of the left operand gets modified.

Lastly, the comparison operators help in value comparison and return the value in the form of Boolean i.e. True or False. The list of comparison operators for the numeric values is defined in the below table.

Name	**Symbol**	**Description**
Equal	=	In case the value of the two operands is equal, then the value becomes True
Not equal	!=	With two operands value not equal, the value is True
Less than or equal to	<=	In case 1st operand is smaller than or equal to the 2nd operand, the value is True

Greater than or equal to	>=	With the 1st operand value more than or equal to the 2nd operand, the value is True
Greater than	>	Having the 1st operand value more than the 2nd operand, the value of the operator is True
Less than	<	If the value of the 1st operand is smaller than the 2nd operand, the operator value is True

With the application of the comparison operators, the code is designed with the inclusion of 4 variables; a,b, c, and d. The code for the comparison is

```
a = 10
b = 23
c = 23
d = 2
print(a==b)
print(a!=b)
print(a>=b)
print(c<=b)
print(c<a)
print(a>d)
```

The output for the same code is:

```
False
True
False
True
False
True
```

The results show that 'a' value is 10 while 'b' is 23, and both are not equal. Therefore the value of *equal to* operator is False while of *not equal to* operator is True. Similarly, for all other operators, the comparison could be drawn for the values.

Apart from the main operators, the numeric operations also include the usage of functions. A function is defined as the block of code which is used for performing specific tasks and runs only when it's called. The popular mathematical functions i.e. exponential and logarithm are called from the Python *math* module. The module is the file which consists of Python statements and definitions. The modules could be used in Python for defining variables, classes, or functions. The functions used for the working include

- exp*(a)* – for returning the value of e raised to power a i.e. e^a
- *log(a,b)* – for computing the logarithm value of a with base b. In case the base is not mentioned for a function, then the value derived is of natural log.

- *log2(a)* – the function derives the value of log a having a base of 2.
- *log10(a)* – the function compute log a value with a base of 10
- *pow(a,b)* – the function is used for deriving a raised to b power value i.e. a^b
- *sqrt()* – the function helps in computing the number of square root

Therefore, the numeric operators while designing python code enable the arithmetic operations and comparisons.

1.2. String

The string operators are the ones which could be used for working with string variable type. The string is part of an ordered objects set called sequence. Consisting of a sequence of characters, the string in Python is represented using single, double or triple quotes.

```
Subject = 'Programming with Python'
name = "Rahul"
education = '''Post Graduate'''
print(Subject, name, education)
```

3 variables are created in the above code using single, double and triple quotes i.e. subject, name, and education which store string value. The assessment of characters in string could be done in three forms i.e.

1. Indexing – One method is treating the string as the list. Herein, index values are used for accessing characters. The concept will be discussed in detail in the next chapter.
2. Negative Indexing – The method is similar to a list wherein using index values only characters are assessed but the method starts index value from the ending point of the string i.e. assessment is backwards.
3. Slicing – This method focuses on accessing the characters range using the slicing operator i.e. colon. Slicing simply means dividing the given text into small parts.

Using the 1st 2 variables as defined in the previous example i.e. Subject and name, the 3 forms of string assessment are applied. The code for the same is

```
Subject = 'Programming with Python'
name = "Rahul"
print(Subject[4])
print(Subject[-4])
print(name[2:4])
```

Herein, the indexing is used for deriving the 4th index character, negative indexing for deriving the -4 index character and slicing for deriving 2:4 index value range characters. The output for

the code is r, t, and hu. The index value always starts from 0 so the 4th index character in a Subject variable is r while negative indexing starts with -1 so the value is t.

There are many operations which can be performed with the usage of strings. These are

Name	**Symbol**	**Description**
Equal operator	==	The operator is used for comparing two strings. If equal the result is True else False
Unequal operator	!=	The operator returns the value True if not equal else False
Concatenation	+	The operator enables the joining 2 or more strings
Assignment	=	The operator is used for assigning the value
Repetition	*	The operator enables n times repeating of the string using string *n
Slicing	[]	The operator helps in the assessment of the string for a specific index. The positive index starts from 0 and the left side while the negative index from -1 and from the right side. Herein, the command[a:b] slices characters from a index to b index while [a:] slices characters from a index to the end of the string. [:-b] is used for returning characters from the start of the string to the -b index.
Reversing	[::-1]	The operator returns the string in reverse order
Membership	in and not in	The operator used for searching whether some specified character is present in a given input string or not
Escape	\	The operator eliminates a particular character assessment. Double quotes are used with the escape sequence operator
Formatting	%	The operator helps in formatting the string as per preference. % used as a prefix for defining what type of value needs to be inserted. %d for signed decimal integer, %u for unsigned decimal integer, %c for character, %s for string, and %f for floating point real integer

For defining in usage of each of the mentioned operators the code is developed.

```
name = "Rahul"
surname = "Jain"
eliminate = "Hello I am learning\"Pyton\""
age = 25
course = "B.A."
percentile = 8.12
print(name ==surname)
print(name!=surname)
print(name)
print(name + surname)
print(name*5)
print(name[2],name[-2], name[1:3],name[1:],name[:2],name[:-1])
print(name[::-1])
print("a" in surname)
print(eliminate)
print("My name is %s, and my age is %d" % (name, age))
print("My course is %s, and my score is %f" % (course, percentile))
```

The output for the code is

```
False
True
Rahul
RahulJain
RahulRahulRahulRahulRahul
h u ah ahul Ra Rahu
luhaR
True
Hello I am learning"Pyton"
My name is Rahul, and my age is 25
My course is B.A., and my score is 8.120000
```

The output shows that as name and surname variable values are not equal thus the result is False while for not equal to the operator it is True. For the slicing operator as per the relevant indexing, the values are derived. And with the reverse operator, the name variable is reversed. Lastly, the formatting operator helped in inserting different types of data while printing the string. Therefore string operators could be used for working with string variables. Some other working with string include the usage of different operators which simplifies the coding procedure. The popular methods used for string operations are:

- len() – to measure the length of the string
- upper() – for converting string into uppercase
- lower() – for converting the string to lowercase
- partition() – to return the tuple
- replace() – to replace the inside substring
- find() – to return the substring's first occurrence index

- rstrip() – to remove the trailing characters from the string
- split() – to split the string from left
- index() – to define the string index
- isnumeric() – to check the numeric characters
- startswith() – to check if a particular string starts with a specified string

1.3. Logical operations

Logical operators are the ones used for combining conditional statements. The operations help in returning boolean values i.e. True or False. The popular logical operators are

Name	Symbol	Description
Conjunction	and	The operator helps in knowing whether both statements given on the left or right side of the operator are true or not. If both statements are True then return True else if any of the operands or both are false then the result is False
Disjunction	or	The operand ensures at least one of the conditions is true. Thus, only if both the condition i.e. on the right and left of the operator is False then the result is False else True
Negation	not	The operator is used for reversing the result. If the result of the *and* operator is True then usage of *not* operator with *and* operator makes the value False. Similarly *not* reverse the value for *or* operator.

Using each of the defined operators, the coding is done.

```
print(4>2 and 4== 1+3)
print(4>2 and 4 == 1+2)
print(4>2 or 4 == 1+2)
print(4<2 or 4 == 1+2)
print(not(4>2 and 4== 1+3))
print(not(4<2 or 4 == 1+2))
```

The output for the code is

```
True
False
True
False
False
True
```

The above result shows that for the first statement, both values are *True,* making the result *True* but with the econd statement, one is *not True* leading to a *False* result. Similarly, for the Disjunction operator, the third statement result is *True* while the fourth statement value is *False*. In the fifth statement Negation operator reverses the output, making the result *False*.

In addition to the given conditional statements, the operators which could be used for comparison are *if*, *else* and *elif*. The processing for each of these operators is such that:

- In the case of the *if* operator, if the condition is True then code inside if is executed otherwise the processing is skipped.
- In the *if*-else operator, if the condition is True then the code inside if is executed otherwise the code under else is executed.
- For the *elif* operator, the choice is made between multiple alternatives i.e. if condition 1 is true then code under it is executed otherwise condition 2 is checked and in case of condition 2 being true code under it is executed. If condition 2 is false then else command further is checked.

The working of conditional statements could be stated as:

```
a = 5
b = 6
c = 10
if a > b:
    print(a,b)
if a > b:
    b = a
else:
    a = b
print(a,b)
if (a > b and a > c):
    print("a is greatest")
elif (b > a and b > c):
    print("b is greatest")
elif (c > a and c > b):
    print("c is greatest")
else:
    print("Can't say anything")
```

The output for the above code is:

```
6 6
c is greatest
```

Here, code under the first if statement is skipped as the value of 'a' is not greater than 'b'. Now, for the second statement as the value of 'a' is not more than 'b' thus else statement condition is executed resulting in changing the value of 'a' to 6 and making output 6 6. Finally, as condition c> a and c>b is True making output as 'c' is greatest.

Lastly, the logical operators including the looping statement under which using statements are executed sequentially. There are majorly two types of loops i.e. *for* and *while* loop. The inclusion of multiple for and while loops is known as a nested loop.

- *For* loop is used for iterating through objects like strings, lists, or tuples. The loop has an index starting from 0.
- *While* loop enables execution of statements block repeatedly until the condition becomes False. The loop is similar to if but herein the loop executes and continues to run until the condition becomes false. Therefore, the while loop could be infinite if not modifying condition is included.

The case of the usage of the *for* and *while* loop is given below:

```
b = 0
name = "Rahul"
for a in name:
    print(a)
while b < 10:
    print(b)
    b = b + 1
```

The results of the code is:

```
R
a
h
u
l
0
1
2
3
4
5
6
7
8
9
```

As with the *for* loop, the iteration is performed. Therefore, each character of the string is printed. Further, while loop is performed which continued till the value of b < 10. Herein, the statement for changing the 'b' value was included i.e. b = b+1. In case the statement is not included, the while loop becomes infinite. The results for the case will change with nested loop use i.e.

```
b = 0
name = "Rahul"
for a in name:
    print(a)
    while b < 10:
        print(b)
        b = b + 1
```

The output for the case will be:

```
R
0
1
2
3
4
5
6
7
8
9
a
h
u
l
```

Herein, with first iteration of *for* loop R is printed, then *while* loop is performed. Now, with the second iteration of the *for* loop the value of 'b' is already 9, therefore *while* loop is not applicable. The loop includes sub-loops therefore it is known as a nested loop.

2. Summary

The symbols used in Python for describing the computation are referred to as the 'operator' while the value derived is known as 'operand'.

There are three types of operations i.e. numeric, string and logical.

The operations performed with numeric values are numeric operations. It includes assignment, arithmetic and comparison operators like equal to, not equal to, addition, or subtraction.

String operations are used for working with string variables. It consists of operators like concatenation, assignment, slicing, repetition, or escape.

Logical operations help in dealing with conditional statements like conjunction, disjunction, negation, if, else-if, while or for loop.

Chapter: 5 Lists, dictionaries, tuples & sets

Objectives

To discuss the data structures of python

- Explore the Python lists by discussing its concept, application, modification, and working
- Examine the dictionaries by discussing its features, applications, and working
- Describe tuples and its working
- Using sets for performing different operations

0. Introduction

Python consists of different in-built data structures like lists, tuples, dictionaries, and sets which help in organizing and storing the data efficiently. For learning programming, knowledge of existing data structures for creating the code is essential. This chapter focuses on examining lists, tuples, dictionaries, and sets along with exploring their working with Python.

1. Lists

Python lists are dynamic-sized arrays which can be declared in other languages like C++ or Java. It is simply the means of storing multiple items in a single variable. The list is enclosed by square brackets [] and the items which are included in the variable are separated by comma. The items stored in the list are known as list elements. Some of the essential features of the list are

- As the elements of the list could be modified, the list is said to be mutable.
- The order in which elements are present in the list is defined making the list as ordered.
- The list has the feature of supporting duplicate values inclusion, providing the option of storing the same element at different indexes.
- The values of the list could be accessed using the index value.

In Python, a list could be created in two ways as follows.

1. Using the list() function.
2. By using square brackets and either keeping the brackets as blank or adding elements in it.

The code for list declaration could be stated as:

```
a = list()
b = []
c = [1, 2, 3]
print(a,b,c)
```

The output for the code is:

```
[] [] [1, 2, 3]
```

Here, ‘a’ and ‘b’ are empty lists created using the list() function or square brackets while ‘c’ is a list created with numeric elements.

There are no defined data types which need to be stored in the list. The data structure supports the usage of numeric, string, or Boolean data types and even could contain a mix of data types. The list is data structure type only, therefore using the type() function, the class derived for the list data structure is *class list.*

The index is the means of accessing elements of the list and by specifying the new value for any particular index in the list, the values of the list could be updated. Suppose a list is defined by storing 4 students' names and printing them. Now, if one of the names is stated wrongly and changed back to the correct one, then the new list is generated. The code for this case is stated below:

```
names = ['Neha', 'Sneha', 'Sachin', 'Rahul']
print(names[1])
names[1] = "Sakshi"
print(names)
```

The output for the code is:

```
Sneha
['Neha', 'Sakshi', 'Sachin', 'Rahul']
```

With list indexing is processed the same way as for string, therefore, list elements could be accessed using the slice operator i.e. []. The indexing for the list starts from 0 and goes to length -1. For the same case as above, the names from index value to 2 to ending value are generated.

```
names = ['Neha', 'Sneha', 'Sachin', 'Rahul']
print(names[1])
names[1] = "Sakshi"
print(names)
print(names[2:])
```

The result of the code is:

```
Sneha
['Neha', 'Sakshi', 'Sachin', 'Rahul']
['Sachin', 'Rahul']
```

For the Python lists, the values could be added using the append and insert function. The append function enables adding the value to the list end while insert enables adding the value at the point mentioned by the developer. Suppose a name is added using the append function and another by the insert function, the code could be mentioned as:

```
names = ['Neha', 'Sneha', 'Sachin', 'Rahul']
names.append("Sakshi")
print(names)
names.insert(2,"Samkit")
print(names)
```

The result of the code is:

```
['Neha', 'Sneha', 'Sachin', 'Rahul', 'Sakshi']
['Neha', 'Sneha', 'Samkit', 'Sachin', 'Rahul', 'Sakshi']
```

In case of removing, the functions, 'pop', 'remove' and 'delete' could be used. The 'pop' function deletes the last element of the list, 'remove' eliminates first occurred element from the list, and 'delete' enables complete deletion of the element for the given index of the list. The code for the same is:

```
names = ['Neha', 'Sneha', 'Sachin', 'Sneha','Rahul', 'Sneha', 'Anisha']
names.pop()
print(names)
names.remove('Sneha')
print(names)
del names[2:5]
print(names)
```

The output of the code is:

```
['Neha', 'Sneha', 'Sachin', 'Sneha', 'Rahul', 'Sneha']
['Neha', 'Sachin', 'Sneha', 'Rahul', 'Sneha']
['Neha', 'Sachin']
```

Apart from the different specified functions, the operations could also be used with lists for working. The operations that are supported with lists are concatenation, repetition, membership, and length. The code for the implementation of operations is

```
price = [10, 11, 24]
newprice = [12, 20]
Totalprice = price + newprice
print("list repetition is", price*2)
print("Concatenated prices are", Totalprice)
print("Length of total prices is ", len(Totalprice))
print(20 in Totalprice)
```

The output of the code shows that a new list consists of 2 times repetition for 'price' list, 'totalprice' is derived by concatenating 'price' and 'newprice' list. Price has 3 elements and 'newprice' has 2 elements thus length of 'Totalprice' is 5 and as 20 is present in 'Totalprice' therefore the result is *True*.

```
list repetition is [10, 11, 24, 10, 11, 24]
Concatenated prices are [10, 11, 24, 12, 20]
Length of total prices is  5
True
```

Further, the list also supports the iteration function wherein using *for* and *in*-loop command, the values could be derived.

```
price = [10, 11, 24]
for p in price:
    print(p)
```

In the above code, 'p' holds each element of list price and the loop enables iteration through each element of price. In a similar way while loop could also be used for accessing every element of the list. Apart from these functionalities, the list also consists of certain methods which are most commonly used for supporting the coding.

- list.extend(list2) – the method is used for adding the elements of list2 to the list end
- list.index(elem) – for searching the index value of the given element
- list.sort() – the method is used for sorting the list in place
- list. reverse() – the method enables reversing the list in place

2. Dictionaries

The dictionary in Python could be defined as the data values ordered collection which is used for data values storing in the form of a map. Unlike the other data structures wherein an element has only one value, the elements of a dictionary consist of two parts i.e. *key: values*. The *key: value* pair allocate the key to the specific values. Colon is used to separate the key from values wherein the left side of the colon is the key while the right side is the value. In the dictionary, the elements are stored using curly brackets i.e. {} and a command is used for separating the different elements. Some of the main features of the dictionary are as follows.

- The index values define the position of elements making position a relevant aspect. Thus, the dictionary is ordered.
- The elements of the dictionary could be changed leading to mutable feature.
- No repeated values could be present in the dictionary.

In Python, the dictionary can be created in 2 ways:

1. Using the built-in function dict() – the function creates an empty dictionary to which values could be assigned
2. Curly brackets for defining the dictionary either by keeping empty or by adding elements.

The case for the creation of a dictionary is stated below.

```
a = {}
b = dict()
c = {"age":25,"name":"Sachin"}
print(a,b,c)
```

The output for the code is:

```
{} {} {'age': 25, 'name': 'Sachin'}
```

Here, two empty dictionaries using built-in functions and curly brackets are created along with another dictionary named 'c' having 2 elements i.e. *age* and *name*. There is no restriction on the nature of data that could be stored as an element in a dictionary. It could be string, numeric, Boolean or a mix of all. When studying the type of the created dictionary using the type() function, the output derived is *class dict* which confirms that the variable created for storing elements is of dictionary data structure. The length of a dictionary could be derived using the len() function. For access to the elements of the dictionary, the key could be used inside the square brackets or the built-in function *get.*

```
a = {}
b = dict()
c = {"age":25,"name":"Sachin"}
print(c["age"], c.get("age"))
```

The above code verifies that using key value or get, the output derived would be 25 only. Further, the dictionaries have methods for accessing all the keys or the values i.e. keys() or values(). The method helps in creating a list with storage of all keys or values. Even the dictionary is mutable, thus, using the key, the value specified for that particular key could be changed. The dictionary has one more method for accessing all elements as separate tuples. The method is items().

```
a = {}
b = dict()
c = {"age":25,"name":"Sachin"}
print(c.keys())
print(c.values())
c["age"] = 30
print(c.values())
print(c.items())
```

For the above code, the derived output is

```
dict_keys(['age', 'name'])
dict_values([25, 'Sachin'])
dict_values([30, 'Sachin'])
dict_items([('age', 30), ('name', 'Sachin')])
```

The output shows initially all the keys and values. Further, as the value of the age key is changed, the updated values could also be seen in the output. Lastly, the list of tuples is created for each element of the dictionary.

The modification of the dictionary is not only done by changing value but also using the update() method. Herein, not just the value but the key also needs to be specified i.e.

```
a = {}
b = dict()
c = {"age":25,"name":"Sachin"}
print(c.keys())
print(c.values())
c["age"] = 30
print(c.values())
c.update({"age":35})
print(c)
```

The updated dictionary has changed the value of *age* i.e. 35

```
dict_keys(['age', 'name'])
dict_values([25, 'Sachin'])
dict_values([30, 'Sachin'])
{'age': 35, 'name': 'Sachin'}
```

Modifications in the dictionary can also be done in the form of addition or removal. By defining a new index and its value, the new element could be added to the dictionary. Herein too update could be used as the method checks in the dictionary and if the key does not exist then it will add the key as a new element. For the removal of the element methods like *pop, popitem, del,* and *clear* could be used. Pop() removes the element with a specific key, popitem() removes the last inserted item, *del* keyword again removes the element with a specified key or the complete dictionary and *clear* is used to empty the dictionary.

```
c = {"age":25,"name":"Sachin"}
a = {"score":58}
b = {"total":100}
c["course"] = "statistics"
print(c)
c.update({"level":"graduation"})
print(c)
c.pop("name")
print(c)
c.popitem()
print(c)
a.clear()
print(a)
del c["age"]
print(c)
del b
```

Herein, the output is

```
{'age': 25, 'name': 'Sachin', 'course': 'statistics'}
{'age': 25, 'name': 'Sachin', 'course': 'statistics', 'level': 'graduation'}
{'age': 25, 'course': 'statistics', 'level': 'graduation'}
{'age': 25, 'course': 'statistics'}
{}
{'course': 'statistics'}
```

The dictionary in 1st output has 1 more element with the key *course*, similarly, the *level* key element is also added. Further, using *pop* name key is removed and using *popitem* last inserted value so the *level* is removed. Now, using *clear* the dictionary a is derived as an empty dictionary while eliminating *age* from *c*, the updated dictionary has only the *course* key left. Finally, the *b* dictionary is deleted.

Looping through the dictionary could be done using a *for* loop and an *if-in* command could be used to check whether a key exists in the dictionary or not.

```
c = {"age":25,"name":"Sachin"}
for a in c:
    print(a)
if "age" in c:
    print(c["age"])
```

Herein, the *for* loop print key while the *if-in* command checks if the *age* key exists and if it does, then its value is printed.

```
age
name
25
```

Apart from the loopings and the modifications, there is the existence of methods which can be used in the dictionary for functioning.

- Copy() – for returning dictionary copy
- Fromkeys() – returning a dictionary with specified keys and values
- Setdefault() – to return the value of the specified key. In case the key does not exist, the key is added to the mentioned value

3. Tuples

The Python tuples are defined as the Python objects collection which are separated by commas. Tuples in many ways are similar to lists like with their functioning of nested objects, indexing, or repetition but as the elements of a tuple can't be altered therefore tuple is different from a list. The tuple is created using parenthesis i.e. () wherein each element is separated via a comma. Some of the important features of tuples are

- The order of elements is of relevance making tuples ordered
- The elements of tuples cannot be changed leading to have immutable form
- The repeated value could be added as an element in the tuple

The tuple is created by using tuple(), empty parenthesis, having a single element followed by a comma, or with multiple elements. Herein when assessing the type of tuple, the result derived is a *class tuple*. The elements access procedure for the tuple is the same as that of the list wherein the index values help in accessing the values starting from 0 to -1. There is no defined data type of tuple. It could include numeric, string, boolean or mixed data types. The length of the tuple is determined using the len() function. The index value is presented within square brackets for accessing tuple items and the range of indexes helps in slicing the elements.

```
a = tuple()
b =()
c =(25,)
d = ("name", 30, "age", 56, 100)
print(a,b,c,d)
print(len(d))
print(d[0], d[:-1],d[0:2])
```

The output for the defined code is

```
() () (25,) ('name', 30, 'age', 56, 100)
5
name ('name', 30, 'age', 56) ('name', 30)
```

Herein, 2 empty tuples are created for *a* and *b* while 1 tuples with a single element. Lastly, *d* consists of a tuple having a length of 5. Using indexes and slicing the elements of the tuple could be accessed.

As a tuple is immutable thus the values can't be changed but herein, the tuple could be converted into a list and then changed and later on converted back again to a tuple.

```
d = ("name", 30, "age", 56, 100)
a = list(d)
a[1] = 35
d = tuple(a)
print(d)
```

In the above code, initially, at 1st index, the value in the tuple was 30 but to change the value, the *d* tuple is converted into list *a* and then the value at index 1 is changed. Once the value is changed, the list *a* is converted back to tuple *d*. Now, the changed tuple is derived.

```
('name', 35, 'age', 56, 100)
```

A similar process needs to be followed to remove the element from the tuple. Further, modifications in a tuple could be done by adding another tuple to the tuple using concatenation

```
d = ("name", 30, "age", 56, 100)
a = ("graduate",)
d+=a
print(d)
```

The below shown output represents the clubbed result i.e. the addition of tuple *a* in tuple *d*.

```
('name', 30, 'age', 56, 100, 'graduate')
```

Further, the operations which can be performed with a tuple are length, membership, and repetition.

```
d = ("name", 30, "age", 56, 100)
a = ("graduate",)
d= a+d
print(len(d))
print(d)
print (a*3)
print(30 in d)
```

The output for the above code is

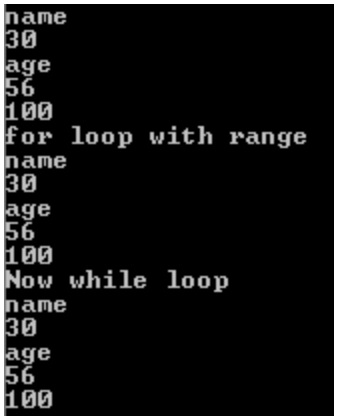
```
6
('graduate', 'name', 30, 'age', 56, 100)
('graduate', 'graduate', 'graduate')
True
```

Herein, the length of *d* is 6, the value of the *d* tuple is derived by concatenation, and repetition of a tuple is done 3 times. Finally, as the 30 value is present in tuple *d*, thus, the result is *True*.

Apart from modifications, with tuples, the looping could also be done using range and length functions. The *for* and *while* loop could be created. The code using the 2 loops is stated below

```
d = ("name", 30, "age", 56, 100)
for a in d:
    print(a)
print("for loop with range")
for j in range(len(d)):
    print(d[j])
print("Now while loop")
i = 0
while i < len(d):
    print(d[i])
    i = i+1
```

The result of the looping code is



Herein, in the *for* loop, the simple iteration is done to print all elements of the tuple. Following it is *for* loop with range. In this case, too, the iteration is done by determining the length of the *d* tuple i.e. 5 and creating its range [0,1,2,3,4]. Lastly, for the *while* loop, the processing is done until the value of *i* is less than the length of *d*.

Apart form these iterations, the methods which could be used with tuples are

- Count() – to derive the number of times a value occurs in tuple
- Index() – for searching a specified value and finding its position

4. Sets

A Python set could be defined as the data type collection which is mutable, iterable and consists of no duplicate values. The items of the set are stated using a curly bracket i.e. {} and separated using commas. The set class of python is similar to the mathematical set concept. The features of sets are

- The position of the elements in the set has no relevance making them unordered
- The element of the set could be changed but of frozen set is immutable
- There is no acceptance of duplicate values

Herein, the data stored with a set could be of any type i.e. numeric, string, Boolean or a mix. The method used for the creation of a set is set() or the curly brackets with values separated by a comma. The length for the set could be derived using the len() function. Herein, the values could not be accessed using an index or key. But using the for-in loop, it could be checked whether the value is present in the set or not.

```
a = set()
b = {10, "rahul", 25}
print(len(b))
for i in b:
    print(i)
```

Based on the above code, the output will be

```
3
25
10
rahul
```

Herein, the length of the *b* set is 3. Iterating through the set the values in the *b* set are derived.

The set does not support values change but the addition of the elements could be done using the add() method. Even the update() method could be used for adding items of another set into the current set. The item could also be removed using remove(), discard(), pop(), clear() and del method.

```
b = {10, "rahul", 25}
b.add("graduate")
print(b)
b.update({30})
print(b)
b.remove(10)
print(b)
b.discard(25)
print(b)
```

The above code shows that 2 values are added in set *b* i.e. graduate and 30 while 2 are removed i.e. 10 and 25. So, the output is

```
{25, 10, 'rahul', 'graduate'}
{'rahul', 'graduate', 25, 10, 30}
{'rahul', 'graduate', 25, 30}
{'rahul', 'graduate', 30}
```

Along with all these functions, the sets also include certain methods which could be used for processing i.e.

- Union() – for joining 2 or more sets
- Intersection_update() – to have an element which exists in both sets
- Intersection() – derivation of new set having all values present in both sets

- Symmetric_difference_update() – to keep elements which are not present in both sets
- Symmetric_difference() – to have a new set derivation for including elements which are not present in both sets
- Copy() – for copying sets
- Difference() – to return set having a difference between two sets
- Difference_update() – to compute the difference between two sets
- Isdisjoint() – to return whether two sets have an intersection or not
- Issubset() – to check whether a set is part of another set or not

Hence, using the stated functions and the modification, the code could be designed and work can be done with sets.

5. Summary

The list is enclosed by square brackets [] and the items which are included in the variable are separated by comma. It is mutable, ordered, has duplicate values, and could be assessed using an index

Dictionaries are data structures wherein elements are stored using curly brackets i.e. {}. The dictionaries are mutable, ordered, and could be assessed using the index, but do not contain duplicate values

Tuples are created using parenthesis i.e. () and are of form immutable, ordered, can be assessed using index, and have acceptance of duplicate values

Set are stated using curly bracket i.e. {} with no duplicate values inclusion and are unordered. It can be mutable but if the set is frozen then the set becomes immutable.

Chapter: 6 Executing code through Spyder Command Prompt

Objectives

To discuss the working with Spyder command prompt

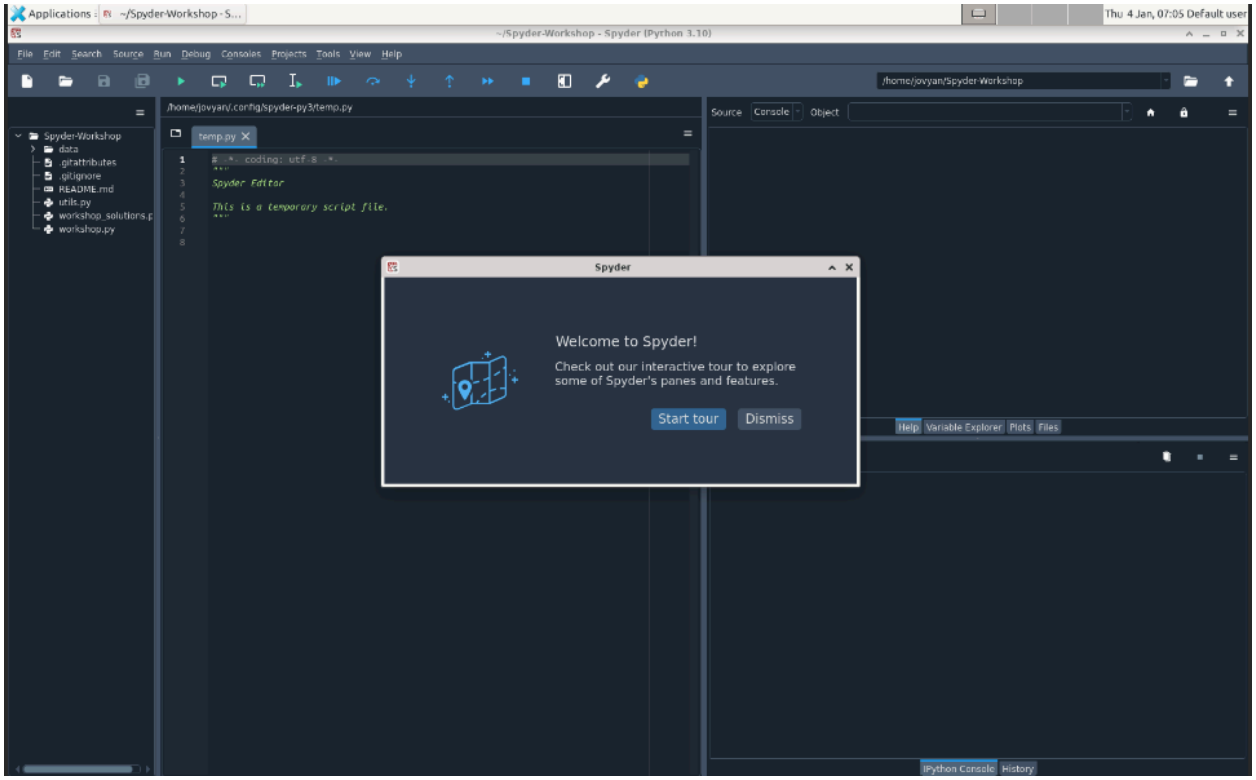
- Understand the procedure of setting up Spyder environment
- Examine the procedure of working with python files by discussing thier execution, arguments passing, or code debugging
- Describe the common problems while running files with Spyder command prompt and means of troubleshooting them

0. Introduction to Spyder Command Prompt

Scientific Python development environment, commonly known as Spyder, is the IDE which is part of Anaconda. It is simply an open-source and free scientific environment which is written for Python, in Python, and is designed for and by data analysts, engineers, and scientists. The environment provides features of introspection, debugging, interactive testing, and editing. Having a unique feature combination of profiling, debugging, analysis, and editing functionality, the environment serves as a comprehensive development tool with the inclusion of scientific packages supporting beautiful visualization capabilities, interactive execution, deep inspection, and data exploration. This chapter focuses on discussing working with the Spyder command prompt along with identifying the main problems in coding with the environment and the means of resolving them.

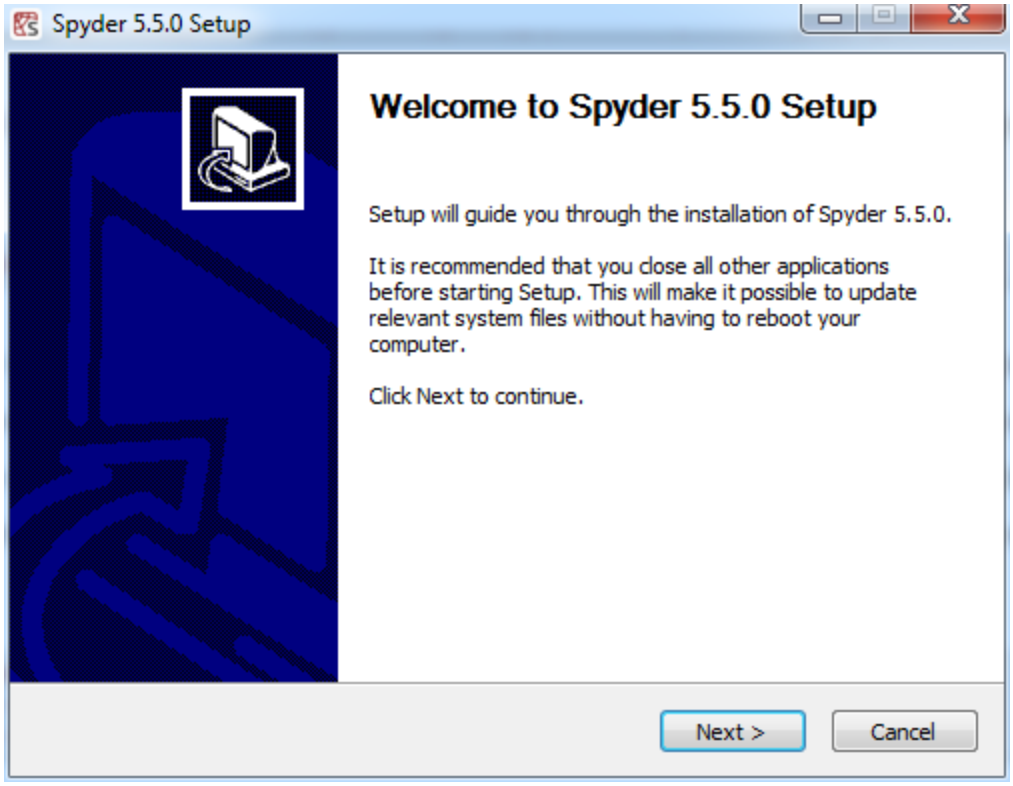
1. Setting Up Your Environment

The environment for Spyder could be set up in three forms i.e. having Spyder online, using standalone installers or using a virtual environment such as Anaconda. The Spyder has an online platform for working which is known as Syder Binder.



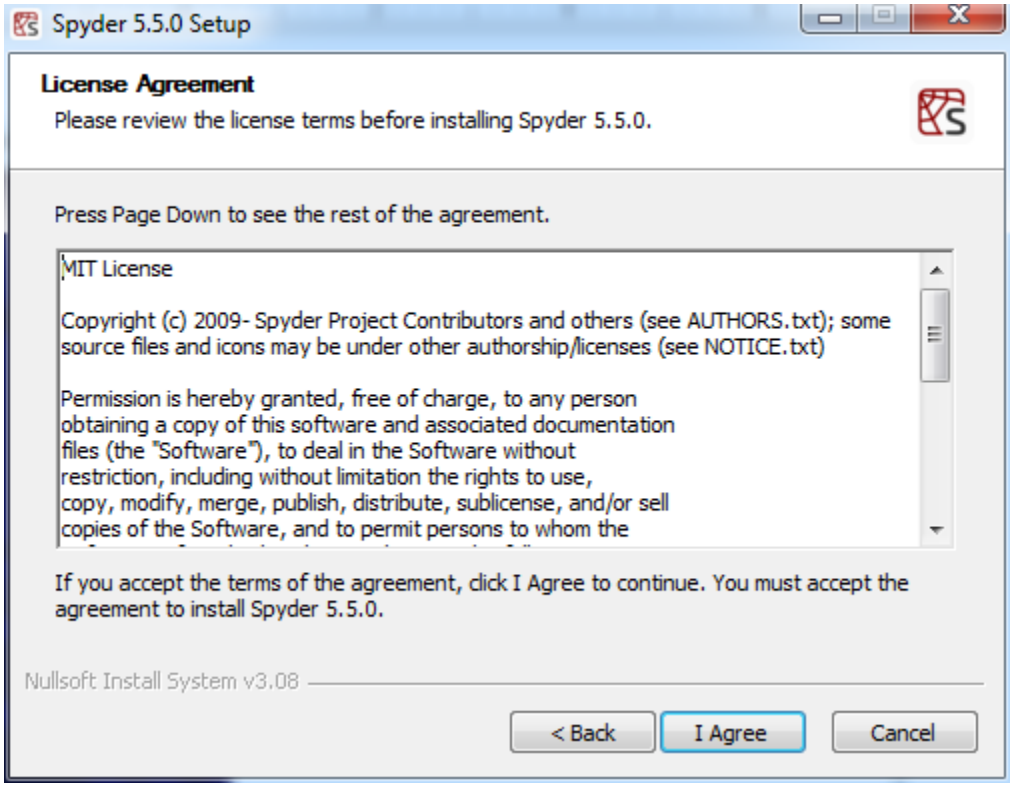
Despite this online platform, the most recommended method of setting up the environment is using standalone installers. Currently, the installers are available for macOS and Windows while for Linux the environment is under development. The installer has a built-in Python environment with scientific libraries like Matpotlib, Pandas, or NumPy. The procedure of installation is

1. Download the installer for Windows or MacOS. Version 5.5.0 is available currently.
2. Click on yes to install the Syder environment
3. The dialog box will appear to start the Spyder installation

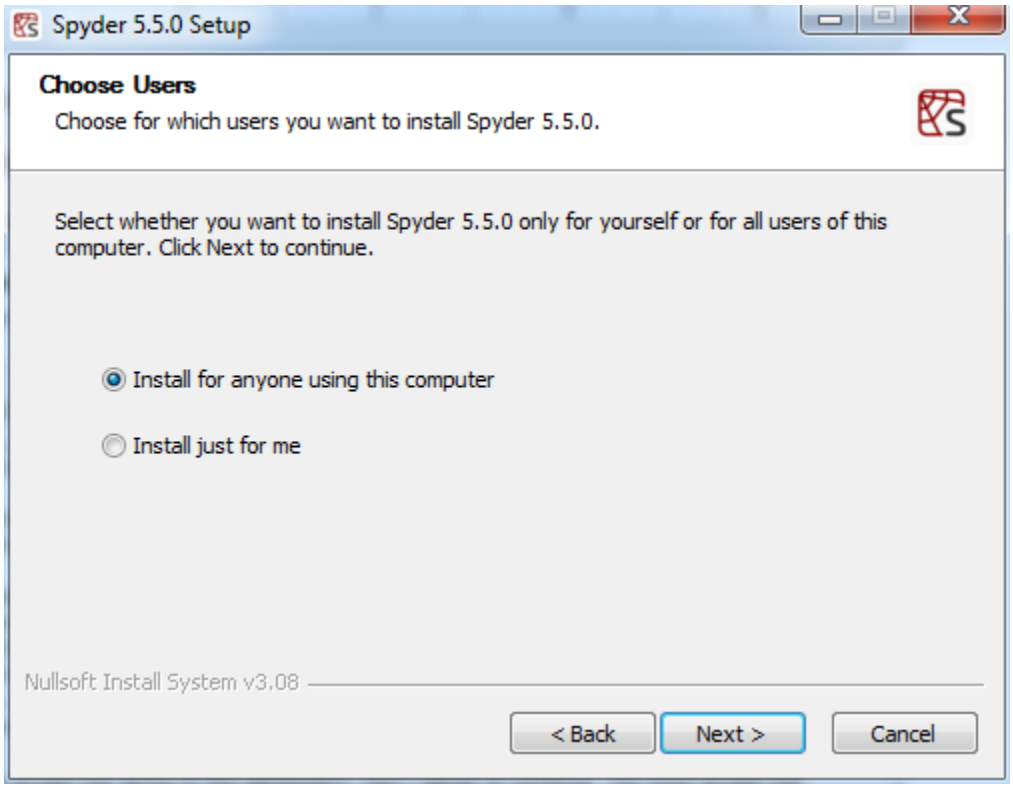


Click on next

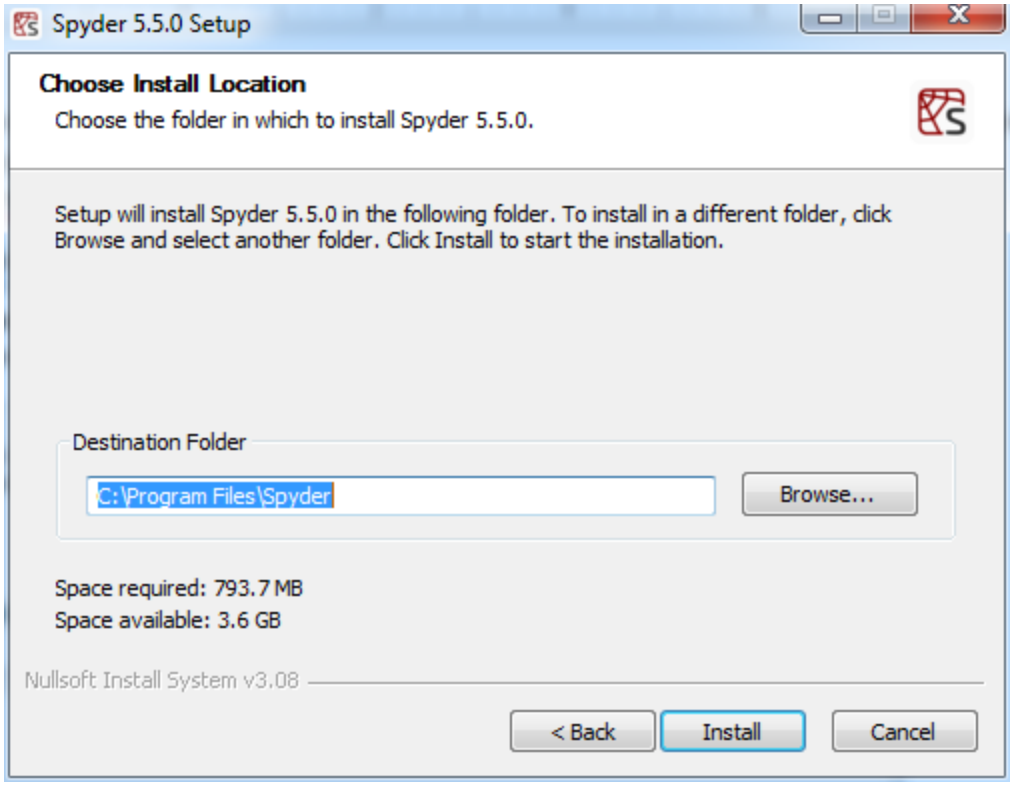
4. A license agreement window will appear. Read all license details and click on I agree



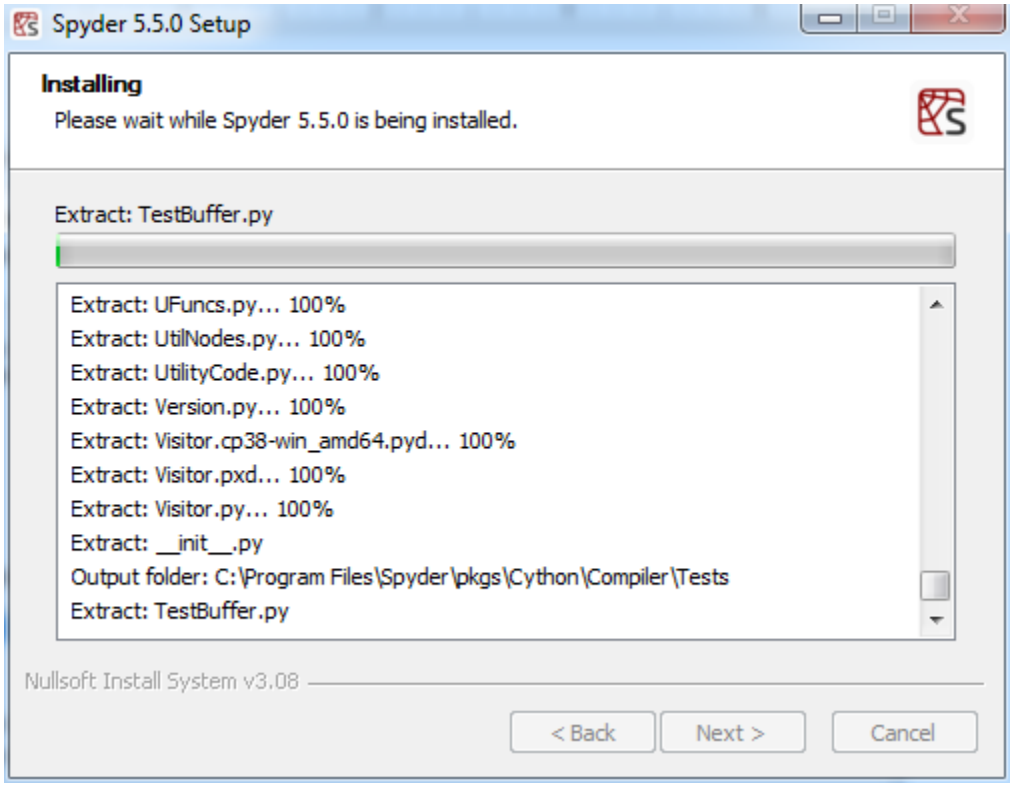
5. A window will appear to choose for whom the installation is done. It could be for a single person or anyone using the computer. So, choose accordingly. The default option is to install it for anyone using the computer. Click on Next after making a selection



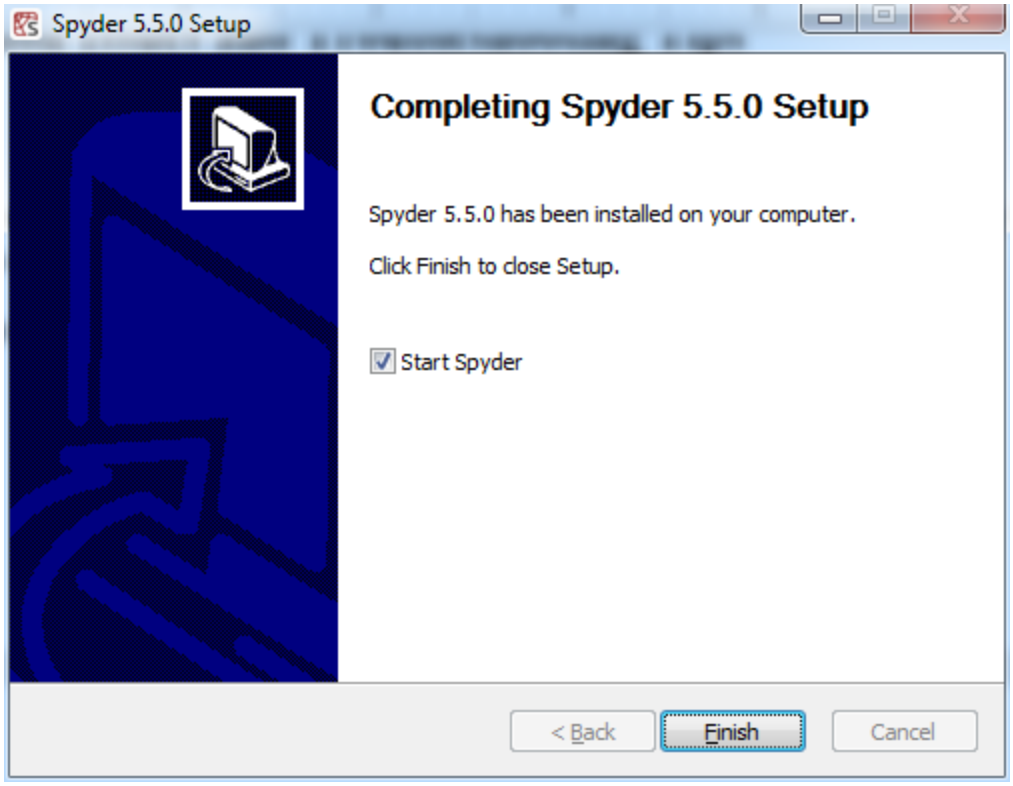
6. The window for choosing the installation location will appear. Select the desired location (like here in local disk sub-folder program files) and click on install



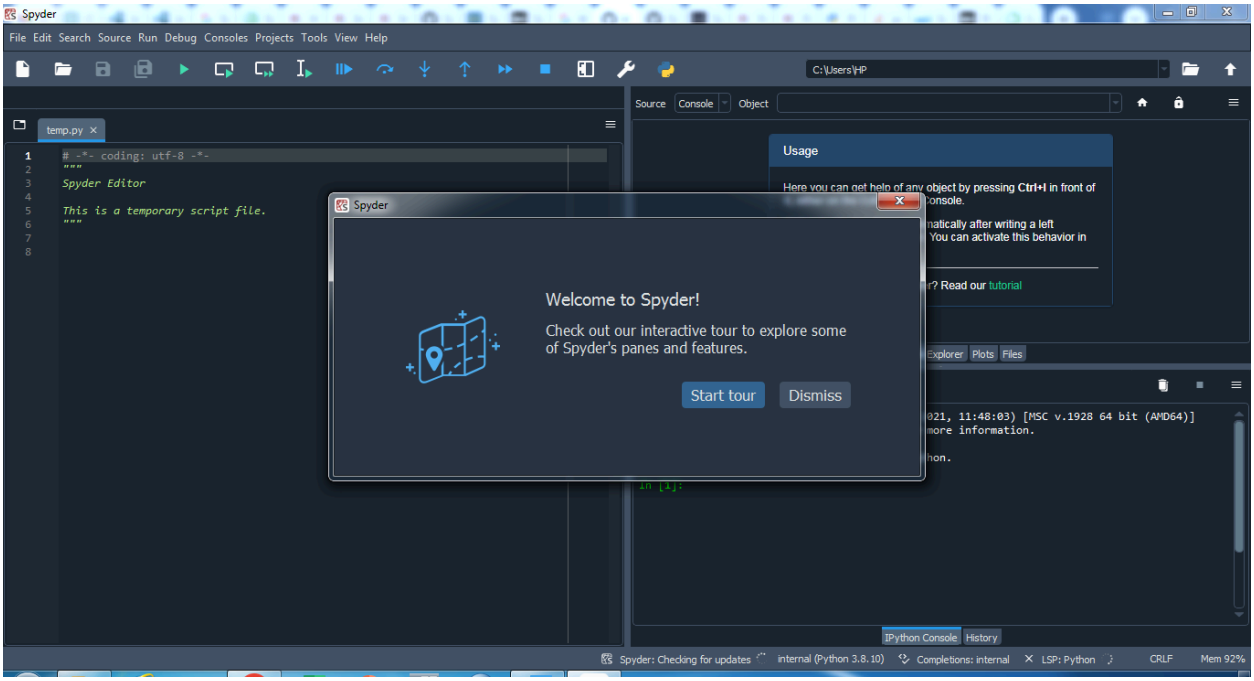
7. The installation procedure will begin



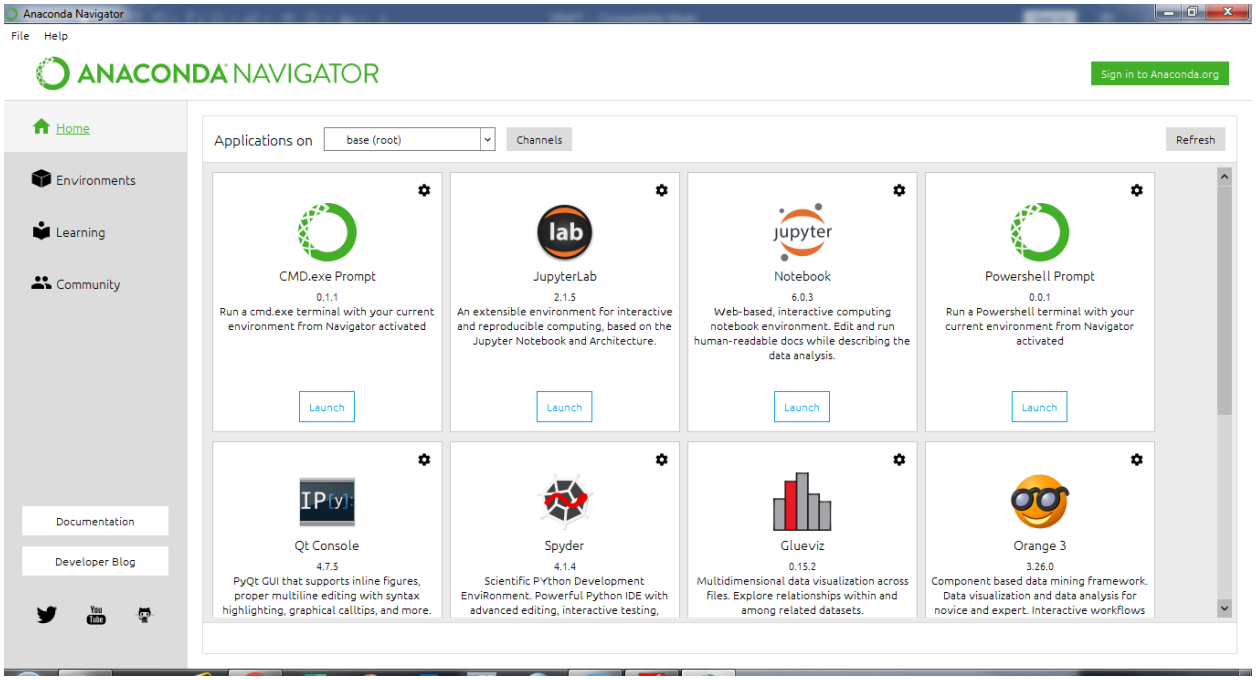
8. Once installation is complete, the window for closing setup will appear. Click on finish.



9. The following window will appear to show the complete setup of the Spyder environment



Another means is having conda-based use of Spyder environment. The virtual environment of Anaconda shows the presence of Spyder environment.



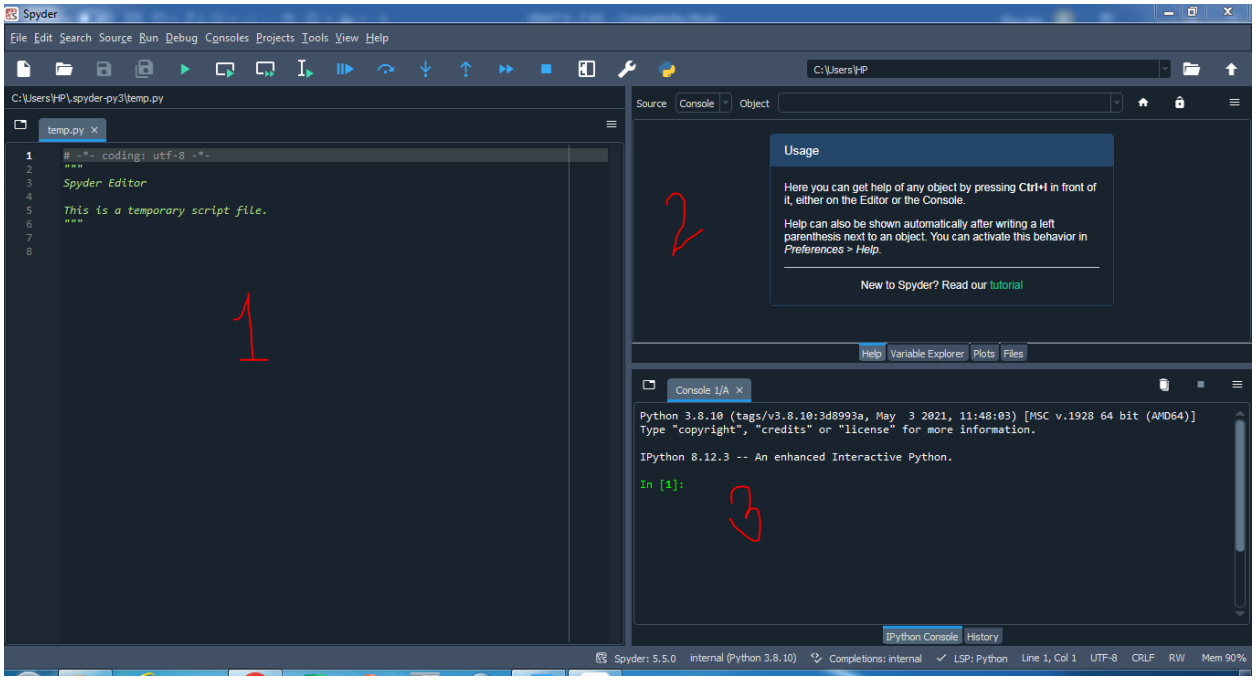
In the Anaconda navigator, simply scroll to Spyder and click on launch.

Following any of the three stated methods, the Spyder environment could be set up.

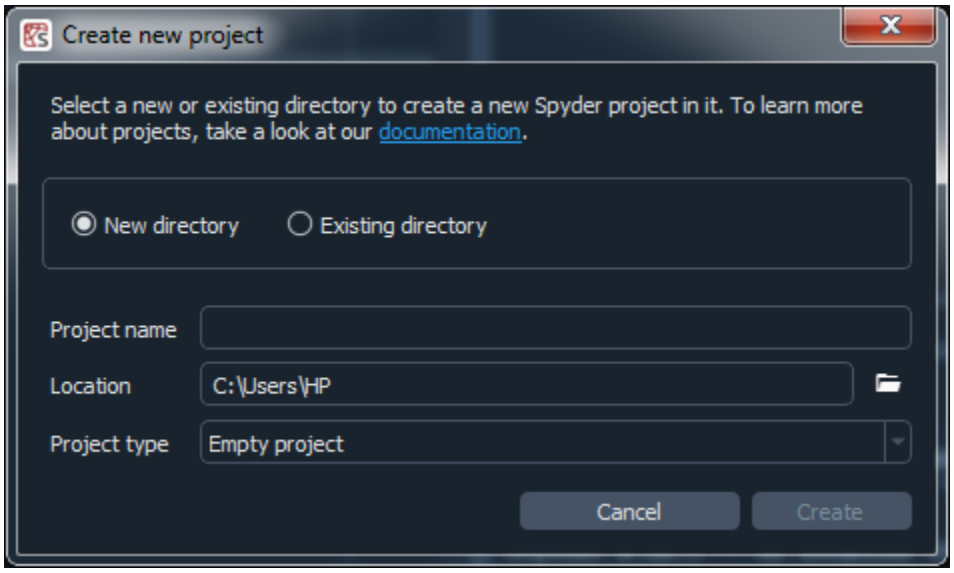
2. Using the Command Prompt to Run Python Code

The Spyder IDE has three main windows.

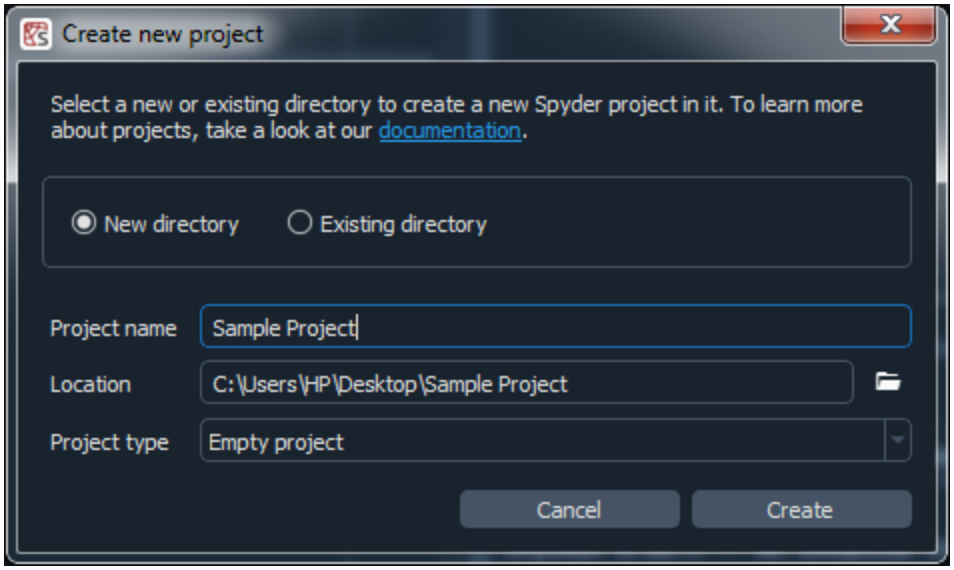
1. Editor window – the window used for writing Python scripts
2. Object inspector – to provide a place wherein browsing could be done through folders and information about functions, procedures, and modules which are used in the script. It consists of help, variable explorer, plots and files.
3. Console – the window for accessing the ipython shell and script results



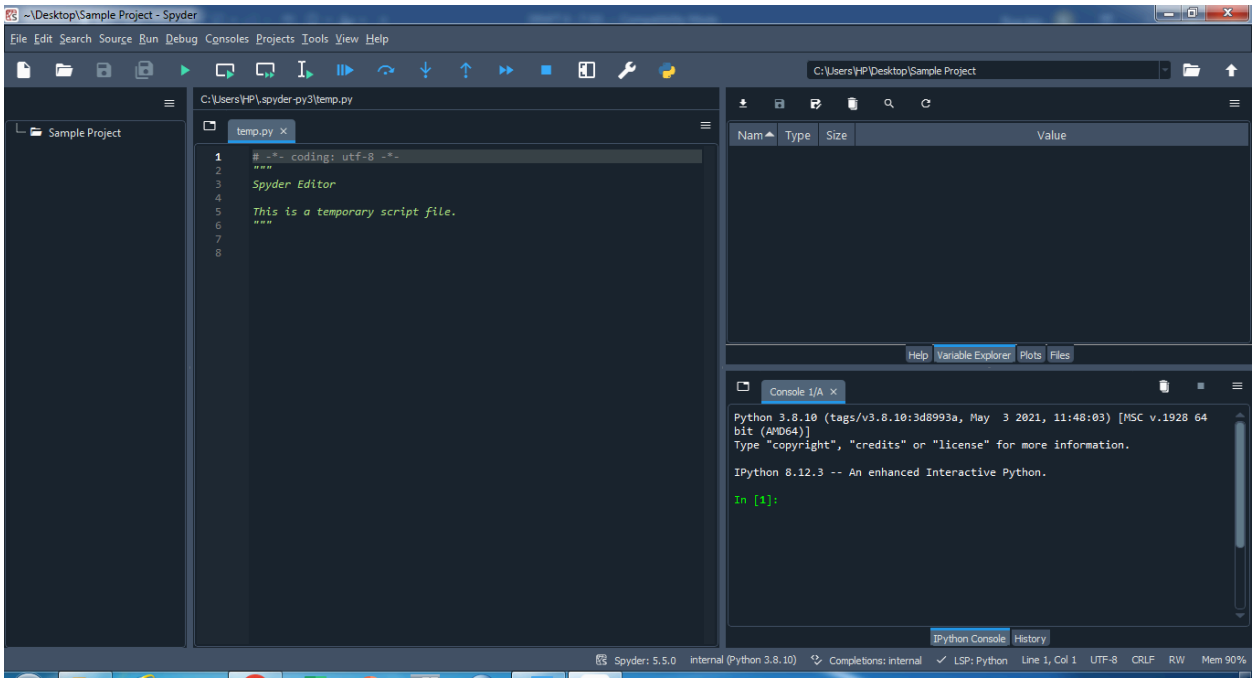
To start a new project in Spyder IDE, the need is to create a new project using projects >new project



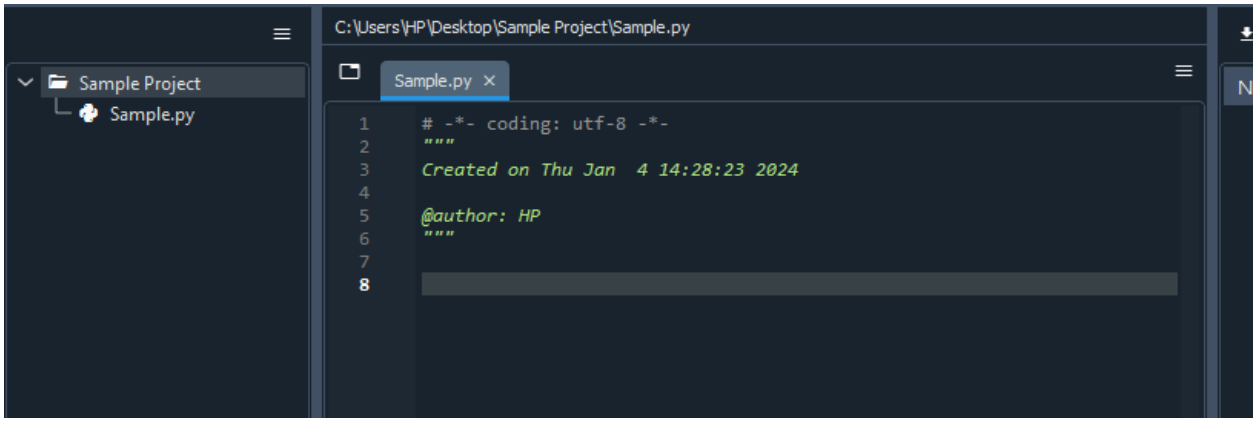
Fill in new project information like project name, location, and project type. Herein, we are choosing the name as *Sample project* and saving the project on the desktop.



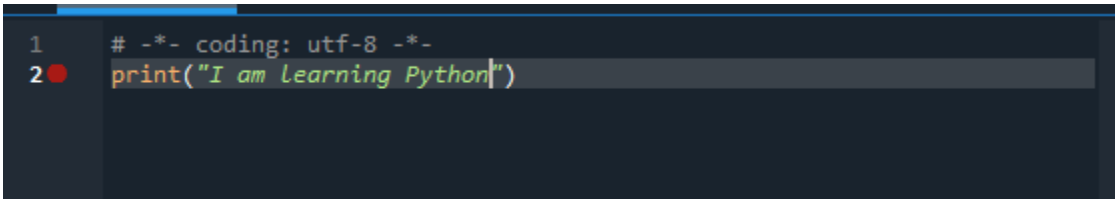
The project folder will appear in the Project Explorer pane



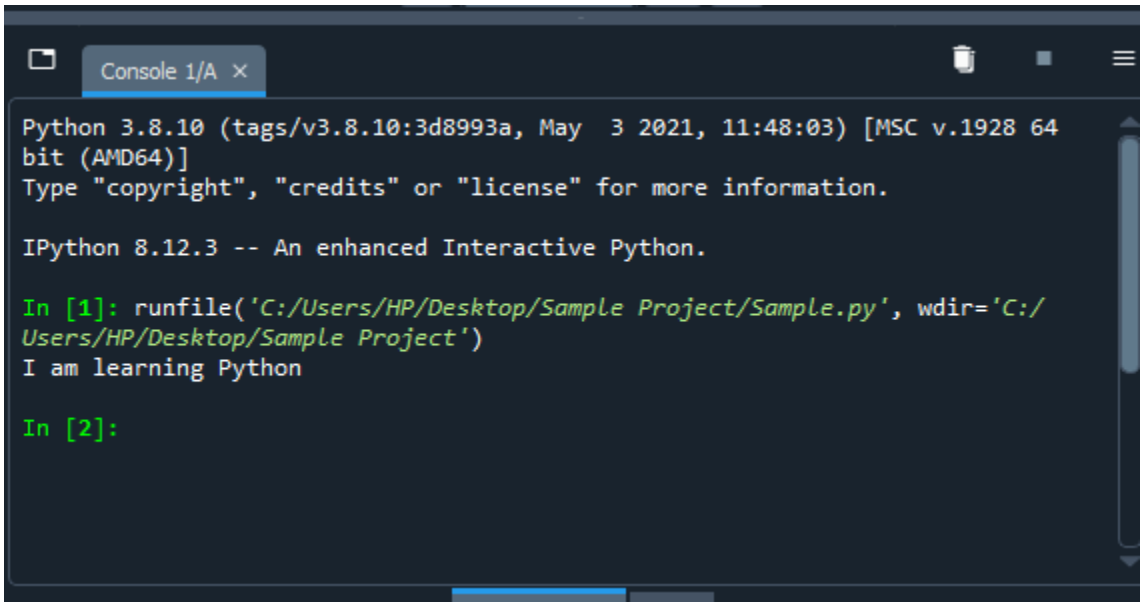
As every project has some files like text files, images, or code, therefore, we write the Python script by right-clicking on the sample project and selecting new>Python file



The file is saved under the project as a *sample*. The file again appears under the created project. Herein, we can write the code like



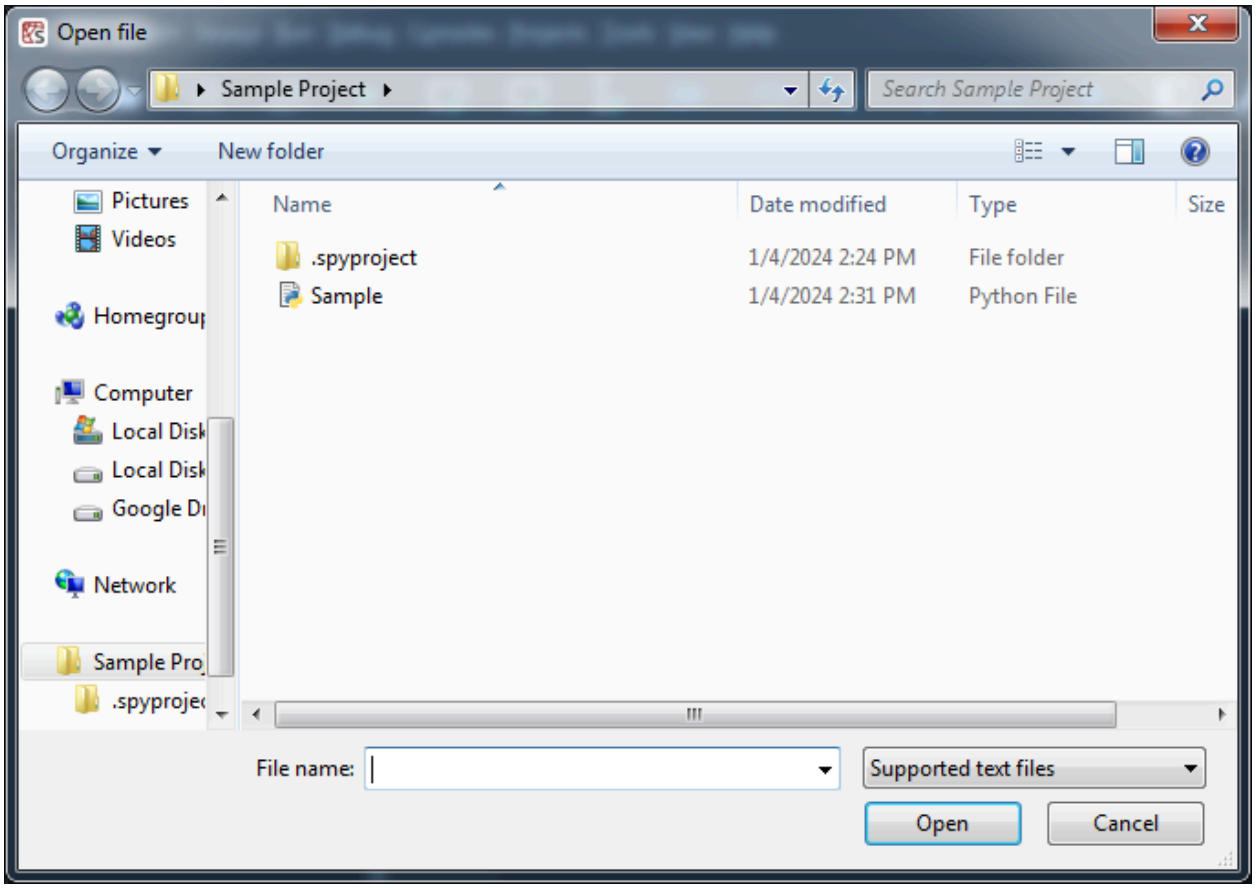
Save the file for processing by clicking on File>Save or using Ctrl S. Click on the  icon from the main menu or use key F5 to run the code. The result is derived in the console window.



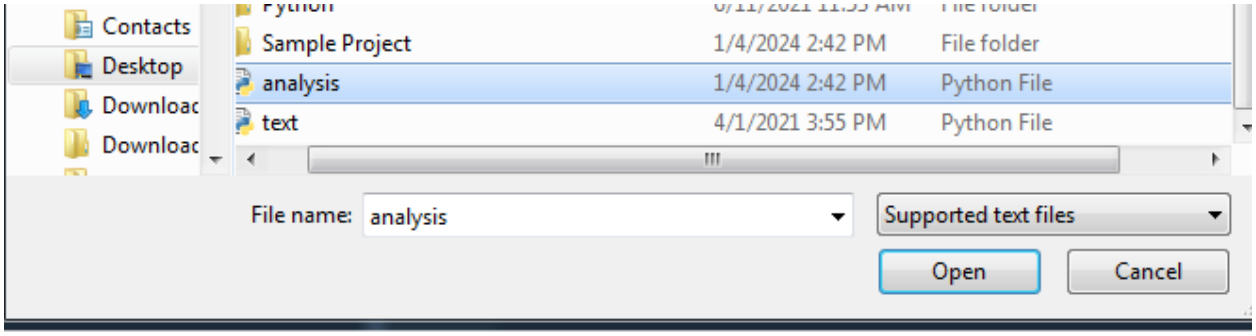
However, the option is not limited to running the entire file. But single cells or multiple cells can also be processed using  and  .

3. Executing Python Files from the Command Prompt

The Spyder IDE supports working with existing Python files. To open the file which is saved on any other location, click on File > Open.



The window shows the location of the project which you have created. If you want to access the script from any other location, select the desired location and the file.



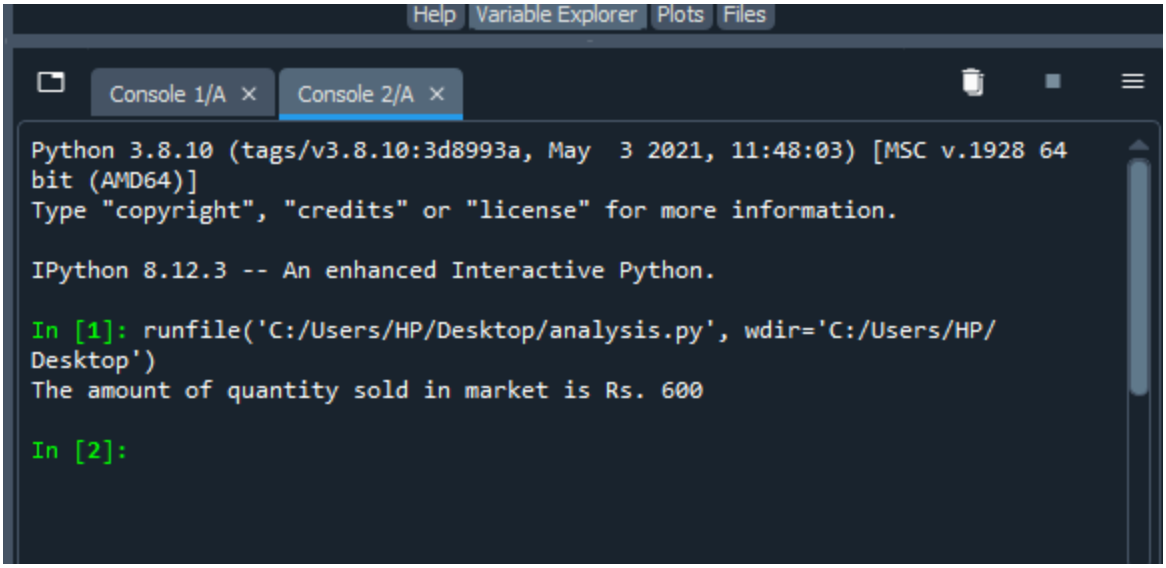
With this, click on open. The saved Python script will open. The code stated in the script is

```
1  a = 10
2  b = 60
3  c = a*b
4  print ("The amount of quantity sold in market is Rs.",c)
```

Click on the run icon to execute the file. In the same console, the output of the file is derived i.e.

```
In [2]: runfile('C:/Users/HP/Desktop/analysis.py', wdir='C:/Users/HP/
Desktop')
The amount of quantity sold in market is Rs. 600
```

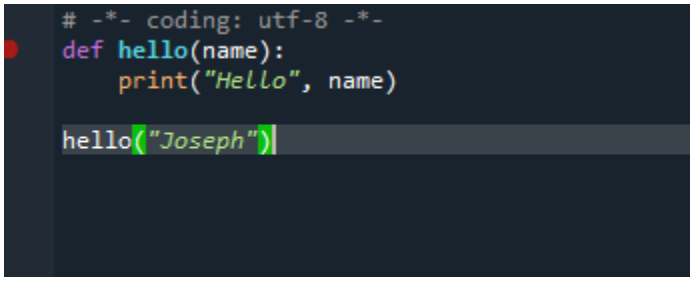
For deriving results in a separate console, click on consoles>new console (Default settings) or you can select ctrl+T as a shortcut for creating a new console. Execute the code now and the result will be derived in the new console.



4. Passing Arguments to Your Script

The Spyder environment also provides the feature of passing arguments to scripts. Arguments are simply defined as the variables which are passed in the code. For example – a coder might want to pass the data file name or several iterations to be performed. It is simply the means wherein the user has to enter the information while running the script i.e. on the console window. The arguments could be parsed in two ways

1. Explicitly wherein arguments are added only for test cases. In this case, only the test case will have access to the argument



   Herein, as the name *Joseph* is given specifically for *hello*, therefore, it is an explicit argument wherein only the *hello* method has access to the name data i.e. *Joseph*

2. Implicitly is by using *sys.argv* variable. The method provides access to variables used or maintained by the interpreter. The access to arguments is not limited to a test case. instead, the arguments or variables could be accessed in the script. The variable serves as

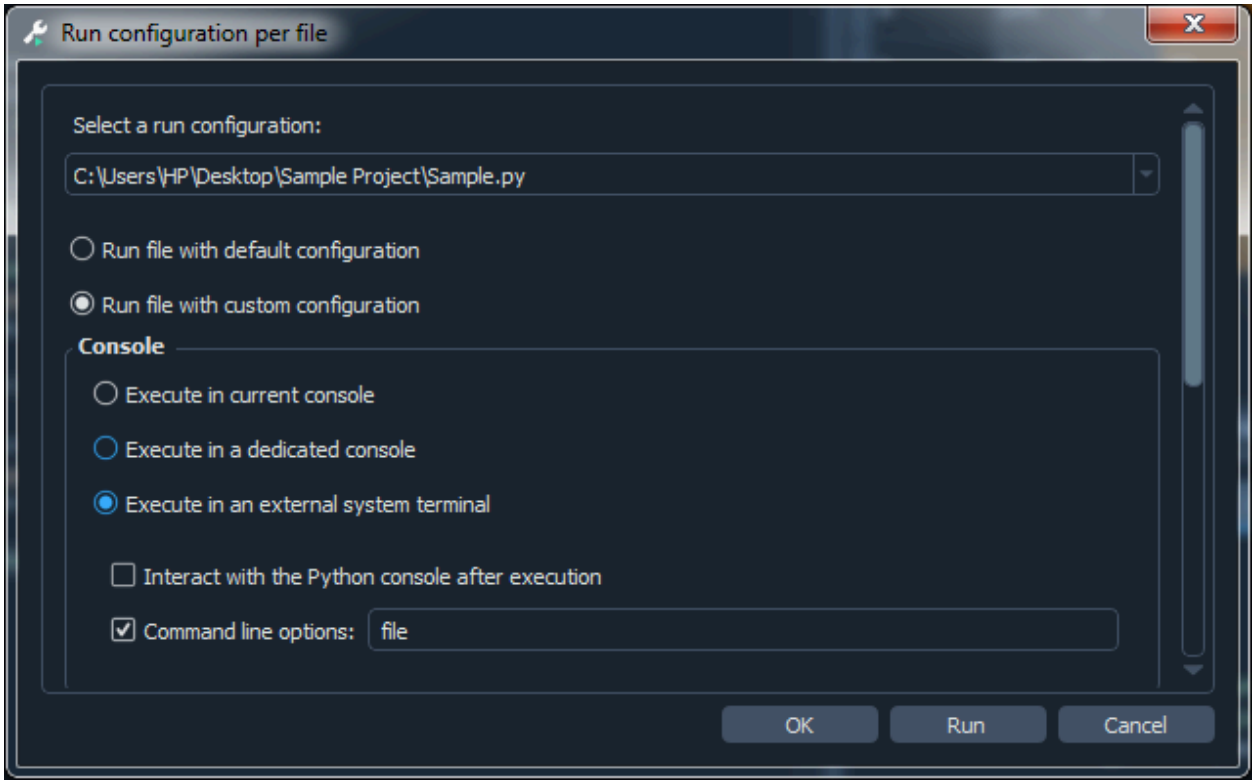
the command line argument. Herein, *len(sys.argv)* is the command used for determining argument length while the sys.argv[0] defines the Python script name. For example – There is a Python script wherein two numbers need to be added and passed as command-line arguments. The code developed for this is

```
import sys

# total arguments
length = len(sys.argv)
print("Total number of arguments passed:", length)

# name of file
print("\nName of script file:", sys.argv[0])

print("\nValues of arguments passed:", end = " ")
for j in range(1, length):
    print(sys.argv[j], end = " ")

# multiplication of passed numbers
product = 1
# Using argparse module
for a in range(1, length):
    product*= int(sys.argv[a])

print("\n\nProduct result:", product)
```

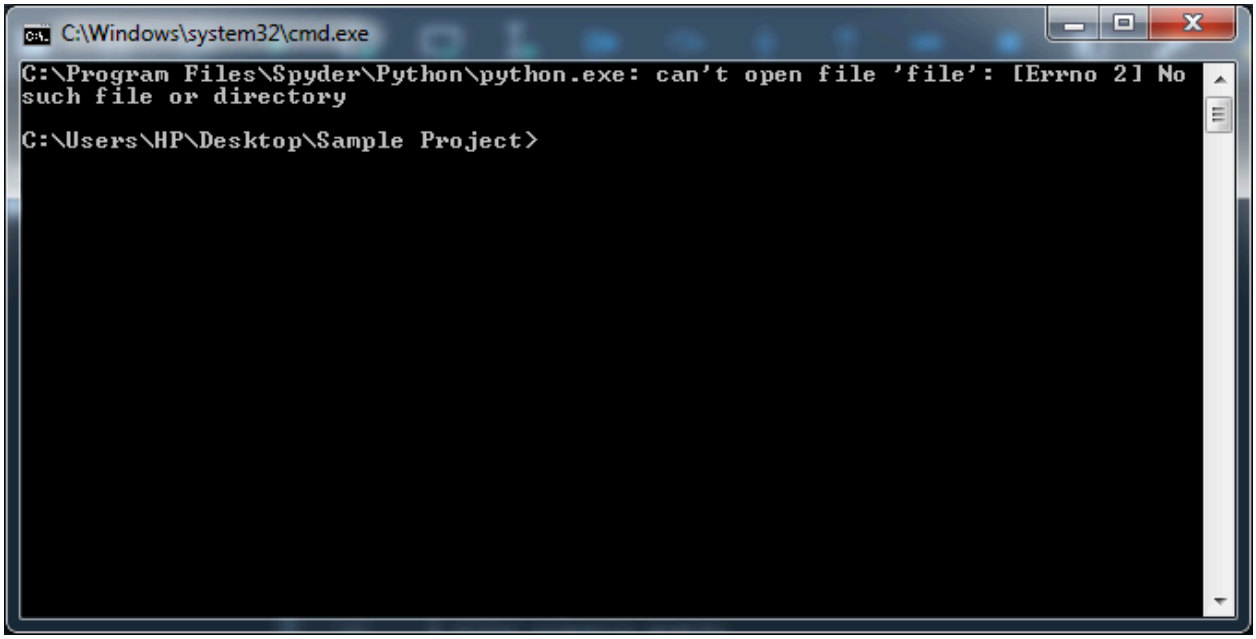
For this code, the derived output is

```
In [11]: runfile('C:/Users/HP/Desktop/Sample Project/Sample.py', wdir='C:/
Users/HP/Desktop/Sample Project')
Total number of arguments passed: 1

Name of script file: c:\users\hp\desktop\sample project\sample.py

Values of arguments passed:

Product result: 0
```

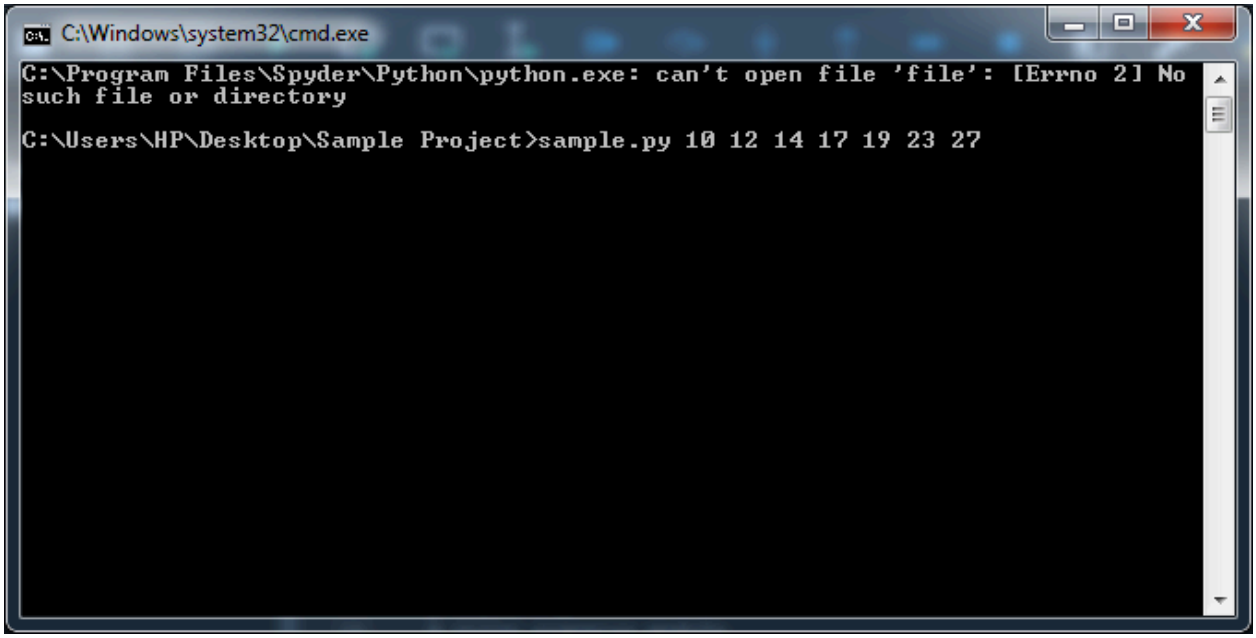
Herein, the need is to interact with the console for passing the arguments. To do it click on run>configuration per file



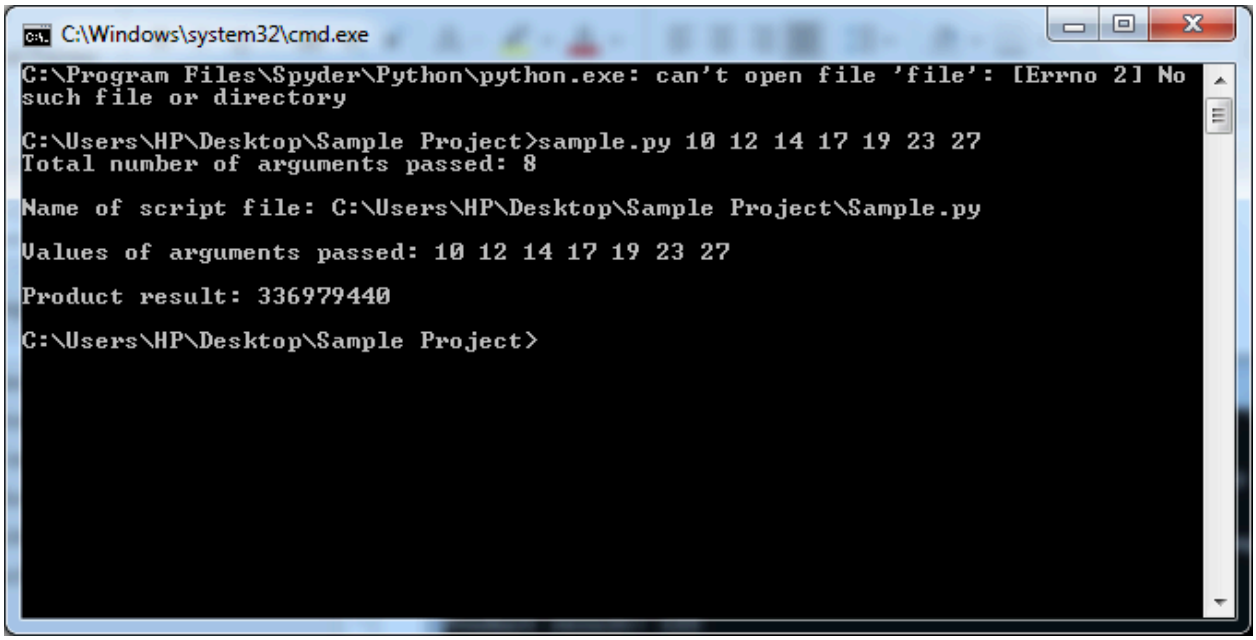
Under the command line option, type a file name and click on run



A command prompt window will appear. Type here the file name and the inputs or arguments you want to pass, herein it's a sample.py file and the arguments passed are 10, 12, 14, 17, 19, 23 and 27. Post this, click on enter



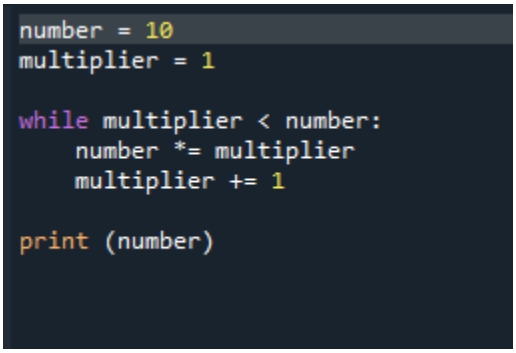
The output window will appear



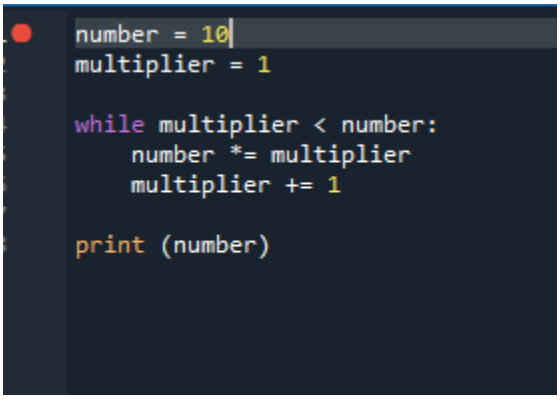
The above figure shows that a total of 8 arguments were passed i.e. 1 file name and 7 values. The script file name is shown, then values which were passed are stated and finally their product result i.e. 336979440.

5. Debugging Your Code in Spyder Command Prompt

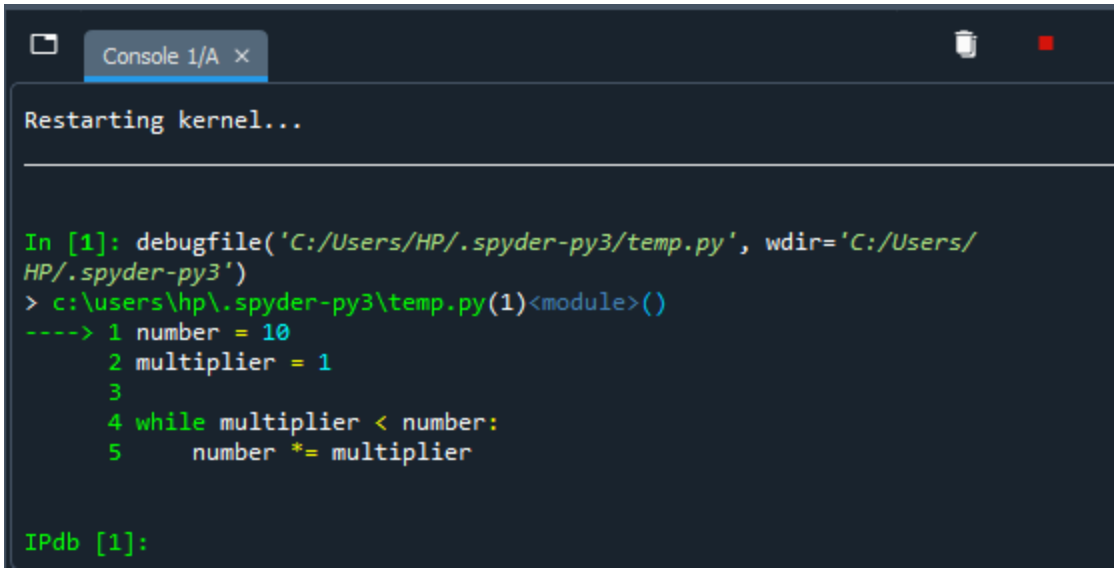
The spyder command prompt enables the active running of Python scripts but sometimes due to non-syntactic errors during coding, there is a derivation of the error summary in the console pane. This presence of error prevents the execution of the Python script. To correct these errors, there is a presence of debugging in the spyder command prompt. The debug tool serves as the means of looking at the values for catching logical errors along with helping the coder to identify the line which is causing the script to crash. For example – the factorial needs to be created for an integer i.e. 10. To compute this the code used is

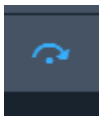


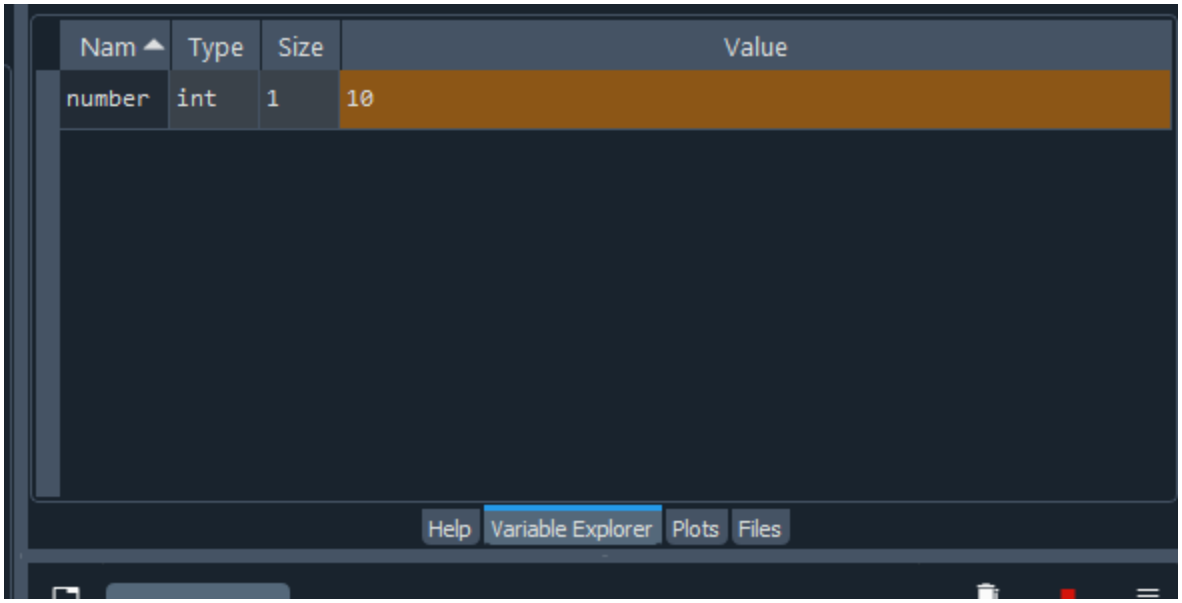
To debug the code initially set the breakpoint. The breakpoint is defined as the point at which the coder wants to stop code from running so that line-to-line examination of code can be done. Often the breakpoint is set in the middle of long scripts so that even a single line of code need not be examined. As the code is small so we are adding a breakpoint at the start of the code i.e. 1st line. This breakpoint could be set by moving the cursor to the left of 1st line. A red dot could be seen. Click on it and the breakpoint is added.



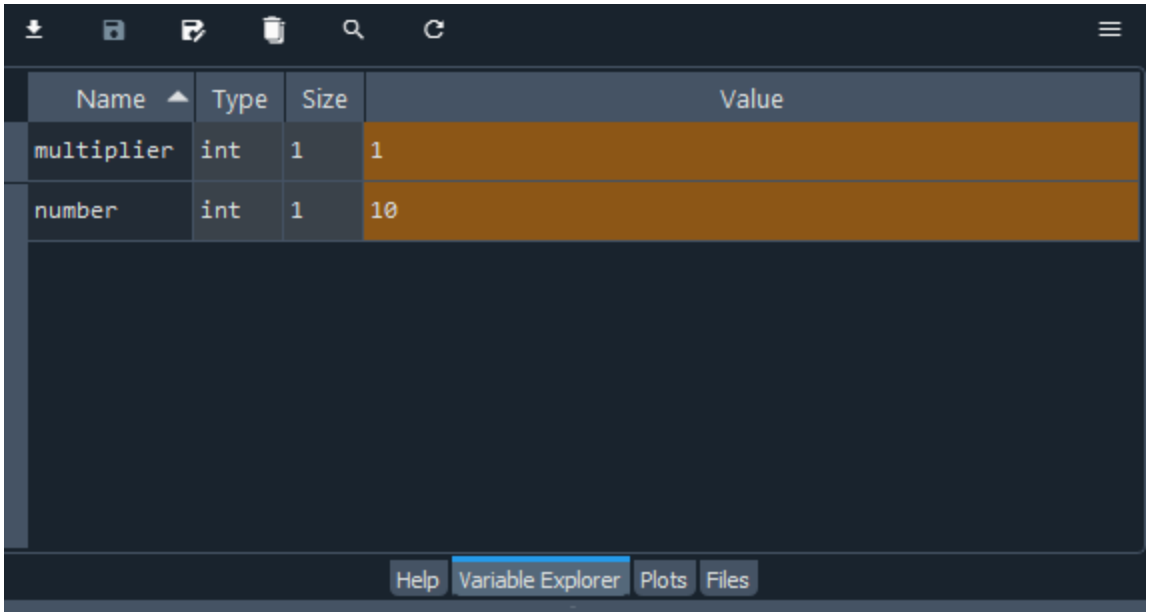
Click on debug. This leads to breakpoint execution. The debugfile() function is running on the script and not the runfile() function so instead of a normal prompt, it's a debug prompt.



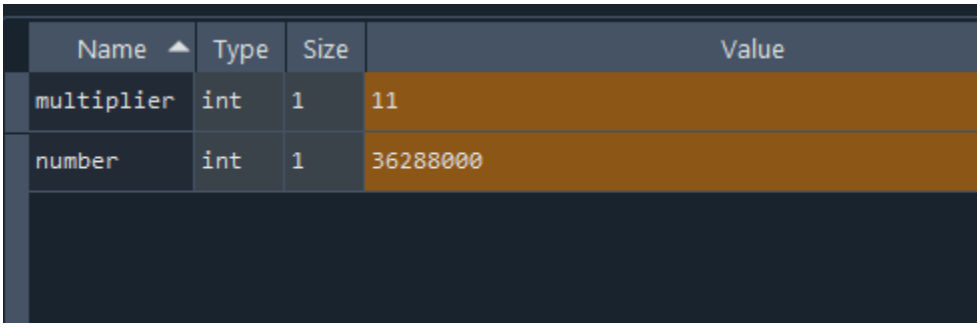
Click on  , run the current line, or  , step into the function, icon. The icon enables running 1st line i.e. number = 10. Both functions execute the selected line. Before going further, check variable explorer wherein a variable is added number with the value 10



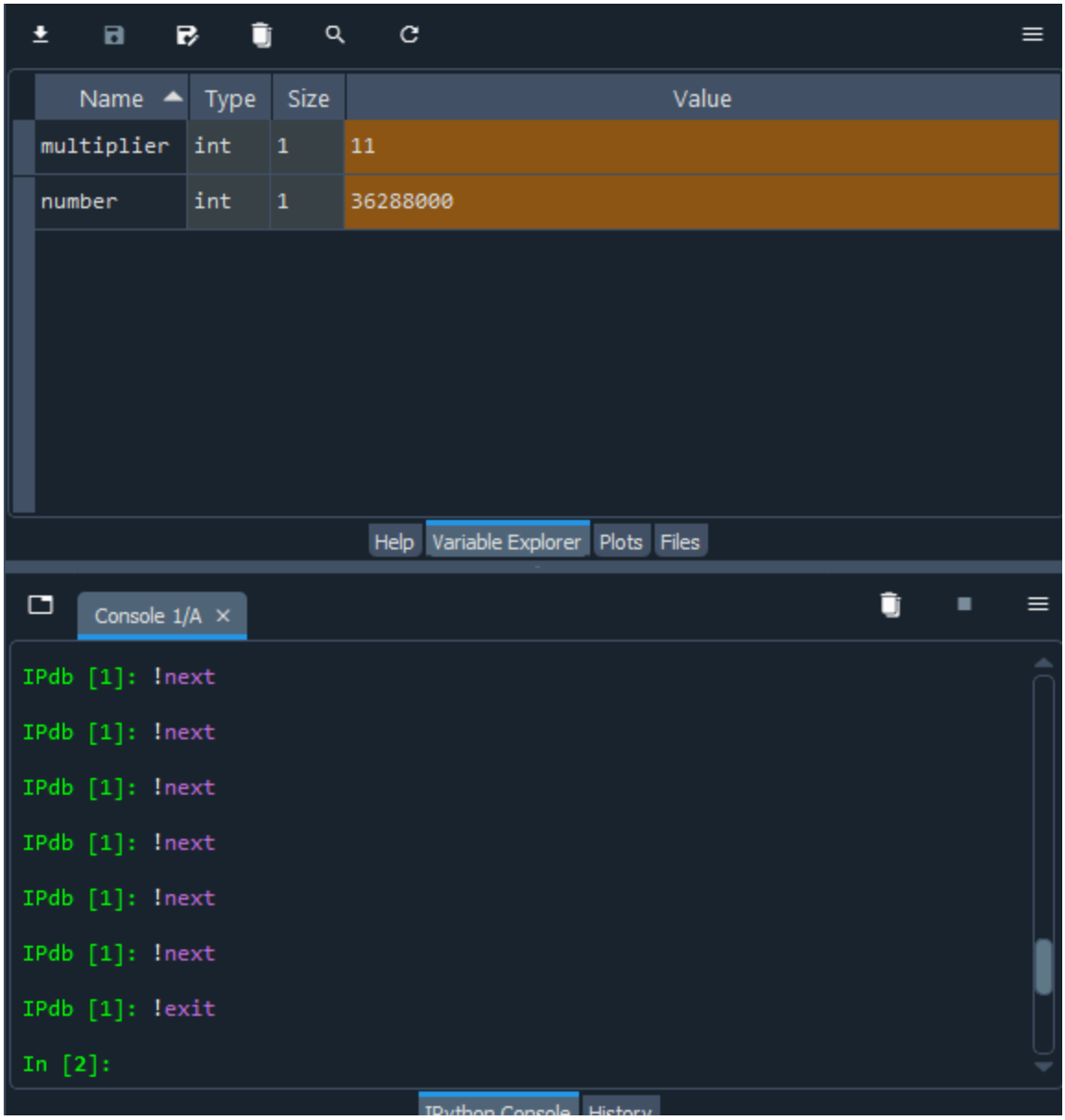
Similarly, execute the second line and see another variable addition.



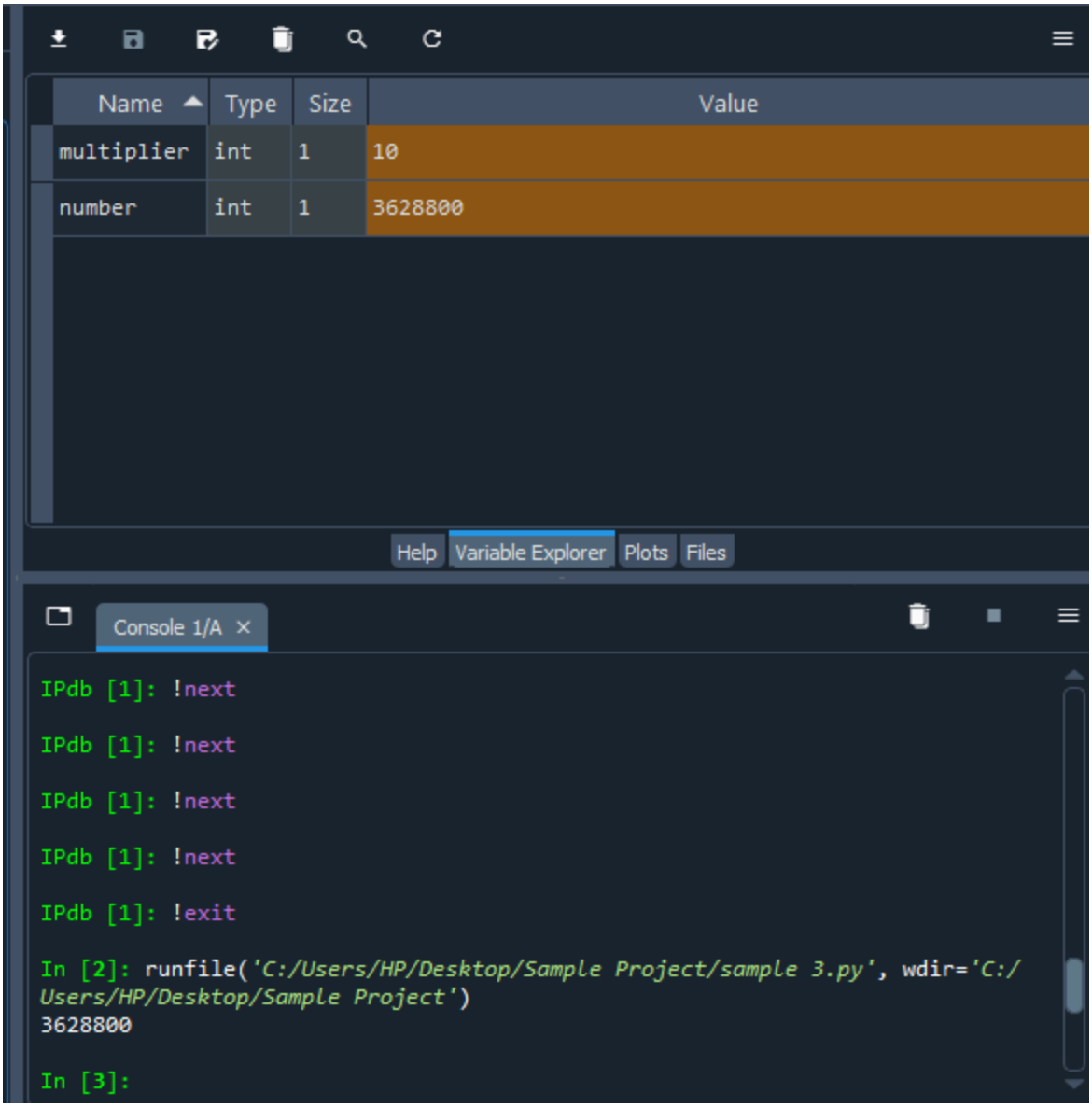
Continue this step until the multiplier value is derived to be 10



The code was required to stop at the value of less than 10 but it continued showing the presence of error in steps. So, herein first correction is to change logical error i.e. have the *while* loop with *multiplier < 10*. Click the stop button  to stop debugging.



Though the debugging has stopped but the variable window still shows a value wherein the session was stopped i.e. when the multiplier value was 11. After this the code was executed and the result was

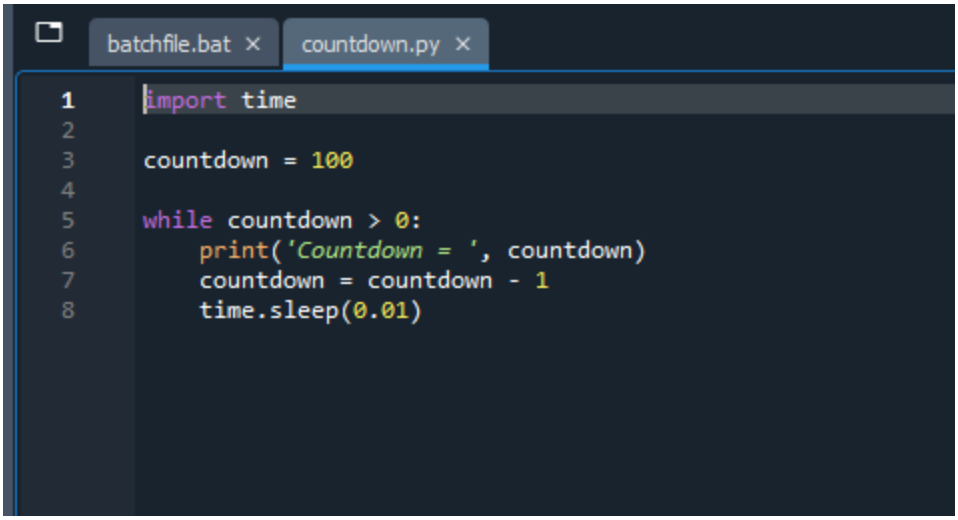


The above shows the correct value of 10 factorial i.e. 3628800.

Thus, debugging is an effective means of solving errors in the code and reviewing the work of the code.

6. Creating and Running Batch Files

Batch files are simply said to be storage of commands which need to be executed in serial order. To start with the creation of batch files, initially create a Python script. Suppose a script is prepared to have a countdown.



The file is saved on the desktop with the name *countdown.py*. Now, to create a batch file, check the location of the python.exe file i.e. click on start and search python. Right-click on the app and check the open file location. The location derived for us was

"C:\Users\HP\AppData\Local\Programs\Python\Python38-32\python.exe"

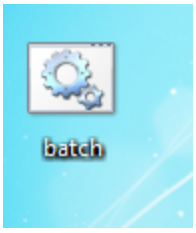
Similarly, find the location of the countdown file i.e.

"C:\Users\HP\Desktop\Sample Project\countdown.py"

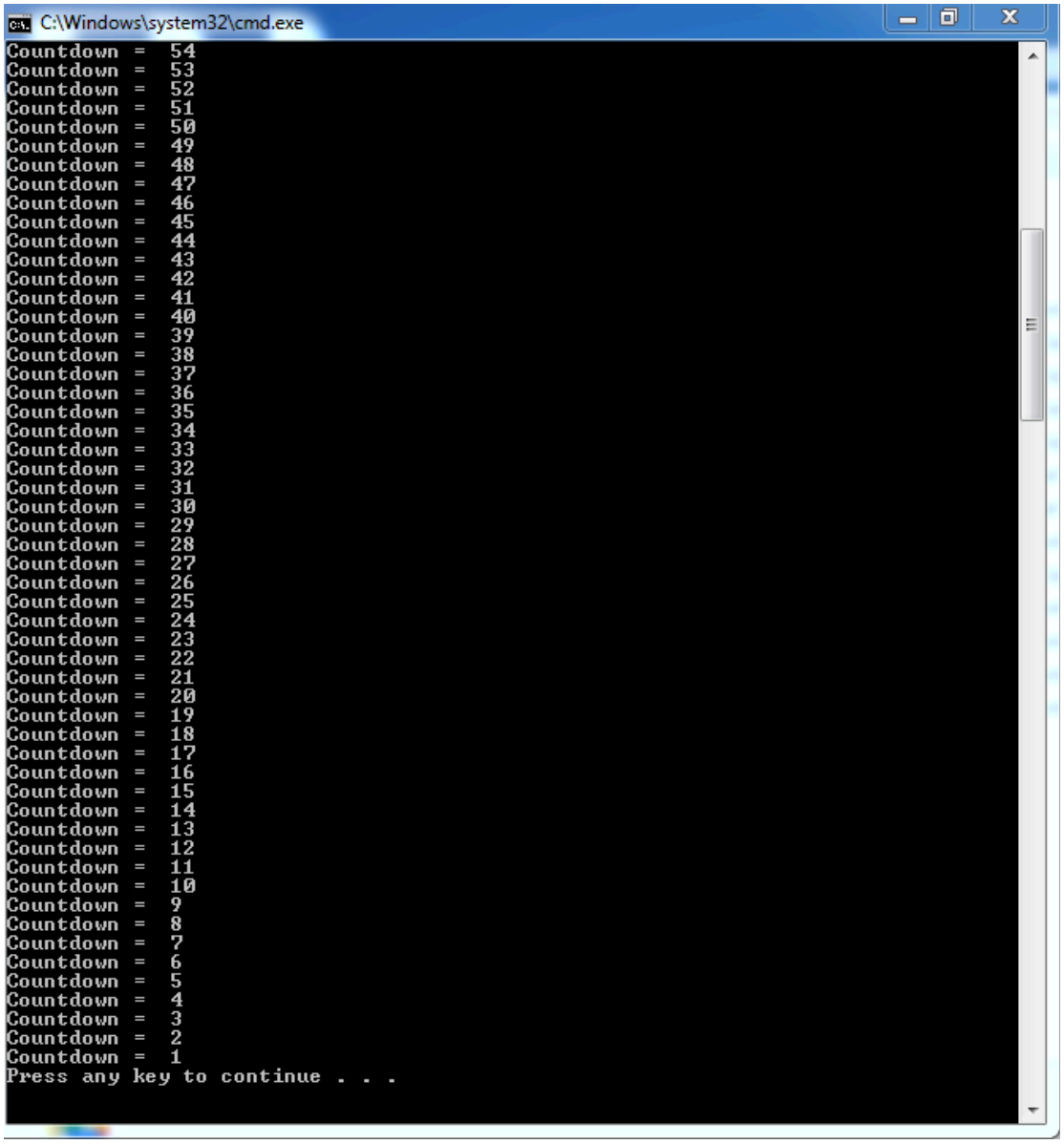
Create a new text file and paste the code there as

```
File  Edit  Format  View  Help
@echo off
"C:\Users\HP\AppData\Local\Programs\Python\Python38-32\python.exe"
"C:\Users\HP\Desktop\Sample Project\countdown.py"
pause
```

Save the file as *.bat* file.



Double-click on the file to see the results of a Python script.



This shows that the countdown is derived.

7. Creating and Running Shell Scripts

The shell script is defined as the text file which contains a sequence of commands for running on the UNIX operating system. The scripts are created by below stated steps

1. Open the terminal and go to the directory wherein the script needs to be created
2. Create the file using .sh
3. The editor is used for writing the script in a file

Once the file is created, the script could be executed using *chmod +x <fileName>* command and running the script using *./<fileName>* command like if the file name is analysis then it could be *chmod +x analysis* and *./analysis*

8. Common Issues and Troubleshooting Tips

Spyder won't start

The most common problem with Spyder is it crashes, freezes or receives error messages while running the environment. To overcome this the coder could

- Restart spyder
- Upgrade spyder to get the latest release using the check for updates command under help. The command for updating the version in Anaconda is *conda update spyder* or *conda update anaconda*
- Update the Sypder's environment and dependencies using conda with the command *conda update – all*
- Restart the machine
- Resetting the environment by using the command *spyder – reset*
- Install spyder into new conda environment
- If none of the options worked, uninstall the spyder and reinstall it using an anaconda environment.

Performance issues

Sometimes, the processing time is very slow with Spyder. This could be due to lots of unnecessary console presence. So, try to

- Close unnecessary consoles
- Restart spyder
- Reduce the number of objects in the variable explorer

Kernel connection issues

There is a loss of connection with the kernel i.e. ipython console or script execution. To troubleshoot it

- Restart the kernel
- Check the Ipython console for the error message

Package installation issue

The Spyder is not able to install different packages. The troubleshooting of package installation issues could be done by

- Using the package manager or Anaconda navigator for installation like pip or conda
- Verify that the Python interpreter is using the correct environment for installation

Appearance or Font issue

Sometimes, the font or presentation of code is not as per the coder's preference. To resolve this problem, the coder could

- Adjust font settings for better appearance and readability issues
- Ensure display settings are as per preference

9. Summary

Scientific Python development environment (Spyder) is an open-source and free scientific environment used by data analysts, engineers, and scientists for introspection, debugging, interactive testing, and editing.

The environment with Spyder could be set up by having Spyder online, using standalone installers or using a virtual environment such as Anaconda.

The Spyder environment consists of three windows i.e. editor, console, and object inspector.

The Spyder environment also provides the feature of passing arguments to scripts either implicitly or explicitly.

To correct these errors, there is a presence of debugging in the spyder command prompt.

Batch files could be executed by double-clicking on the .bat file extension file and having the statement of commands to be executed in it.

There are many problems like starting issues, performance issues or kernel connection issues with Spyder but by adopting easy steps, the issues can be resolved.

Chapter: 7 Working with Jupyter Notebook

Objectives

To discuss the working with Jupyter notebook

- Understand the Jupyter notebook environment
- Working with Jupyter notebook files
- Visualization of data with Jupyter
- Debugging code with Jupyter
- Describe the issues with Jupyter notebook and means of troubleshooting them

0. Introduction

The Jupyter Notebook is an open-source web-based interactive application which can be used for sharing or creating documents which have live text, code, visualizations, and equations. The notebook is an IPython project spin-off project which includes the IPython notebook itself. Jupyter supports core languages i.e. Julia, Python and R and provides the IPython kernel for writing programs. The Notebook provides all things in one place, is easy to convert and share, and supports interactive code, thus, the Jupyter Notebook usage is popular among data scientists. This chapter focuses on providing information about basic and advanced concepts of the Jupyter

Notebook along with discussing some of the common issues with the Jupyter Notebook and ways of resolving them.

1. Installing and Launching Jupyter Notebook, Interface Overview, Markdown Basics

The Jupyter Notebook is not installed along with Python. To install the notebook, there are two ways – one is using CPython which is Python's reference version and another is using Anaconda. For the 1st step, the condition is to have Python installed (refer to Chapter 1 for Python installation). It can be checked by opening a command prompt and typing the following command

```
C:\Users\jains>py --version
Python 3.12.1
```

This shows we have Python 3.12.1 installed. Now, pip installation can be checked using the command

```
C:\Users\jains>py -m pip --version
pip 23.2.1 from C:\Users\jains\AppData\Local\Programs\Python\Python312\Lib\site-packages\pip (python 3.12)
```

The output shows that pip is installed in the environment. Type the command *py -m pip install jupyter* and click enter.

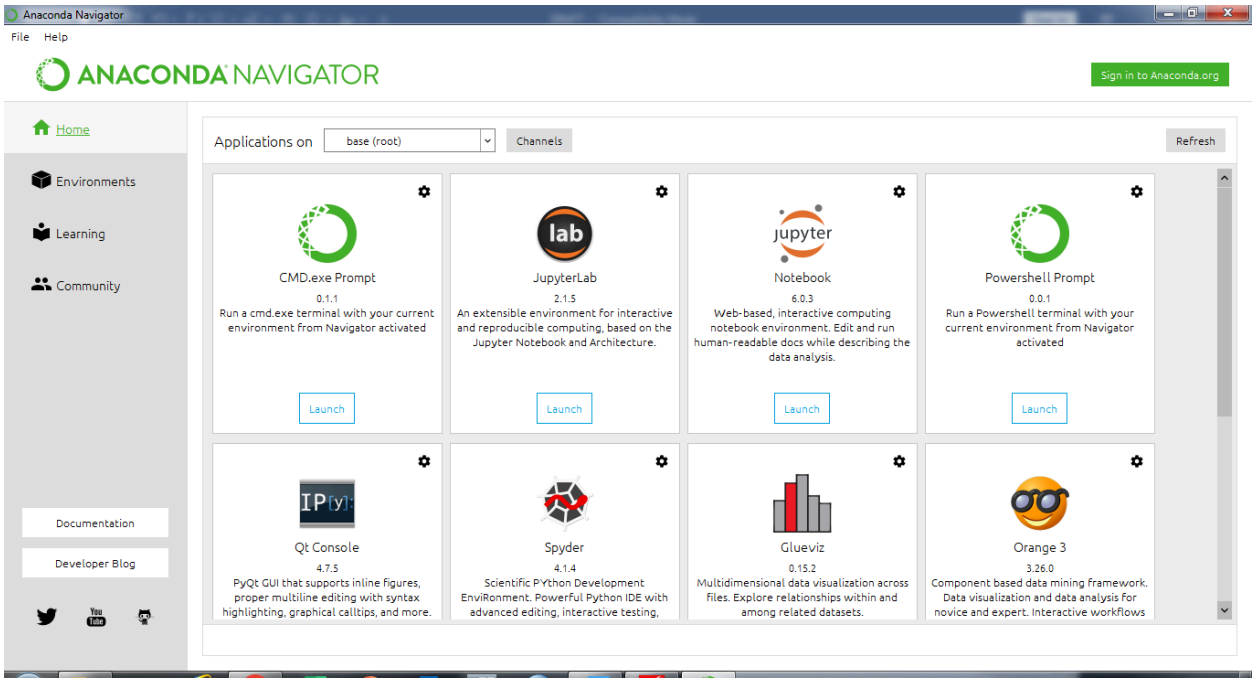
```
C:\Users\jains>py -m pip install jupyter
Collecting jupyter
  Downloading jupyter-1.0.0-py2.py3-none-any.whl (2.7 kB)
Collecting notebook (from jupyter)
```

After successful installation, the message displayed is this

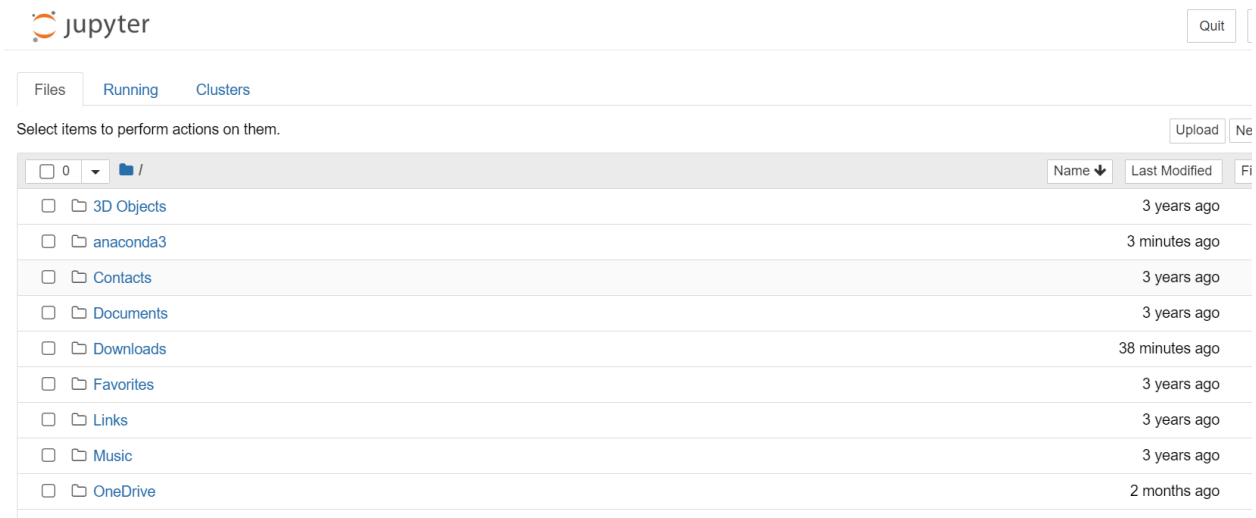
```
Successfully installed anyio-4.2.0 argon2-cffi-23.1.0 argon2-cffi-bindings-21.2.0 arrow-1.3.0 asttokens-2.4.1 async-lru-
2.0.4 attrs-23.2.0 babel-2.14.0 beautifulsoup4-4.12.2 bleach-6.1.0 certifi-2023.11.17 cffi-1.16.0 charset-normalizer-3.3
.2 colorama-0.4.6 comm-0.2.1 debugpy-1.8.0 decorator-5.1.1 defusedxml-0.7.1 executing-2.0.1 fastjsonschema-2.19.1 fqdn-1
.5.1 idna-3.6 ipykernel-6.28.0 ipython-8.19.0 ipywidgets-8.1.1 isoduration-20.11.0 jedi-0.19.1 jinja2-3.1.2 json5-0.9.14
 jsonpointer-2.4 jsonschema-4.20.0 jsonschema-specifications-2023.12.1 jupyter-1.0.0 jupyter-client-8.6.0 jupyter-consol
e-6.6.3 jupyter-core-5.7.0 jupyter-events-0.9.0 jupyter-lsp-2.2.1 jupyter-server-2.12.2 jupyter-server-terminals-0.5.1 j
upyterlab-4.0.10 jupyterlab-pygments-0.3.0 jupyterlab-server-2.25.2 jupyterlab-widgets-3.0.9 markupsafe-2.1.3 matplotlib
-inline-0.1.6 mistune-3.0.2 nbclient-0.9.0 nbconvert-7.14.0 nbformat-5.9.2 nest-asyncio-1.5.8 notebook-7.0.6 notebook-sh
im-0.2.3 overrides-7.4.0 packaging-23.2 pandocfilters-1.5.0 parso-0.8.3 platformdirs-4.1.0 prometheus-client-0.19.0 prom
pt-toolkit-3.0.43 psutil-5.9.7 pure-eval-0.2.2 pycparser-2.21 pygments-2.17.2 python-dateutil-2.8.2 python-json-logger-2
.0.7 pywin32-306 pywinpty-2.0.12 pyyaml-6.0.1 pyzmq-25.1.2 qtconsole-5.5.1 qtpy-2.4.1 referencing-0.32.0 requests-2.31.0
 rfc3339-validator-0.1.4 rfc3986-validator-0.1.1 rpds-py-0.16.2 send2trash-1.8.2 six-1.16.0 sniffio-1.3.0 soupsieve-2.5
stack-data-0.6.3 terminado-0.18.0 tinycss2-1.2.1 tornado-6.4 traitlets-5.14.1 types-python-dateutil-2.8.19.14 uri-templa
te-1.3.0 urllib3-2.1.0 wcwidth-0.2.12 webcolors-1.13 webencodings-0.5.1 websocket-client-1.7.0 widgetsnbextension-4.0.9
```

Herein, you can launch the Jupyter using the command *jupyter notebook*.

Another method is of Anaconda navigator. The virtual environment already has Jupyter Notebook installed. So it can directly be launched from the anaconda navigator by scrolling to the notebook and clicking on Launch.

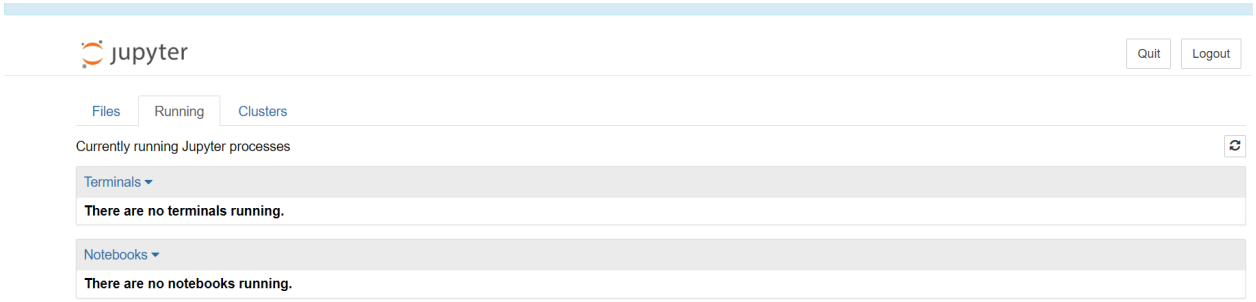


The jupyter window server

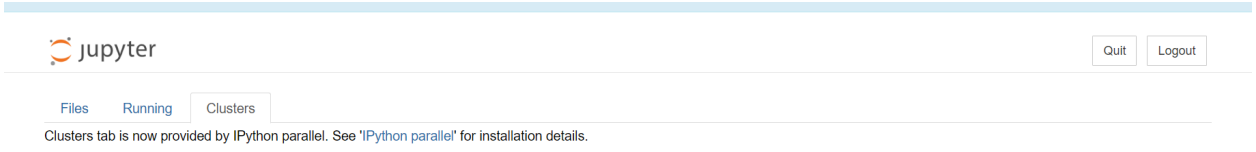


The dashboard of Jupyter shows three tabs i.e. files tab which displays the folders and files present in the current directory. The upload option enables uploading any new document and

supports new notebook creation. The last modified status defines the time when the file was last saved. The second tab is running

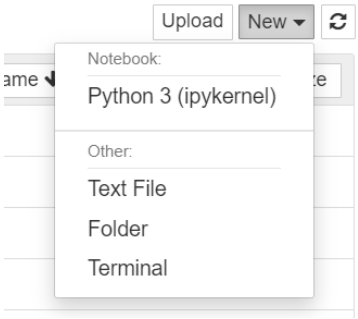


The tab represents which notebooks are running currently. The last tab is of cluster tab

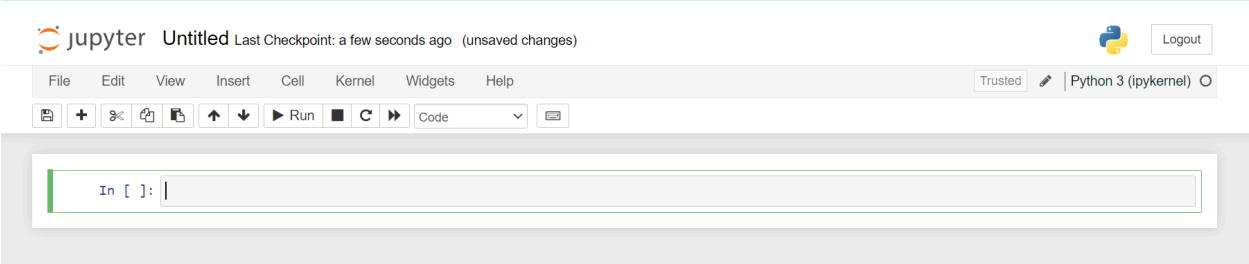


This tab is given by IPython parallel.

To understand the user interface, create a new notebook by clicking on new>python3



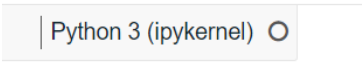
A notebook is created.



In this, untitled represents the notebook name. By clicking on it, the name of the notebook can be changed. Then there is a menu bar which includes different options like file, edit, or view.

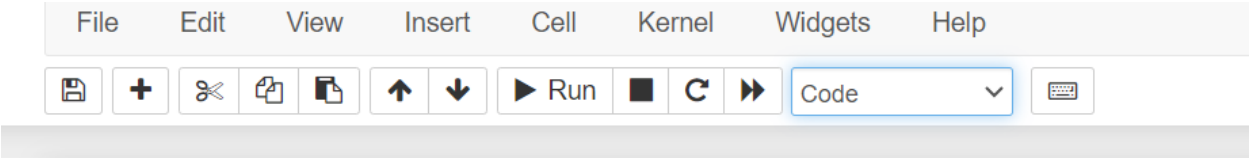
- In the File menu, the options are to create a new notebook, open, save as, rename, save, revert (revert to earlier checkpoint), and download.
- The Edit menu includes buttons to cut, paste or copy cells; split and merge cells, delete selected cells, move cells up and down, cut/copy attachments, find and replace with notebook, and have insertion of images.
- The View menu is for displaying or hiding the toolbar, header or cell numbers.
- The Insert menu provides an option for inserting cells after or before a cell.
- The Cell menu enables the user to run specific cells or multiple cells along with having the option to set cell type to code type, raw nbconvert type or markdown.
- The Kernel menu helps in starting, stopping, restarting or interrupting the kernel. Even a new kernel can be started
- The Widget menu helps in saving, downloading, embedding or clearing the widget state.
- Lastly, the Help menu helps in editing shortcuts as per convenience and displays existing pre-defined shortcuts.

Lastly is the cells or rows icon wherein the operations can be performed. The notebook generally has 2 modes i.e. edit mode and command model. The notebook enters edit mode when clicking on the cell for writing code.

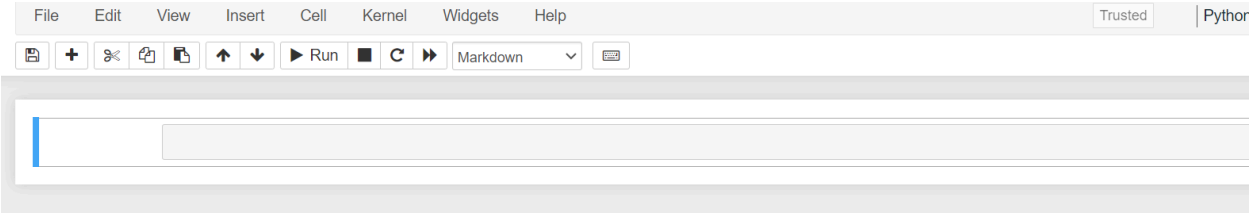


Kernel status is displayed by this circle wherein hollow means idle while solid depict busy.

The Jupyter notebook interface has markdown cells which helps in displaying text using markdown language.



For considering the cell to be markdown, initially change the dropdown from code to markdown.



Now, the markdown cell could have the creation of header 1 by # header 1, header 2 by ## header 2 and so on. For example, for # *Introduction,* ## *Overview,* ### *Brief*, the output is

Introduction

Overview

Brief

The bold form of text could be derived by putting text between double underscores or asterisks for bold while italic form by putting text between single asterisks or underscore. The ordered list can be created by starting the first number as 1 and subsequent items can be given any number. It will automatically render it in an ordered way. For the sublist, use indent. Suppose, a list is stated

```
1. Age
    1. 18-20
    7. 20-30
6. Gender
    1. Male
    9. Female
```

Then its output is

1. Age
    A. 18-20
    B. 20-30
2. Gender
    A. Male
    B. Female

Similarly, for a bullet list if the list starts with -, then a solid circle is displayed while if the list starts with * the solid square symbol is displayed for the list. Further, the markdown cell text starting with HTTP or HTTPS is automatically regarded as a hyperlink For attaching a link to text, place text in [] and link in () like

```
[Google](http.google.com)
```

Output for a given cell is

Google

To add an image in the markdown cell, click on Insert an image from the edit menu and browse to the desired image. Lastly, a table can be constructed in a mardown cell by using dash (-) and pipe (|) symbols for stating rows and columns. Symbols need exact alignment. Suppose a table is constructed to keep the name and age of the person

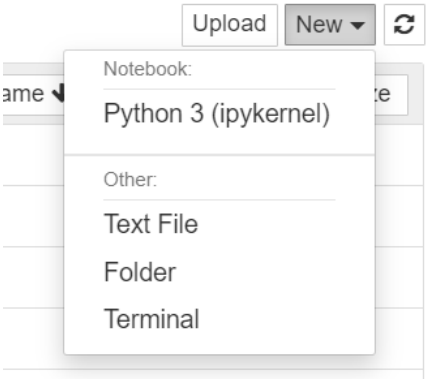
```
|Name|Age|
|----|---|
|Yogesh|34 years|
```

The output for the code is

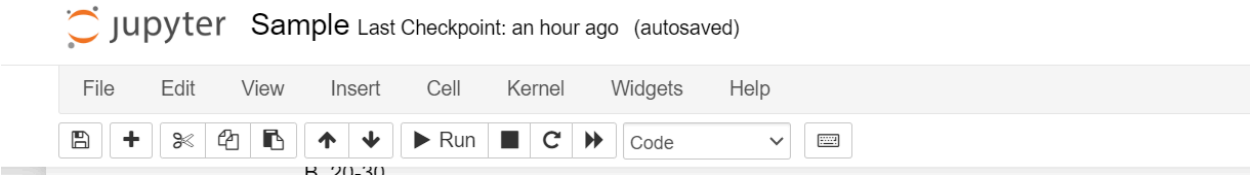
Name	Age
Yogesh	34 years

2. Creating and Managing Jupyter Notebook Files

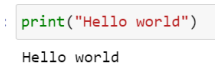
To examine the creation of a new notebook with Jupyter, click on new and select Python 3



This will create a new notebook. The name of the notebook can be edited by clicking on the text *'untitled'*. Rename the untitled text to the desired file name like the sample here.



The first cell becomes accessible with the creation of a new notebook. The code could be written in the cell and using an icon or shortcut shift + return, the cell could be executed. The output will be derived underneath the code cell. Suppose, a simple command i.e. print hello world is written, the the same output is derived.



3. Working with Code Cells

The Jupyter Notebook enables the execution of files. Along with managing the new notebook files and already created notebook files, Jupyter also enables adding checkpoints to the notebook. The checkpoint saves the current notebook state enabling it to revert later in case changes are made to the notebook. For creating a checkpoint click on file > save and checkpoint. This adds checkpoints and saves notebook files. In case you want to move to an earlier saved checkpoint, then select File > revert to checkpoint option.

When a code is executed, a new cell is automatically inserted enabling insertion of code or mardown.

```
In [4]: print("Hello world")
        Hello world

In [ ]:
```

To write multiple lines in a single cell, just click on enter. For example, the sum of a and b is printed

```
In [5]: a = 10
        b = 5
        print(a+b)

        15
```

Along with writing in one cell, each command could be returned to a different cell using the icon.

Further, the cells could also be added in edit mode using Alt+shift + A and in command mode using A for adding the cells above. For adding cell below in edit mode select Alt+Shift+B and in command mode select B. For selecting the cell, click gutter next to the cell and for selecting multiple cells, click gutter cell by holding on shift for consecutive cells or ctrl for non-consecutive cells. The cells could be copied in command mode by clicking on the copy icon or pressing ctrl+C. To paste the copied cell click on the paste icon or ctrl+V. For pasting the cell above the selected cell press shift along with shift +V/ Ctrl +V. Cells could also be merged by right-clicking on the cell and then selecting option merge cell below or merge cell above. A cell can be deleted using the delete icon or pressing delete. Cells could be executed individually or altogether using or  . Herein, the latter icon not only executes all cells but also restarts the kernel.

Along with executing the code, the notebook could also be exported using the command file>download. These files could be downloaded as notebook, HTML, python, markdown, LaTeX, rest, or PDF.

4. Running and Debugging Code, Using Magic Commands

The notebook cells could be executed in many ways like using ctrl + enter for the current cell or shift
+ Enter for running the current cell and selecting the below cell. In case the result of the cell relies on some other cell, then the previous cell should be executed first. Also, jupyter though all

codes are present in one cell prints results of only the last line without a *print* command. Suppose a code is developed

```
In [1]: a = 10

In [2]: b = 4

In [3]: a+b
        a-b

Out[3]: 6

In [4]: print(a+b)
        print(a-b)

        14
        6
```

The above code shows that in line 3 only out for the last code is printed i.e. *a-b*. But when individually printed using the print command, the results of both codes are printed. Working with complicated working, suppose a list is created and processing of it is done.

```
In [1]: value =[1,2,3,4]
        value

Out[1]: [1, 2, 3, 4]

In [2]: for v in value:
            print(v)

        1
        2
        3
        4
```

The above code shows the creation of a list and its value derivation as output. Following that, each value of the list was printed. Now, to understand the properties of a list, the code *value?* could be executed i.e.

```
In [4]: value?

In [ ]:
```

```
Type:        list
String form: [1, 2, 3, 4]
Length:      4
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.
```

The Jupyter Notebook supports the execution of code and also in case of error, debugging can be done. Support pandas need to be imported and the code typed to import is

```
In [1]: import panda as pd
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 import panda as pd

ModuleNotFoundError: No module named 'panda'
```

Herein error derived is *No module named panda*. This shows that the wrong module name is typed i.e. panda instead of pandas. Trying the work with another example and using the magic command. Magic command is simply the group of special commands which are used for helping with non-coding tasks. Some of the built-in magic commands are *%alias, %alias_magic, %autowait, %autocall, %automagic, %bookmark, %cd, %code_wrap, %debug,* or *%conda.* Now, to apply debugging with a complicated problem let's try the complicated problem. Suppose the following code is added

```
In [1]: a=1+1
        b=1-1
        c=a/b
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[1], line 3
      1 a=1+1
      2 b=1-1
----> 3 c=a/b

ZeroDivisionError: division by zero
```

Now, to debug the code, the magic command *%debug* is typed in the next cell. A box is added that inspects code without creating new cells.

```
[*]: %debug
     > c:\users\jains\appdata\local\temp\ipykernel_19804\3738679003.py

     ipdb>
```

Herein, each line of the code is typed and at the value of *c*, the code ends representing the error in line. We found the bug that the value of *b* is 0, so the code is not executing. Thus, the code is debugged.

```
In [1]: a = 2+1
        b = 2-1
        c = a/b
```

Now, the code is executed successfully, showing no error presence. Therefore, *%debug* could be used to identify the problem in the code.

5. Visualizing Data

The Jupyter Notebook, not only support the execution of code but also has the option to create plots and graphs for visualization. Data visualization is simply defined as the graphical presentation of data and information in a graphical or pictorial format like a pie chart, bar graph, or line chart. Data visualization helps in pattern recognition, data analysis, decision-making, memory retention, and efficiency of results. The data visualization could be done in the form of a bar chart, pie chart, line chart, histogram, scatter plot, bubble chart, heatmap, treemap, box plot, word cloud, choropleth map, network diagram, radar chart, or other graphs. The most common libraries used for visualization are *matplotlib, seaborn,* and *Plotly*. The important requirement for working with each of these libraries is to install the dependencies using *pip install*.

```
In [1]: pip install matplotlib
```

```
Requirement already satisfied: matplotlib in c:\users\jains\anaconda3\lib\site-packages (3.7.2)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib) (1.0.5)
Requirement already satisfied: cycler>=0.10 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib) (4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib) (1.4.4)
Requirement already satisfied: numpy>=1.20 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib) (1.24.3)
Requirement already satisfied: packaging>=20.0 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib) (23.1)
Requirement already satisfied: pillow>=6.2.0 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib) (9.4.0)
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in c:\users\jains\anaconda3\lib\site-packages (from python-dateutil>=2.7->matplotlib)
(1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

```
In [2]: pip install seaborn
```

```
Requirement already satisfied: seaborn in c:\users\jains\anaconda3\lib\site-packages (0.12.2)
Requirement already satisfied: numpy!=1.24.0,>=1.17 in c:\users\jains\anaconda3\lib\site-packages (from seaborn) (1.24.3)
Requirement already satisfied: pandas>=0.25 in c:\users\jains\anaconda3\lib\site-packages (from seaborn) (2.0.3)
Requirement already satisfied: matplotlib!=3.6.1,>=3.1 in c:\users\jains\anaconda3\lib\site-packages (from seaborn) (3.7.2)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.1->se
aborn) (1.0.5)
Requirement already satisfied: cycler>=0.10 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.1->seabor
n) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.1->s
eaborn) (4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.1->s
eaborn) (1.4.4)
Requirement already satisfied: packaging>=20.0 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.1->sea
born) (23.1)
Requirement already satisfied: pillow>=6.2.0 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.1->seabo
rn) (9.4.0)
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.
1->seaborn) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\jains\anaconda3\lib\site-packages (from matplotlib!=3.6.1,>=3.1
```

```
In [3]: pip install plotly
```

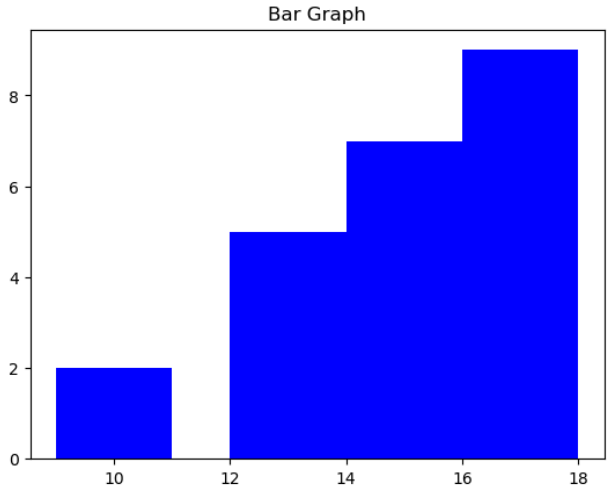
```
Requirement already satisfied: plotly in c:\users\jains\anaconda3\lib\site-packages (5.9.0)
Requirement already satisfied: tenacity>=6.2.0 in c:\users\jains\anaconda3\lib\site-packages (from plotly) (8.2.2)
Note: you may need to restart the kernel to use updated packages.
```

The bar graph using matplotlib could be plotted by importing matplotlib, defining x-axis and y-axis values and plotting the graph using the bar() function with customization options like width or color. Even a title could be added using the title() function and show() could be used to plot the graph

```
In [6]: import matplotlib.pyplot as plt
```

```
In [7]: x = [10, 13, 15, 17]
        y = [2, 5, 7, 9]
        plt.title('Bar Graph')
        plt.bar(x,y,color ='blue',width = 2)
        plt.show()
```

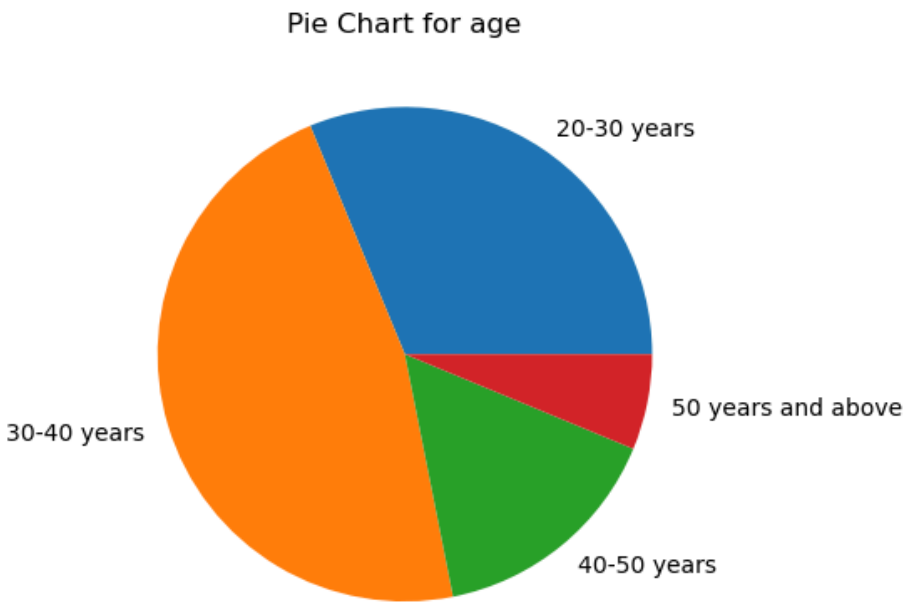
The output for the code would be



The pie chart using matplotlib is plotted by importing matplotlib, taking labels in the array, taking values in the array, plotting the pie chart using pie(), setting the title with the title() function, and showing the graph using the show() method. The code for a sample is stated below

```
In [8]: x = [20, 30, 10, 4]
        y = ['20-30 years', '30-40 years', '40-50 years', '50 years and above']
        plt.title('Pie Chart for age')
        plt.pie(x, labels=y)
        plt.show()
```

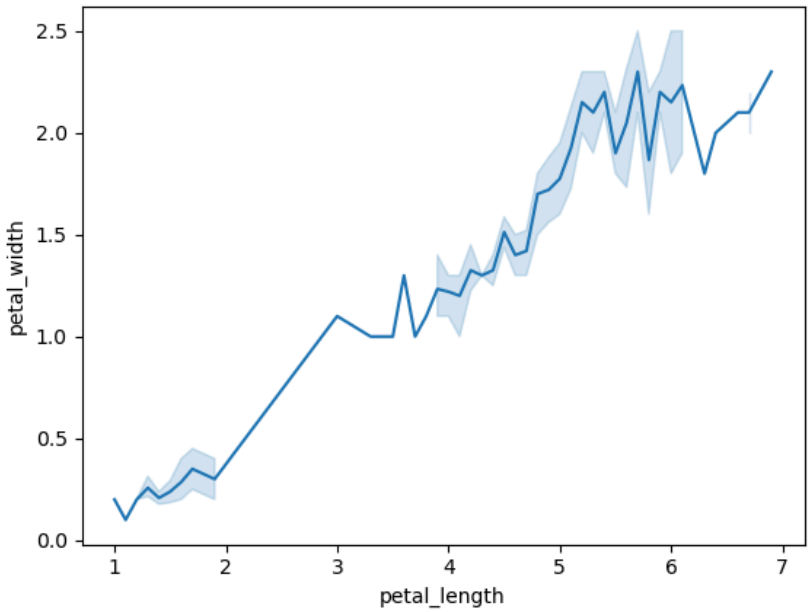
The output for the same is



Line plot could be created using seaborn by importing the module, loading data using load_dataset(), and using the lineplot() method. The sample for the same is

```
In [11]: import seaborn as sns
         data = sns.load_dataset("iris")
         sns.lineplot(x="petal_length", y="petal_width", data=data)
```

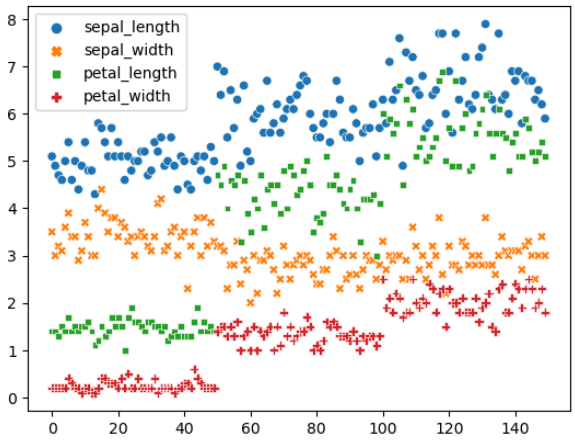
The output for the same is



A scatter graph could be created using import seaborn, loading the dataset using load_dataset(), and using the scatterplot() method. The sample for the same is

```
In [15]: import seaborn
         data = seaborn.load_dataset("iris")
         seaborn.scatterplot(data=data)
```

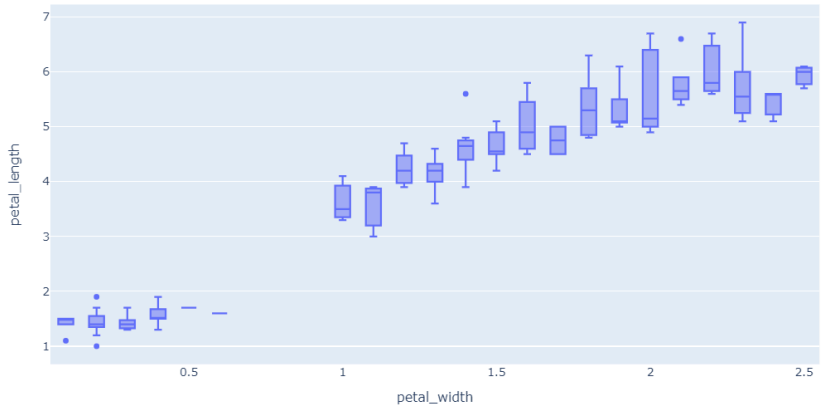
The output for the same is



The boxplot using Plotly could be created y importing the module, and loading the dataset with px.data.dataset_name() method, using box() method for plotting box plot, and show() for showing the figure. For the same, the boxplot could be drawn with the iris dataset

```
In [*]: import plotly.express as px
        df = px.data.iris()
        fig = px.box(df, x="petal_width", y="petal_length")
        fig.show()
```

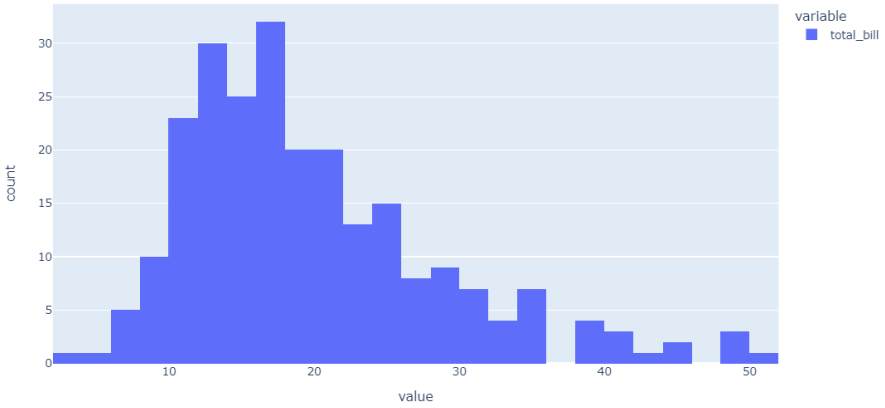
The output of the same is



Lastly, a histogram could be developed using the import module, loading the dataset px.data.dataset_name(), histogram() method, and show() method for showing figure. The sample of the plotting histogram is

```
In [18]: import plotly.express as px
         df = px.data.tips()
         fig = px.histogram(df.total_bill)
         fig.show()
```

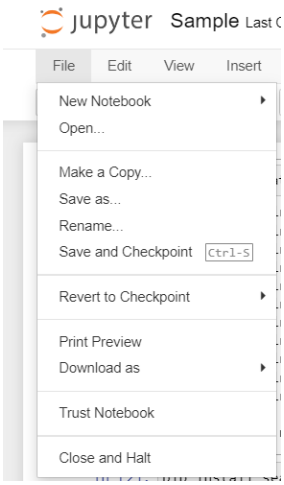
The output in this case would be



Thus, Jupyter Notebook enables the coder to visualize the data and present all information more interactively.

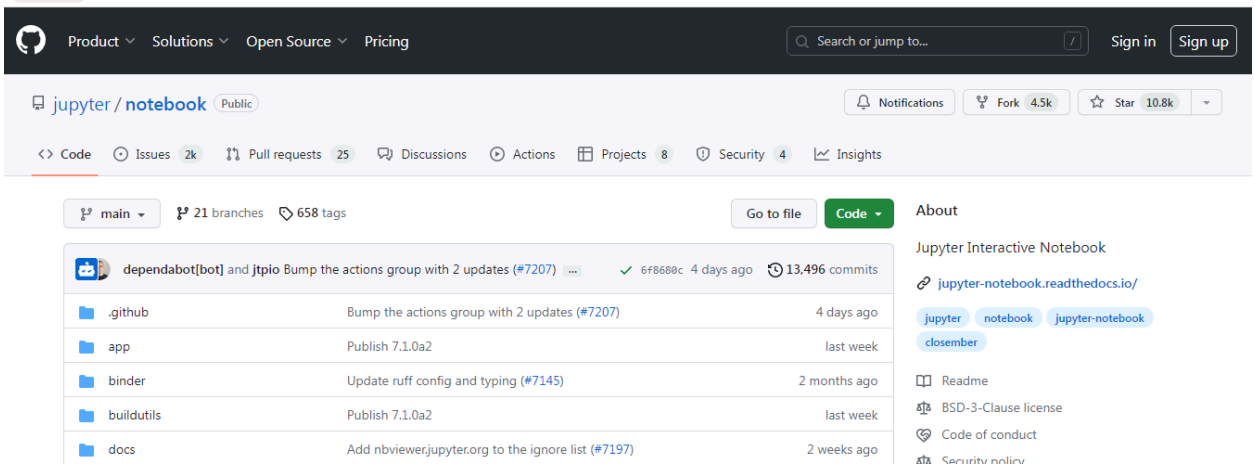
6. Sharing and Collaborating

Jupyter notebooks have great means of sharing code, insights and data analysis with others. The sharing of the Jupyter notebook helps in collaboration, documentation, and results dissemination. Therefore, for validating work, saving code or insights for future reference, and having easy access to work to a wider audience, Jupyter Notebook sharing is important. The Jupyter notebook sharing could be done in Python script, markdown, PDF or HTML format. These could be done This is the file option of sharing Jupyter Notebook wherein by clicking on file>download, the code could be downloaded in any format and the sharing of the downloaded document could be done.

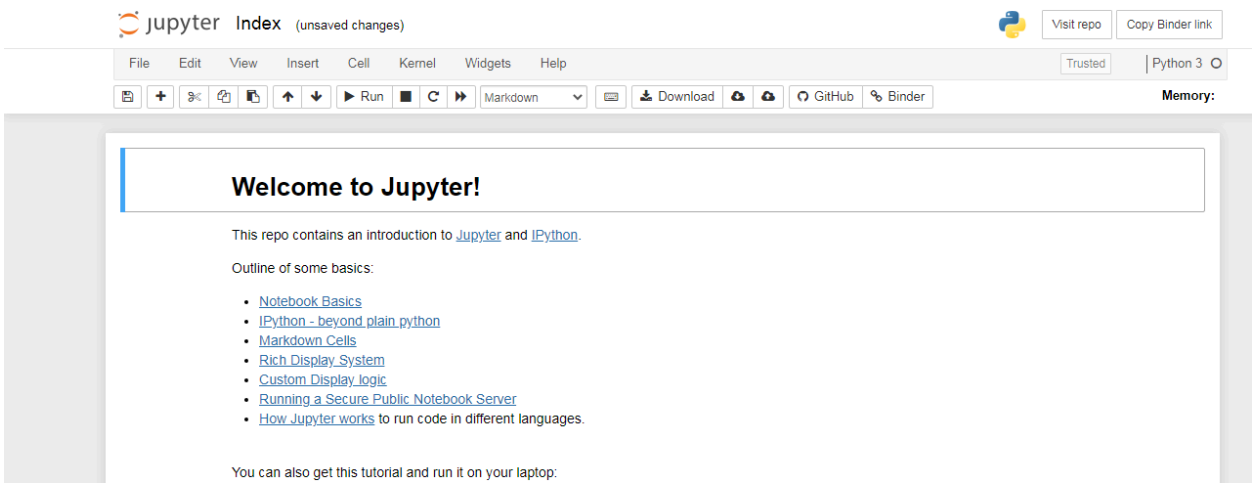


Unfortunately, this method is considered to be an ugly means of sharing due to its limitation of having an absence of related documents like datasets and even the need of the colleague, who is accessing the file, to set the environment from starting. Only after all the required packages are installed and configuration for the environment is derived, then the file becomes accessible. This results in making the process busy work for the audience.

To simplify this procedure, the second option of sharing is using the view option. There is the existence of GitHub repositories which enable the organization of static data notebooks and make the work accessible to other teammates.

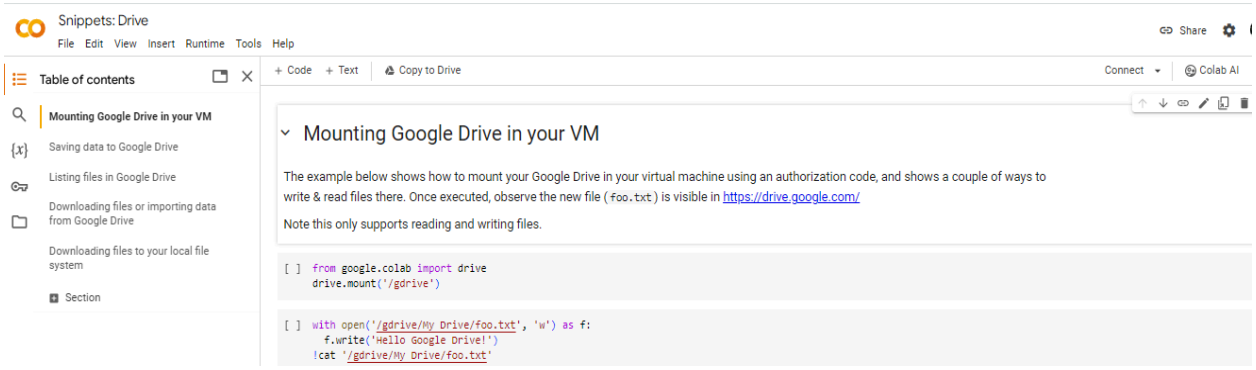


Also, another option is a Binder which helps in turning the work into interactive notebooks collection.



However, the problem with the GitHub method is that the work is static i.e. it can be viewed but not executed thus, options for collaboration, reproducing work, or commenting are not present. Even with Binder, after completing the allocated time of access, the URL for a notebook is not accessible making the work available for collaboration only for a short period.

The last option of sharing is cloud-based technology. The cloud-based technology helps in making the file fully executable with just a link. The environment provides quick reproduction and sharing projects but the ability to share notebooks doesn't mean collaboration is possible. For example – with Google Colab, the file can be shared but multiple people can't at the same time edit the notebook or leave comments. There is a requirement for different execution environments.



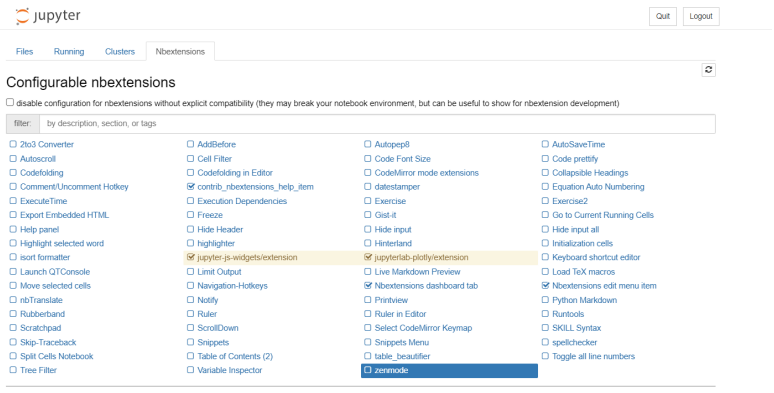
To overcome this, some of the platforms were developed which provide real-time collaboration i.e. Deepnote, Hex, Databricks notebook, Datacamp workspace, JetBrains Datalore, CoCalc, Noteable, and Nextjournal.

7. Jupyter Notebook Extensions and Customization

The installation of Jupyter Notebook has basic features which enable coding and data analysis. However, the performance could be further improved for the notebook by installing some relevant extensions. The Jupyter Notebook extensions as serve as the plugins for adding new functionalities and features to the interface, therefore, for enhancing experience and customization, the installation of extensions is necessary. To install the extensions, the first step is to have installation of Nbextensions using the below code

```
In [1]: pip install jupyter_contrib_nbextensions && jupyter contrib nbextension install
```

This shows the presence of a new tab



The above figure represents all the extensions which can be installed with Jupyter Notebook. Among these the top 12 essential extensions are

- Code prettifier – The extension is used for modifying existing code and leaving undesirable whitespaces. This is an easy and quick way to have consistent styling and make the code look better.
- Collapsible heading – This extension modifies the way of displaying headers thus allowing readers to expand and collapse. Once the extension is installed all the headings will be collapsible.
- Notify – The extension helps in solving the issue by sending the browser notification when a cell has completed running.
- Snippets – The extension helps by providing a handy menu for quickly importing snippets of lengthy code directly into code. Snippets help in the customization of code by reducing the unneeded submenus for reducing menu size and having code snippets addition.
- Tree filter – The extension provides a quick search tool for finding files in the File tab
- Autopep8 – The extension is similar to code prettifier which is useful for modifying the code and making the style of code loot consistent and better
- Execute time – The extension enables displaying duration and start time while executing a cell. It is majorly useful for the cells which take time to process
- Highlighter – The extension contributes to highlighting the selected word or all the matching words of the document
- Scratchpad – This extension allows code to run against the current kernel without having any modification of code.
- Move selected cell – The extension helps in improving efficiency by supporting the usage of some keyword shortcuts like Alt+up or Alt+down for moving the cell.
- Tabnine – The extension contributes to using AI for making accurate, fast, or smart auto-completion for giving an entire list of possibilities.
- Codefolding – The extension helps in folding code snippets which enable reading large code sections.

Thus, extensions should be installed with Jupyter Notebook to gain comfort and efficiency.

8. Common Issues and Troubleshooting

Jupyter fails to start

The common error occurs while installing Jupyter with the command prompt. Herein pip fails to install jupter showing older version. To resolve this check the pip version and update it to the latest version. The command for checking the version is *python -m pip --version* while for the update is *pip install -- upgrade pip*. Doing this *pip* is updated to successfully install Jupyter Notebook.

There is the possibility of failure in the installation of Jupyter due to the non-suitability of a system for installation. Ensure to have the latest version of Python and *pip* for better installation and Windows of 64-bit as Jupyter dependencies don't function appropriately with 32-bit.

Kernel not connecting

The kernel could fail in connecting or re-starting continuously. To resolve it either the kernel could be restarted by clicking on kernel>restart or kernel> restart and clear output. In case the issue persists, the entire Jupyter Notebook server needs to be started.

Jupyter Notebook not starting

The Jupyter Notebook fails to connect or launch in the browser. The issue can be resolved by checking whether Jupyter Notebook is running or not. Also, the browser cache could be cleared or try a different browser. If these don't work, check the specified port used and choose a different one by commanding *jupyter notebook –port=8889*.

Execution of cell takes time

The cell execution is slow or not completed. The issue can be resolved by optimization of code for better performance, profiling the code to identify bottlenecks, or checking for unintended recursion or infinite loops.

Missing modules or libraries

The *importerror* could be seen for modules or libraries. It could be resolved by ensuring the library is installed in the Jupyter environment or using *it !conda install* or *!pip install* to install the missing packages.

Plotting issues

Sometimes with matplotlib or other visualization libraries, the plots are not displayed. To resolve this issue, *%matplotlib inline* could be used in starting of the notebook for displaying the plots. Also, ensure that all the plotting libraries are imported and check for error messages before running the plotting code.

9. Summary

Jupyter Notebook is an open-source web-based interactive application which can be used for sharing or creating documents which have live text, code, visualizations, and equations.

Jupyter supports core languages i.e. Julia, Python and R

To install the notebook, there are two ways – one is using CPython which is Python's reference version and another is using Anaconda.

The dashboard of Jupyter shows three tabs i.e. files tab, the running tab, and the clusters tab.

The Jupyter Notebook consists of a File menu, Edit menu, View menu, Insert menu, Cell menu, Kernel menu, Widget menu, and Help menu.

The Jupyter Notebook interface has markdown cells which help in displaying text using markdown language.

Magic command is simply the group of special commands which are used for helping with non-coding tasks. Some of the built-in magic commands are %alias, %automagic, %code_wrap, %debug, or %conda.

Data visualization helps in pattern recognition, data analysis, decision-making, memory retention, and efficiency of results. The data visualization could be done in the form of a bar chart, pie chart, line chart, histogram, or scatter plot. The most common libraries used for visualization are matplotlib, seaborn, and Plotly.

The Jupyter Notebook could be shared and collaborated using a file menu, GitHub or Binder, and Collab networks.

The Jupyter Notebook extensions serve as the plugins for adding new functionalities and features to the interface. Among the existing extensions, 12 are the most popular.

Chapter: 8 Data Exploration

Objectives

To discuss the data exploration process in Python

- Understand the concept of data exploration
- Discussing the different types of data structures

0. Introduction

Data science at its core is mainly about extracting information from the data. This data can be derived from different sources like sensors, social media, reports, or transactions. To draw meaningful information, there is a need to explore the data by processing the datasets for findings relationships and patterns. Exploring can help in understanding the data in a better way, therefore highlighting the need to understand the data exploration concept and its relevance. This chapter focuses on providing an overview of data exploration along with discussing the data structures.

1. Basic

1.1. Overview

Data exploration is said to be the main aspect of model building of data analysis. Without spending too much time on data understanding and identifying its patterns, an effective predictive model cannot be developed. Data exploration enables the coder to reduce the major amount of time spent in understanding the data by providing information about key aspects of data processing and cleaning i.e. data loading, basic information, statistical summary, handling missing values, data visualization, outlier detection, correlation analysis, categorical variable analysis, feature engineering, or documentation.

1.2. Importance

Data exploration is an essential data analysis process which provides valuable insight and a foundation for making the decision. Some of the major reasons for conducting data exploration are

- The data exploration helps in providing deeper knowledge of data like its distribution, characteristics or structure. Thus, data exploration helps in understanding data for drawing accurate conclusions and having meaningful interpretations.
- Using statistical analysis and visualization, data exploration helps in identifying trends and patterns to recognize potential features and variables for modelling or analysis.
- The data exploration helps in the identification of missing data and choosing appropriate strategies like removal or imputation for maintaining the quality of data
- The data exploration enables the identification of anomalies, inconsistencies or outliers of data which are impacting the analysis. This enables raising findings' reliability
- Lastly, data visualization helps in the communication of information to both non-technical and technical audiences resulting in the simplification of complex data and making it easier to convey information.

1.3. Concepts of data exploration

Data exploration consists of many important concepts which are essential for understanding dataset characteristics and gaining information. These concepts are

- Data loading – Various formats i.e. JSON, CSV, TXT, or XLS could be used for loading data from different sources using pandas libraries.

Functions	Details
read_table	Reading delimited data from the file using tab ('\t') delimiter
read_csv	Reading delimited data from the file using tab (',') delimiter
read_fwf	Reading data having fixed width format for column
read_excel	Reading data from an Excel file
read_clipboard	Reading data from the clipboard. The function helps in converting web page tables

- Converting variables to different data types-converting data type of a variable to another data type is an important procedure for loading data. This could be conversion from numerical to string and vice versa or character to date.

Functions	Details
str(numeric)	Converting numeric value to the string
int(string)	Converting string value to integer
float(string)	Converting string value to integer
datetime.strptime(character, '%b %d %Y %I:%M%p')	Converting character to date

- Dataset or dataframe transpose - *using dataframe.pivot*, transpose of a table could be done
- Sorting pandas dataframe – the data can be sorted using *dataframe.sort().* The order of sorting i.e. ascending or descending could be stated like *df.sort(['Product'], ascending =[True])*
- Creation of plots – Using libraries like *matplotlib, seaborn,* or *Plotly*, the visualization of the data could be done.
- Frequency tables with pandas – The frequency tables can be created for categorical variables to understand their distribution. The dataframe.groupby is used for computing frequency. Suppose a given dataset is imported i.e. *iris*

```
In [1]:  import pandas as pd
         import seaborn as sns
executed in 1.69s, finished 21:02:20 2024-01-06

In [3]:  data = sns.load_dataset('iris')
         data
executed in 77ms, finished 21:02:49 2024-01-06
```

```
In [4]:  group= data.groupby(['species'])
         group.size()
executed in 18ms, finished 21:03:39 2024-01-06

Out[4]: species
        setosa        50
        versicolor    50
        virginica     50
        dtype: int64
```

Now, *data.groupby* could be used for computing frequency based on species.

- Select a sample of the dataset – The sampling of data could be done to understand data more quickly using *random* and *Numpy* libraries.
- Remove duplicate values – For removing duplicate observations, use *dataframe.drop_duplicates()*. For the same *irirs* dataset, when the remove duplicate values function was applied, the result is

```
In [12]:  dup = data.drop_duplicates(['sepal_width'])
          dup
executed in 33ms, finished 21:13:38 2024-01-06
```

Out[12]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
5	5.4	3.9	1.7	0.4	setosa
6	4.6	3.4	1.4	0.3	setosa
8	4.4	2.9	1.4	0.2	setosa
10	5.4	3.7	1.5	0.2	setosa
14	5.8	4.0	1.2	0.2	setosa

- Grouping values in Pandas – For understanding the sum, count, and average of a variable using dataframe.describe() and groupby() function. The describe function for the *iris* dataset could be applied in this form.

```
In [14]: summ= data.groupby(['species'])
         summ.describe()
```
executed in 93ms, finished 21:16:47 2024-01-06

Out[14]:

	sepal_length								sepal_width		...	petal_length		petal_width						
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	mean	std	min	25%	50%	75%	max
species																				
setosa	50.0	5.006	0.352490	4.3	4.800	5.0	5.2	5.8	50.0	3.428	...	1.575	1.9	50.0	0.246	0.105386	0.1	0.2	0.2	0.3	0.6
versicolor	50.0	5.936	0.516171	4.9	5.600	5.9	6.3	7.0	50.0	2.770	...	4.600	5.1	50.0	1.326	0.197753	1.0	1.2	1.3	1.5	1.8
virginica	50.0	6.588	0.635880	4.9	6.225	6.5	6.9	7.9	50.0	2.974	...	5.875	6.9	50.0	2.026	0.274650	1.4	1.8	2.0	2.3	2.5

3 rows × 32 columns

- Handling missing values – The missing values could be checked using dataframe.isnull().

```
In [15]: data.isnull()
```
executed in 15ms, finished 21:18:50 2024-01-06

Out[15]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	False	False	False	False	False
1	False	False	False	False	False
2	False	False	False	False	False
3	False	False	False	False	False
4	False	False	False	False	False
...	...	...	...	...	...
145	False	False	False	False	False
146	False	False	False	False	False
147	False	False	False	False	False
148	False	False	False	False	False
149	False	False	False	False	False

For treating missing values, the imputation method could be used. The imputation could help in detecting missing and outlier values. Herein, the mean function of Numpy library could be used for calculating the mean value (like meanvalue = np.mean(dataframe.variable))) and these mean values could be replaced with missing values with a command like dataframe.variable.fillna(meanvalue).

- Merging or joining datasets – The option could be used for integrating datasets from different sources. The command for the operation is pd.merge(df1, df2, how = 'inner', left_index = True, right_index = True). Herein, inner is how defines *inner join*, outer in it would define *outer join*, left will represent *left join* and right will show *right join*.

2. Data structures

Python consists of different data structures which help in manipulating and organising data efficiently. The data structures are the basics of the programming language around which the entire program is developed. Therefore, the knowledge of all data structures of Python is essential.

2.1. Data types

The variables in Python can store different data types i.e. *text, numeric, sequence, mapping, set, Boolean, binary*, and *none*. Among these data types – *int, float* and *complex* are numeric data types while other types like *str, bool,* or *none* type are regarded as non-numeric data types. A detailed explanation of each of the data types and their working is already discussed in Chapter 2.

2.2. Data containers

The storage of many values sometimes is done in a single variable. This single variable is called collection or container. These containers are used for holding an arbitrary number of other objects. There are three types of data containers i.e. *list, dictionary*, and *tuple*. A *list* is ordered and mutable, a *dictionary* is unordered and mutable, and a *tuple* is an ordered and immutable collection of objects. The reach of these containers and working with them is already discussed in Chapter 5.

2.3. Stacks

The stack in the data structure is the means of object arranging over another. It works similarly as the means of memory allocation in data structure. So, stack data structure allows operations only at the top of the stack. Thus, we can remove or add the element only at this end of the stack. In the stack, the last inserted element will come at the top and as we can remove data only from the stack top, thus, this feature is called Last in First out (LIFO). This operation of removing and adding elements is called POP and PUSH. The addition or removal of the element could be done using remove() and add() functions. For an empty list, the removal or addition of data elements can be done using pop() and append() methods. This working could be seen with the below code wherein, an empty stack was created and two values were added to it. Finally, using pop, an element from the top of the stack was removed.

```
In [1]: stack = []
        executed in 3ms, finished 22:00:58 2024-01-06

In [2]: stack.append('divine')
        stack.append('ed-sheeran')
        executed in 5ms, finished 22:02:18 2024-01-06

In [3]: stack.pop()
        executed in 12ms, finished 22:02:24 2024-01-06

Out[3]: 'ed-sheeran'
```

2.4. Sets

The set is defined as the unordered data collection which is mutable and does not permit the inclusion of duplicate elements. Though the element of the set is not mutable but set as a whole is mutable. Also, no index is attached to the element of the set. The set() function is used for creating a set or placing new elements within the curly braces pair. The addition, deletion, or union operations could be applied on sets. The working with sets can be seen in the below code

```
In [1]: names = set(['Joseph', 'Avinash', 'Marie', 'Archie', 'Cherry'])
        na = {'Richie', 'Nirvaan'}
        date = {20, 19, 4, 9}
        executed in 4ms, finished 21:41:51 2024-01-06

In [2]: print(names)
        print(na)
        print(date)
        executed in 8ms, finished 21:41:51 2024-01-06

        {'Archie', 'Joseph', 'Marie', 'Avinash', 'Cherry'}
        {'Nirvaan', 'Richie'}
        {9, 19, 20, 4}

In [3]: for n in na:
            print(n)
        executed in 5ms, finished 21:42:16 2024-01-06

        Nirvaan
        Richie

In [4]: na.add("Tanish")
        executed in 5ms, finished 21:42:32 2024-01-06

In [5]: na
        executed in 15ms, finished 21:42:37 2024-01-06

Out[5]: {'Nirvaan', 'Richie', 'Tanish'}
```

```
In [6]: na.discard("Richie")
        na
        executed in 6ms, finished 21:42:57 2024-01-06

Out[6]: {'Nirvaan', 'Tanish'}

In [7]: totalnames = names|na
        totalnames
        executed in 7ms, finished 21:43:28 2024-01-06

Out[7]: {'Archie', 'Avinash', 'Cherry', 'Joseph', 'Marie', 'Nirvaan', 'Tanish'}
```

The above code shows that 3 sets were created, each value from the set was printed, a new element 'Tanish' was added to the set *na*, another element 'Richie' was deleted from the set *na*, and finally union of the two sets i.e. *names* and *na* was done.

2.5. Binary search trees

A binary search tree is a data structure type which resembles a tree. Each tree node consists of left and right nodes at most. Herein left subtree of the parent node has a child node with a value lower than the parent node while the right subtree of the parent node has a child node higher than the parent node. Also, no duplicate values are included in the binary search tree. A sample for the binary search tree is presented below

```
In [9]: class Node:
            def __init__(self, key):
                self.left = None
                self.right = None
                self.val = key

        # Example BST
        bst = Node(5)
        bst.left = Node(2)
        bst.right = Node(7)
```

executed in 34ms, finished 22:35:34 2024-01-06

In the above tree, the main nodes are 5 with a left node value of 2 and a right node of 7 i.e. 2<5 and 7>5.

2.6. Sequences

In Python, sequence is another type of data structure which is a generic term for stating the ordered set. This means that the items entered in the set will remain the same when we access it. There are 6 types of sequences i.e. *strings, lists, tuples, bytes sequences, byte arrays,* and *range* objects. The strings consist of characters grouped inside double or single quotes, lists enable the creation of heterogenous items collection, tuples are Python objects sequence which has commas as separating items, byte() is used for retuning immutable byte sequence, byte arrays are mutable bytes sequence and range() is used for returning range object i.e. integers from specified start and end point. Applying all sequence types, the sample code is prepared.

```
In [1]: name = "Joseph"
        type (name)
executed in 6ms, finished 22:55:59 2024-01-06
Out[1]: str

In [2]: lst = [10, 20, 30]
        print(type(lst))
executed in 10ms, finished 22:55:59 2024-01-06
<class 'list'>

In [3]: tup =('jolie', 3.5, 1)
        print(type(tup))
executed in 5ms, finished 22:55:59 2024-01-06
<class 'tuple'>

In [4]: size = 2
        a = bytes(size)
        print(a)
executed in 4ms, finished 22:55:59 2024-01-06
b'\x00\x00'

In [5]: print(bytearray(3))
executed in 8ms, finished 22:55:59 2024-01-06
bytearray(b'\x00\x00\x00')

In [6]: b = range(3)
        print(type(b))
executed in 4ms, finished 22:57:34 2024-01-06
<class 'range'>
```

Thus, knowledge of all data structures is essential for problem-solving, algorithm design and data manipulation.

3. Summary

Data exploration enables the coder to reduce the major amount of time spent in understanding the data by providing information about key aspects of data processing and cleaning i.e. data loading, basic information, or statistical summary.

Important concepts of data exploration are data loading, converting variables into different data types, handling missing data, and so on

Python consists of different data structures which help in manipulating and organising data efficiently.

The variables in Python can store different data types i.e. text, numeric, sequence, mapping, set, Boolean, binary, and none.

The storage of many values sometimes is done in a single variable. This single variable is called collection or container. There are three types of data containers i.e. list, dictionary, and tuple.

The stack in the data structure is the means of object arranging over another.

The set is defined as the unordered data collection which is mutable and does not permit the inclusion of duplicate elements.

A binary search tree is a data structure type which resembles a tree.

Sequence is another type of data structure which is a generic term for stating the ordered set.

Chapter: 9 Summarizing numerical data in pandas

Objectives

Learn how to calculate essential statistical measures such as mean, median, variance, and standard deviation using pandas.

0. Introduction to pandas

Pandas is a Python library used for working with data sets. Pandas is a powerful Python library used for analysing, cleaning, exploring, and manipulating data.. It provides data structures and functions needed to work with structured data seamlessly. This chapter focuses on summarizing numeric data using Pandas, an essential skill for data analysis.

Why use pandas?

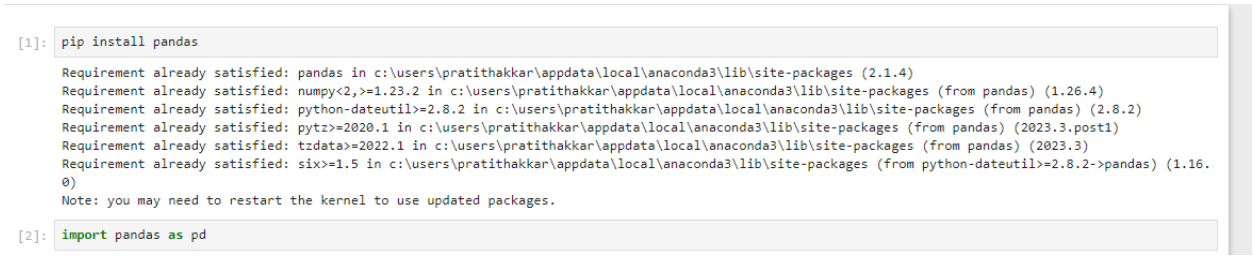
- Pandas allows us to analyse big data and make conclusions based on statistical theories.
- Pandas can clean messy data sets and make them readable and relevant.
- Relevant data is very important in data science.

Data structures available in python

- Series: 1D labelled homogeneously (similar type of data stored) typed array, like a list of numbers from one to 100.
- Data Frame: General 2D labelled, size-mutable tabular structure with potentially heterogeneously typed column.

Installing and loading data in pandas

- Installing and Loading data into pandas



```
[1]: pip install pandas

Requirement already satisfied: pandas in c:\users\pratithakkar\appdata\local\anaconda3\lib\site-packages (2.1.4)
Requirement already satisfied: numpy<2,>=1.23.2 in c:\users\pratithakkar\appdata\local\anaconda3\lib\site-packages (from pandas) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\pratithakkar\appdata\local\anaconda3\lib\site-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in c:\users\pratithakkar\appdata\local\anaconda3\lib\site-packages (from pandas) (2023.3.post1)
Requirement already satisfied: tzdata>=2022.1 in c:\users\pratithakkar\appdata\local\anaconda3\lib\site-packages (from pandas) (2023.3)
Requirement already satisfied: six>=1.5 in c:\users\pratithakkar\appdata\local\anaconda3\lib\site-packages (from python-dateutil>=2.8.2->pandas) (1.16.
0)
Note: you may need to restart the kernel to use updated packages.

[2]: import pandas as pd
```

- Loading data using pandas - Data can be loaded into Pandas Data Frame from various file formats like CSV, Excel, SQL databases, and more:
    - pd.read_csv('file.csv'): Reads data from a CSV file.
    - pd.read_excel('file.xlsx'): Reads data from an Excel file.

1. Basic Statistics

Pandas provides several functions to compute the basic statistics on numerical data. Some of the functionalities available in pandas are as follows:

- mean(): Calculates the average of the data.
- median(): Finds the middle value in the data.
- mode(): Identifies the most frequent value(s).
- min(): Finds the minimum value.
- max(): Finds the maximum value.
- std(): Computes the standard deviation.
- var(): Computes the variance.
- sum(): Calculates the sum of the values.

Input Code and Output :-

```
[44]: data = {
          'A': [10, 15, 10, 20, 15, 10, 25, 30, 15, 10]
      }

      df = pd.DataFrame(data)

      mean_A = df['A'].mean()
      median_A = df['A'].median()
      mode_A = df['A'].mode()[0]
      min_A = df['A'].min()
      max_A = df['A'].max()
      std_A = df['A'].std()
      var_A = df['A'].var()
      sum_A = df['A'].sum()

      print("Column A - Basic Statistics")
      print(f"Mean: {mean_A}")
      print(f"Median: {median_A}")
      print(f"Mode: {mode_A}")
      print(f"Min: {min_A}")
      print(f"Max: {max_A}")
      print(f"Standard Deviation: {std_A}")
      print(f"Variance: {var_A}")
      print(f"Sum: {sum_A}")
```

```
Column A - Basic Statistics
Mean: 16.0
Median: 15.0
Mode: 10
Min: 10
Max: 30
Standard Deviation: 6.992058987801O1
Variance: 48.888888888888886
Sum: 160
```

2. Describe Statistics

Summarization includes counting, describing all the data present in data frame. We can summarize the data present in the data frame using describe() method. This method is used to get min, max, sum, count values from the data frame along with data types of that particular column.

- describe(): This method elaborates the type of data and its attributes, It provides a quick overview of the numeric columns in the DataFrame.

Input Code and Output:-

```
[45]: df.describe()
```

[45]:

	A
count	10.000000
mean	16.000000
std	6.992059
min	10.000000
25%	10.000000
50%	15.000000
75%	18.750000
max	30.000000

- unique() - This method is used to get all unique values from the given column.

Input Code and Output:-

```
[46]: df['A'].unique()
[46]: array([10, 15, 20, 25, 30], dtype=int64)
```

- nunique(): This method is similar to unique but it will return the count the unique values.

Input Code and Output:-

```
[48]: df['A'].unique()
[48]: array([10, 15, 20, 25, 30], dtype=int64)
```

3. Correlation and Covariance

Pandas can calculate the correlation and covariance between different columns in a DataFrame.

- corr(): Computes pairwise correlation of columns, excluding NA/null values. The values range from -1 to 1, where 1 indicates a perfect positive correlation, -1 indicates a perfect negative correlation, and 0 indicates no correlation.
- cov(): Computes pairwise covariance of columns, excluding NA/null values. Covariance is a measure of how much two random variables vary together. Higher values indicate that the variables increase together, while lower (negative) values indicate that one variable increases as the other decreases.

Input Code and Output:-

```python
# Create a sample DataFrame
data = {
    'A': [1, 2, 3, 4, 5],
    'B': [5, 4, 3, 2, 1],
    'C': [1, 1, 1, 1, 1]
}

df = pd.DataFrame(data)

# Calculate correlation
corr_matrix = df.corr()
print("Correlation Matrix:")
print(corr_matrix)
print()

# Calculate covariance
cov_matrix = df.cov()
print("Covariance Matrix:")
print(cov_matrix)
```

```
Correlation Matrix:
     A    B   C
A  1.0 -1.0 NaN
B -1.0  1.0 NaN
C  NaN  NaN NaN

Covariance Matrix:
     A    B    C
A  2.5 -2.5  0.0
B -2.5  2.5  0.0
C  0.0  0.0  0.0
```

4. Grouping

It is used to group one or more columns in a dataframe by using the groupby() method.

Grouping data by certain criteria and applying aggregate functions is a common task in data analysis. Groupby mainly refers to a process involving one or more of the following steps they are:

- Splitting: It is a process in which we split data into group by applying some conditions on datasets.
- Applying: It is a process in which we apply a function to each group independently
- Combining: It is a process in which we combine different datasets after applying groupby and results in a data structure

All the Basic Statistics functions mentioned previously can be used while using the fun groupby().

Functions involved in grouping the data:

- groupby(): Groups data by specified columns.
- agg(): Applies aggregate functions like mean, sum, min, max, etc., on grouped data.

Input Code and Output –

```
grouped_without_agg = df.groupby('City')

print("Grouped without agg:")
for city, group in grouped_without_agg:
    print(f"City: {city}")
    print(group)
    print()
```

```
Grouped without agg:
City: Chicago
      City  Temperature  Humidity
1  Chicago           65        55
4  Chicago           68        52

City: New York
       City  Temperature  Humidity
0  New York           70        50
3  New York           72        45

City: San Francisco
            City  Temperature  Humidity
2  San Francisco           75        60
```

Example without using agg()

This applies to all the columns for specifying different functions for different columns you will have to use the agg method.

```
data = {
    'City': ['New York', 'Chicago', 'San Francisco', 'New York', 'Chicago'],
    'Temperature': [70, 65, 75, 72, 68],
    'Humidity': [50, 55, 60, 45, 52]
}

df = pd.DataFrame(data)

# Group by 'City' and calculate average temperature and humidity
grouped = df.groupby('City').agg({
    'Temperature': 'mean',
    'Humidity': 'mean'
}).reset_index()

print("Grouped with agg:")
print(grouped)
```

```
Grouped with agg:
            City  Temperature  Humidity
0        Chicago         66.5      53.5
1       New York         71.0      47.5
2  San Francisco         75.0      60.0
```

Example using agg()

Other method of grouping the data is using pivot table. Pivot tables are used to summarize data with multi-dimensional grouping.

- pivot_table(): Creates a pivot table from a DataFrame.

Input Code and Output:-

[58]:
```python
data = {
    'Date': ['2024-01-01', '2024-01-01', '2024-01-02', '2024-01-02', '2024-01-01'],
    'City': ['New York', 'Chicago', 'New York', 'Chicago', 'New York'],
    'Temperature': [32, 30, 28, 29, 30],
    'Humidity': [60, 65, 55, 58, 62]
}

df = pd.DataFrame(data)

# Create a pivot table to summarize average temperature and humidity by city and date
pivot_table = df.pivot_table(index='City', columns='Date', values=['Temperature', 'Humidity'], aggfunc='mean')

print("Pivot Table:")
print(pivot_table)
```

```
Pivot Table:
          Humidity            Temperature
Date    2024-01-01 2024-01-02  2024-01-01 2024-01-02
City
Chicago       65.0       58.0        30.0       29.0
New York      61.0       55.0        31.0       28.0
```

[ ]:

Chapter: 10 Summarizing categorical data in pandas

Objectives

Explore methods to summarize categorical data using pandas, including frequency counts, cross-tabulations, and summary statistics.

0. What is Categorical Data?

Categorical data represents types or categories of data, often with a limited number of possible values. These categories can be either ordered (ordinal) or unordered (nominal). Summarizing categorical data is essential for understanding the distribution and frequency of the categories within the data.

Categorical are a pandas data type corresponding to categorical variables in statistics. A categorical variable takes on a limited, and usually fixed, number of possible value. Examples

are gender, social class, blood type, country affiliation, observation time or rating via Likert scales.

All values of categorical data are either in categories or np.nan. Order is defined by the order of categories, not lexical order of the values. Internally, the data structure consists of a categories array and an integer array of codes which point to the real value in the categories array.

1. Need to Summarize Categorical Data?

- Understanding Distribution: Helps in identifying the frequency and proportion of each category.
- Detecting Anomalies: Highlights unusual or unexpected distributions.
- Data Preparation: Assists in transforming and encoding categorical data for machine learning models.
- Data Structure for Categorical Data are of two types: - series and dataframe

2. Basic Summary Statistics for Categorical Data

- value_counts(): Returns a Series containing counts of unique values.
- count(): Returns the number of non-null observations in the DataFrame or Series.

Input Code –

```
data = {
    'Category': ['A', 'B', 'A', 'C', 'B', 'A', 'B', 'C', 'C', 'A'],
    'Values': [10, 20, 10, 30, 20, 10, 20, 30, 30, 10]
}

df = pd.DataFrame(data)
```

```
category_counts = df['Category'].value_counts()
print(category_counts)
```

Output:-

```
Category
A    4
B    3
C    3
Name: count, dtype: int64
```

Input Code –

```
category_count = df['Category'].count()
print(category_count)
```

Output:-

```
print(category_count)

10
```

3. Descriptive Statistics for Categorical Data

- describe(): Provides a summary of the categorical columns.
- unique(): Returns the unique values in a column.
- nunique(): Returns the number of unique values in a column.

Input Code –

```
category_description = df['Category'].describe()
print(category_description)
```

Output:-

```
count     10
unique     3
top        A
freq       4
Name: Category, dtype: object
```

Input Code –

```
unique_categories = df['Category'].unique()
print(unique_categories)
```

Output:-

```
['A' 'B' 'C']
```

Input Code –

```
num_unique_categories = df['Category'].nunique()
print(num_unique_categories)
```

Output:-

```
3
```

4. Grouping and Aggregating Categorical Data

Grouping and aggregating categorical data in pandas involves using the groupby() method to split the data into groups based on one or more categorical columns. After splitting, you can apply aggregate functions like mean, sum, min, max, etc., to each group. This process helps in summarizing and understanding the data distribution across different categories.

- groupby(): Groups data by specified columns.
- agg(): Applies aggregate functions like mean, sum, min, max, etc., on grouped data.

Input Code –

```
grouped_mean = df.groupby('Category')['Values'].mean()
print(grouped_mean)
```

```
grouped_agg = df.groupby('Category')['Values'].agg(['mean', 'sum', 'min', 'max'])
print(grouped_agg)
```

Output:-

```
print(grouped_mean)

Category
A    10.0
B    20.0
C    30.0
Name: Values, dtype: float64
```

```
print(grouped_agg)
```

Category	mean	sum	min	max
A	10.0	40	10	10
B	20.0	60	20	20
C	30.0	90	30	30

5. Cross-Tabulation and Pivot Tables

Cross-tabulation and pivot tables are powerful tools for summarizing categorical data. They allow you to explore the relationships between different categorical variables and compute aggregates across multiple dimensions.

- crosstab(): Computes a simple cross-tabulation of two (or more) factors. It is particularly useful for counting the occurrences of combinations of categories.
- pivot_table(): Creates a pivot table, which can summarize data with multi-dimensional grouping and compute various aggregation functions.

Input Code –

```
data = {
    'Category1': ['A', 'B', 'A', 'C', 'B', 'A', 'B', 'C', 'C', 'A'],
    'Category2': ['X', 'Y', 'X', 'Y', 'Y', 'X', 'X', 'X', 'Y', 'Y'],
    'Values': [10, 20, 10, 30, 20, 10, 20, 30, 30, 10]
}

df = pd.DataFrame(data)
```

```
cross_tab = pd.crosstab(df['Category1'], df['Category2'])
print(cross_tab)
```

Output:-

```
Category2  X  Y
Category1
A          3  1
B          1  2
C          1  2
```

Input Code –

```
pivot_tbl = df.pivot_table(index='Category1', columns='Category2', values='Values', aggfunc='mean')
print(pivot_tbl)
```

Output:-

```
Category2     X     Y
Category1
A          10.0  10.0
B          20.0  20.0
C          30.0  30.0
```

6. Groupwise Summary of Mixed Data in Pandas

A groupwise summary of mixed data involves grouping data based on one or more categorical columns and then computing aggregate statistics for both categorical and numerical columns within each group. This provides a comprehensive overview of the data, highlighting patterns and relationships between different variables.

Key Functions and methods used include:-

- groupby(): Splits the data into groups based on specified columns.
- agg(): Applies one or more aggregate functions to the grouped data, which can handle both numerical and categorical columns.

*Codes for the same have been previously explained.

Chapter: 11 Visualization using Seaborn Package

Objectives

use of Seaborn to create expressive and insightful visualizations of data distributions, relationships, and patterns.

0. Introduction to Seaborn

Seaborn is a Python data visualization library based on Matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics. It simplifies the process of creating complex visualizations and integrates well with Pandas data structures.

Why use Seaborn?

- Beautiful default styles
- Built-in themes

- Integration with Pandas
- Statistical visualization

Installing and Loading Seaborn

To use Seaborn, you need to install it first (if not already installed) and then import it along with Matplotlib for additional customization.

Input Code –

```
# Installing
!pip install seaborn

# Importing
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

data = {
    'Category1': ['A', 'B', 'A', 'C', 'B', 'A', 'B', 'C', 'C', 'A'],
    'Category2': ['X', 'Y', 'X', 'Y', 'Y', 'X', 'X', 'X', 'Y', 'Y'],
    'Values': [10, 20, 10, 30, 20, 10, 20, 30, 30, 10],
    'Scores': [1, 2, 1, 3, 2, 1, 2, 3, 3, 1]
}

df = pd.DataFrame(data)
```

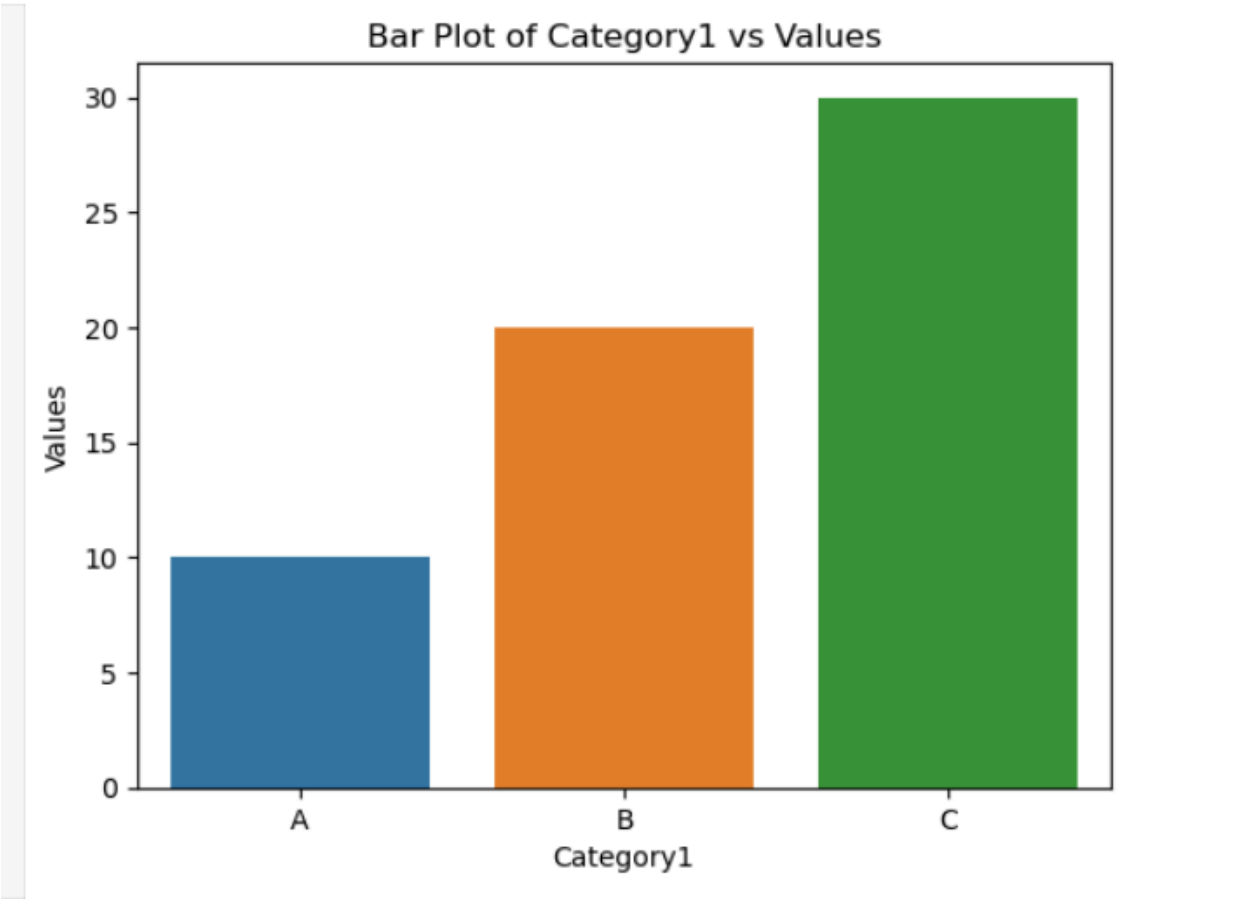
1. Plot Types

- **Bar Plot -** A bar plot displays the relationship between a categorical variable and a numerical variable. Bar plot represents an estimate of central tendency for a numeric variable with the height of each rectangle and provides some indication of the uncertainty around that estimate using error bars.

Input Code –

```
sns.barplot(x='Category1', y='Values', data=df)
plt.title('Bar Plot of Category1 vs Values')
plt.show()
```

Output:-

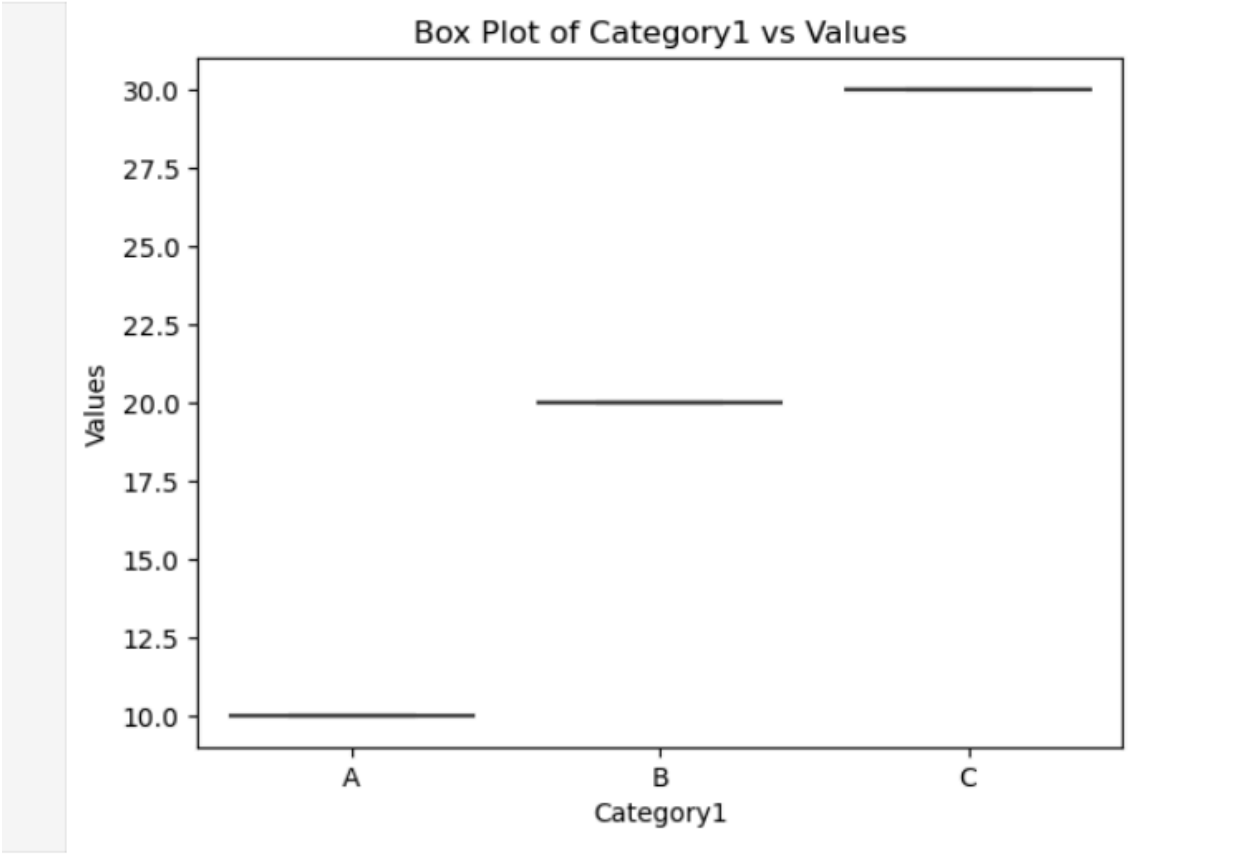


- **Box Plot -** A box plot (or box-and-whisker plot) s is the visual representation of the depicting groups of numerical data through their quartiles against continuous/categorical data. It shows the distribution of numerical data and can help identify outliers.

Input Code –

```
:  sns.boxplot(x='Category1', y='Values', data=df)
   plt.title('Box Plot of Category1 vs Values')
   plt.show()
```

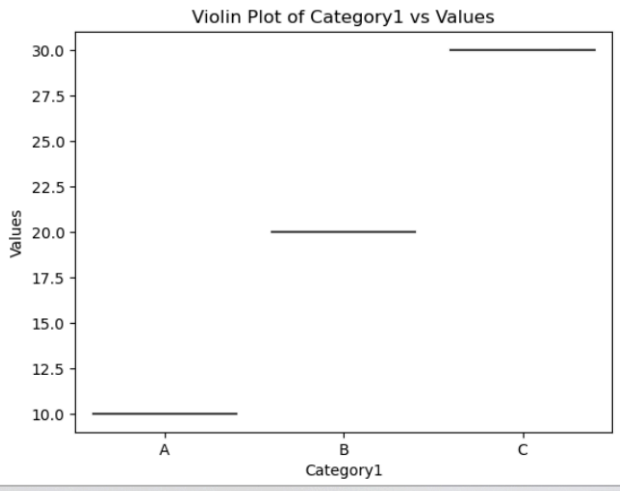
Output:-



- **Violin Plot -** A violin plot is similar to a boxplot. It shows several quantitative data across one or more categorical variables such that those distributions can be compared. It combines aspects of a box plot and a density plot. It shows the distribution of the data across different categories.

Input Code and Output –

```
[19]: sns.violinplot(x='Category1', y='Values', data=df)
      plt.title('Violin Plot of Category1 vs Values')
      plt.show()
```

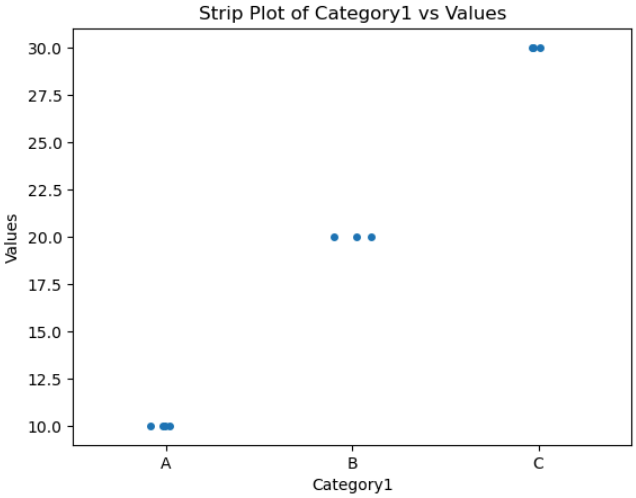


- **Strip Plot -** A strip plot is a single-axis scatter plot that is used to visualise the distribution of many individual one-dimensional values. It shows individual data points along a categorical axis.

Input Code and Output –

```
[20]: sns.stripplot(x='Category1', y='Values', data=df, jitter=True)
      plt.title('Strip Plot of Category1 vs Values')
      plt.show()
```

```
C:\Users\pratithakkar\AppData\Local\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_n
removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
C:\Users\pratithakkar\AppData\Local\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_n
removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```



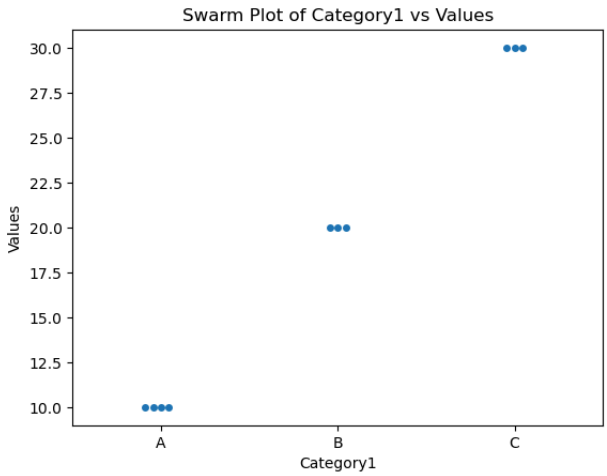

```
[ ]:
```

- **Swarm Plot -** A swarm plot is similar to a strip plot, We can draw a swarm plot with non-overlapping points against categorical data i.e., the points are adjusted to avoid overlap

Input Code and Output -

```
[21]: sns.swarmplot(x='Category1', y='Values', data=df)
      plt.title('Swarm Plot of Category1 vs Values')
      plt.show()
```

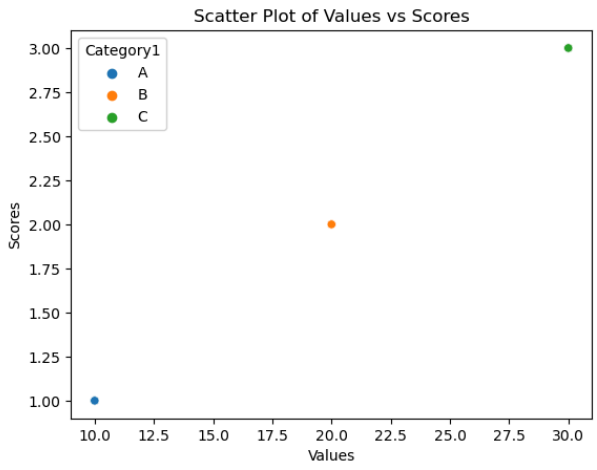
```
C:\Users\pratithakkar\AppData\Local\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and will be
removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
C:\Users\pratithakkar\AppData\Local\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and will be
removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```



- **Scatter Plot/ Pair Plot -** Scatterplot Can be used with several semantic groupings which can help to understand well in a graph against continuous/categorical data. It can draw a two-dimensional graph. It shows relationship between two variable.

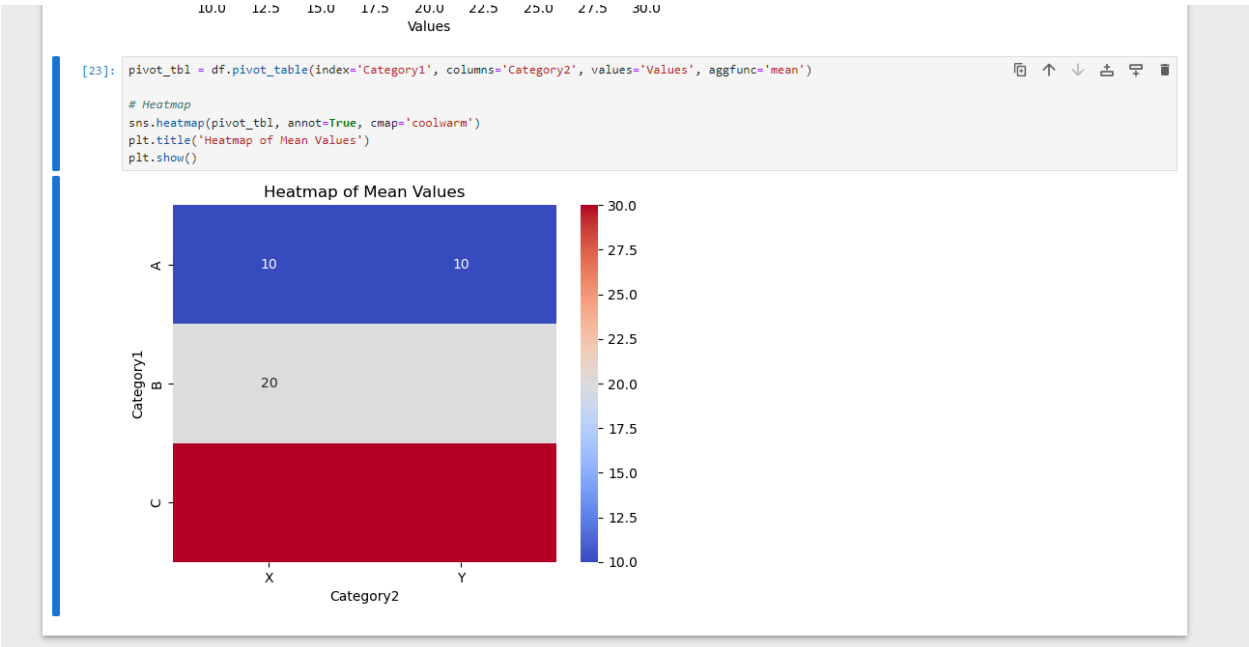
Input Code and Output –

```
[22]: sns.scatterplot(x='Values', y='Scores', hue='Category1', data=df)
      plt.title('Scatter Plot of Values vs Scores')
      plt.show()
```



- **Heatmap -** A heatmap (aka heat map) depicts values for a main variable of interest across two axis variables as a grid of colored squares. It displays data in a matrix form, with color indicating the magnitude of values.

Input Code and Output –

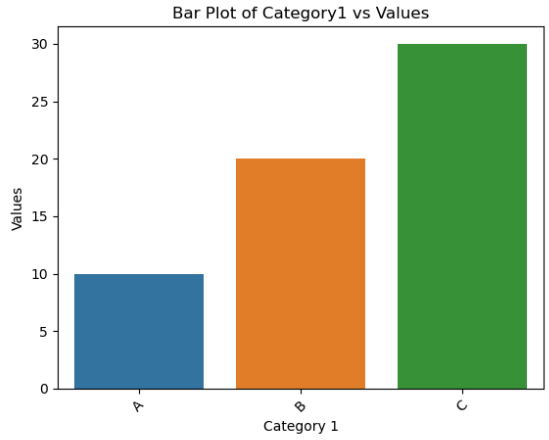


2. Customizing Plots

Customizing plots in Seaborn allows you to enhance the visual appeal and clarity of your visualizations. Seaborn provides various parameters and functions for customization, and you can also integrate Seaborn with Matplotlib to further refine your plots.

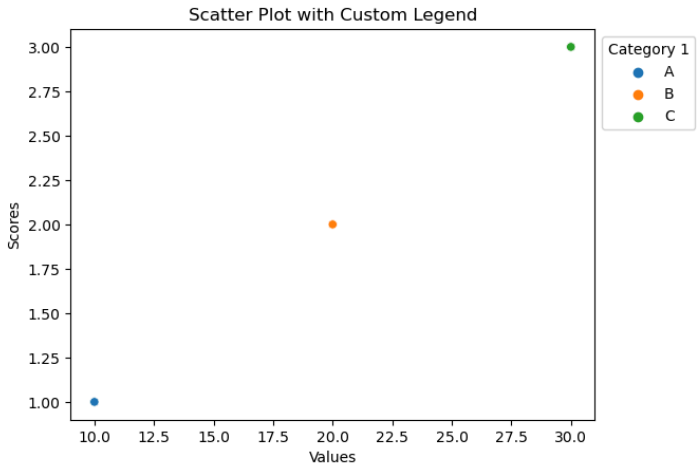
Input Code and Output –

```
[24]: sns.barplot(x='Category1', y='Values', data=df)
      plt.title('Bar Plot of Category1 vs Values')
      plt.xlabel('Category 1')
      plt.ylabel('Values')
      plt.xticks(rotation=45)  # Rotate x-axis labels
      plt.show()
```



Adding titles, axis labels, and modifying ticks.

[25]:
```
sns.scatterplot(x='Values', y='Scores', hue='Category1', data=df)
plt.title('Scatter Plot with Custom Legend')
plt.xlabel('Values')
plt.ylabel('Scores')
plt.legend(title='Category 1', loc='upper left', bbox_to_anchor=(1, 1))
plt.show()
```



Customizing legends

There are many more functionalities and customization that can be added using the Seaborn library.

Chapter: 12 Conditional Statement, control structures and functions

Objectives

Develop a comprehensive understanding of conditional statements (if-else, nested if-else) and control structures (loops, break, continue) in Python along with understanding functions

0. Conditional Statements

Conditional Statements are statements in Python that provide a choice for the control flow based on a condition. It means that the control flow of the Python program will be decided based on the outcome of the condition.

Conditional statements in Python allow you to execute certain pieces of code based on specific conditions. The primary conditional statements are if, elif, and else.

Types of conditional statements:

- If Conditional Statement - The if statement evaluates a condition and executes the code block if the condition is true.

Input Code and Output –

Category1

```
[30]: x = 10
      if x > 5:
          print("x is greater than 5")

      x is greater than 5
```

- If-Else Conditional Statement- The if-else statement provides an alternative code block to execute if the condition is false.

Input Code and Output

```
1]: x = 3
    if x > 5:
        print("x is greater than 5")
    else:
        print("x is not greater than 5")

    x is not greater than 5
```

- If-Elf-Else Conditional Statement - The if-elif-else statement allows multiple conditions to be evaluated in sequence.

Input Code and Output –

```
x is not greater than 5
```

```
[32]: x = 7
if x > 10:
    print("x is greater than 10")
elif x > 5:
    print("x is greater than 5 but less than or equal to 10")
else:
    print("x is 5 or less")
```

```
x is greater than 5 but less than or equal to 10
```

- Nested if-else condition - Nested if..else means an if-else statement inside another if statement. Or in simple words first, there is an outer if statement, and inside it another if – else statement is present and such type of statement is known as nested if statement.

Input Code and Output –

```
x is greater than 5 but less than or equal to 10
```

```
[33]: letter = "A"

if letter == "B":
    print("letter is B")

else:

    if letter == "C":
        print("letter is C")

    else:

        if letter == "A":
            print("letter is A")

        else:
            print("letter isn't A, B and C")
```

```
letter is A
```

```
[ ]:
```

- Ternary Expression Conditional Statements in Python - The Python ternary Expression determines if a condition is true or false and then returns the appropriate value in accordance with the result. The ternary Expression is useful in cases where we need to assign a value to a variable based on a simple condition, and we want to keep our code more concise — all in just one line of code. In simple words it is writing an if else condition in one line.

Input Code and Output –

```
: a, b = 10, 20

print ("Both a and b are equal" if a == b else "a is greater than b"
       if a > b else "b is greater than a")
```

```
b is greater than a
```

1. Loops and Control Structures

Loops allow you to execute a block of code repeatedly. Python provides for loops and while loops.

Types of Loops:

- For Loop - The for loop iterates over a sequence (like a list, tuple, or string) and executes the code block for each item.

Input Code and Output –

```
36]: numbers = [1, 2, 3, 4, 5]
     for num in numbers:
         print(num)

1
2
3
4
5
```

- While Loop - The while loop executes a block of code as long as the specified condition is true.

Input Code and Output –

```
7]: count = 0
    while count < 5:
        print(count)
        count += 1

0
1
2
3
4
```

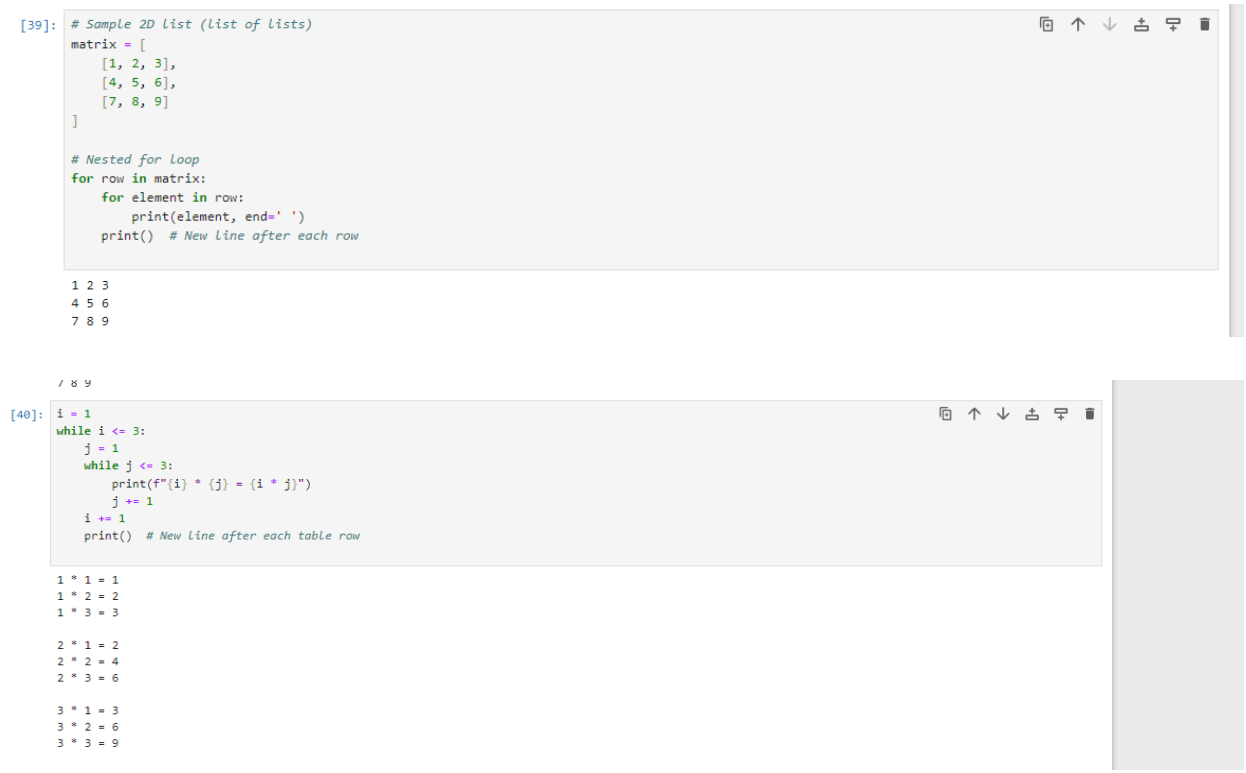
- Break and Continue: (Loop control statements)
    - break: Terminates the loop prematurely.
    - continue: Skips the current iteration and continues with the next iteration.

Input Code and Output –

```
[38]: for num in range(10):
          if num == 5:
              break  # Exit the loop when num is 5
          if num % 2 == 0:
              continue  # Skip even numbers
          print(num)

1
3
```

- Nested Loops (For and While): Nested loops are loops within loops. They are useful for iterating over multi-dimensional data structures like lists of lists, matrices, and more. In Python, you can use both for and while loops in nested configurations.

Input Code and Output –



```
[39]: # Sample 2D list (list of lists)
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Nested for loop
for row in matrix:
    for element in row:
        print(element, end=' ')
    print()  # New line after each row
```

```
1 2 3
4 5 6
7 8 9
```

```
[40]: i = 1
while i <= 3:
    j = 1
    while j <= 3:
        print(f"{i} * {j} = {i * j}")
        j += 1
    i += 1
    print()  # New line after each table row
```

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3

2 * 1 = 2
2 * 2 = 4
2 * 3 = 6

3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
```

2. Functions

Functions in Python are blocks of reusable code that perform a specific task. They can take inputs (parameters) and return outputs (results).

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

Defining a function: Use the def keyword to define a function.

Calling a function: To call a function, use the function name followed by parenthesis

Input Code and Output –

```
]: # Example of a simple function
def greet(name):
    return f"Hello, {name}!"

# Calling the function
print(greet("Alice"))
```

```
Hello, Alice!
```

Default Parameters - Functions can have default parameter values, which are used if no argument is provided.

Key Arguments: Functions can also accept keyword arguments, allowing you to specify parameter values by name.

Input Code and Output –

```
]: def describe_pet(pet_name, animal_type="dog"):
    print(f"I have a {animal_type} named {pet_name}.")

# Calling the function with positional and keyword arguments
describe_pet("Whiskers", "cat")
describe_pet(pet_name="Buddy", animal_type="hamster")
describe_pet(pet_name="Rover")
```

```
I have a cat named Whiskers.
I have a hamster named Buddy.
I have a dog named Rover.
```

Returning Multiple Values: Functions can return multiple values as a tuple. Functions can also return data in the form of dict or a dataframe.

```
[43]: # Function returning multiple values
def get_coordinates():
    x = 5
    y = 10
    return x, y

# Receiving multiple values
coord_x, coord_y = get_coordinates()
print(f"X: {coord_x}, Y: {coord_y}")
```

```
X: 5, Y: 10
```