Day -1 (https://rb.gy/07ojet)

Introduction :

Goals and Expectations:

- ☐ Understand **basic syntax/principles of Python programming.**
- ☐ Write **Python code to solve moderate complex problems**.
- ☐ Exposure to **real-world applications of Python.**

What is Programming ?

Just like we use Nepali or English to communicate with each other, we use a programming language like python to communicate with the computer.

(https://rb.gy/mqqcn4)

What is Python ?

Python is a simple and easy to understand language which feels like reading simple english. This pseudocode nature of python makes it easy to learn and understandable by beginners.

- Python is a high-level, interpreted programming language known for its simplicity and readability.
- Guido van Rossum created Python in the late 1980s, and it has since become one of the most popular languages among developers.
- Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming.

Features:

- Easy to understand
- Free and open source
- High level language
- Portable - works on mac/linux/windows
- Fun to work with

Applications:

- Web development (Django, Flask)
- Data science and machine learning (NumPy, Pandas, TensorFlow)
- Scientific computing (SciPy)
- Artificial intelligence and natural language processing (NLTK, SpaCy)
- Automation and scripting
- Game development (Py-game)
- Finance and trading
- Education and research
- Blockchain development

Installation

Python can be easily installed from python.org

- Check using **python –version** in CMD
- Any to every version works
- Use **python** in CMD to run python one line codes
- Use **exit()** to exit the python interpreter.

IDE: Integrated Development Environment

- Vscode, Pycharm, Vim (linux), Notepad++
- Vscode and Jupyter Notebook Extension

Virtual Environment in Python

- self-contained directory tree that contains its own Python installation
- can have its own set of Python packages/modules installed.
- create isolated environments for different projects, each with its own dependencies, without affecting the system-wide Python installation or other projects.
- Create using **python3 -m venv env_name** in CMD/terminal
- Activate the file: **myenv\Scripts\activate.ps1**
- Deactivate using: **deactivate**

Benefits:

- Isolated from the system-wide Python installation and other virtual environments. (prevents conflicts between different projects)
- Allow you to manage dependencies for each project separately. (You can install specific versions of packages without affecting other projects.)
- Easy to reproduce the development environment on different systems. (can share the project with others, and they can create the same environment using the requirements.txt file)

Solution to the problem: https://rb.gy/ezmj9v

PiP (Python Package Index):

- Package manager for Python, used for installing and managing Python packages
- Simplifies the process of installing third-party libraries and packages that are not included in the standard Python distribution.
- Check using **pip –version** in CMD
- Upgrade using **pip install --upgrade pip** in CMD

Modules:

- A file containing Python code, which can define functions, classes, and variables.
- Need to import it into your Python script using the **import** statement. Eg
  import math
  print(math.sqrt(16))

Package: A collection of modules organized in a directory structure.
Libraries:

- Collections of modules or packages that provide specific functionality for tasks such as data manipulation, web development, scientific computing, etc.
- Third-party libraries are developed by the Python community and provide additional functionality beyond what is available in the standard library.

Commonly Used Libraries:

- **NumPy**: For numerical computing and arrays.
- **Pandas**: For data manipulation and analysis.
- **Matplotlib**: For creating static, interactive, and animated visualizations.
- **Scikit-learn**: For machine learning algorithms and tools.
- **TensorFlow, PyTorch**: For deep learning and neural networks.
- **Flask, Django**: For web development.

Start from Basics

print("Hello World")

Execute this file(day1.py) by typing **python day1.py** and you will see "Hello World" printed on the screen.
Syntax for input and print function with example:

Day -2

```
# Input operation: Prompt the user to enter their name
name = input("Please enter your name: ")

# Output operation: Greet the user with their name
print("Hello, " + name + "! Welcome to the Python workshop.")
```

Comments :

Comments are used to write something which the programmer does not want to execute.

Types:

Single line comments -> Written using #comment

Multi line comments -> `Written using '''Comment'''`

Use **'Ctrl+?'** for commenting in VS Code

Variable, Data Types and Operators:

A variable is the name given to a memory location in a program. For example:

a = 30

b = "Harry"

c = 5.5

Bool = True

Variable : Container to store a value.

Keyword: Reserved words in python.

Data Types :

Variables can store different types of data. Some common data types -

1. Integer (**int**): Represents whole numbers, e.g., 10, -5, 1000.
2. Float (**float**): Represents floating-point numbers, i.e., numbers with a decimal point, e.g., 3.14, -0.5, 2.0.

3. String (**str**): Represents text, enclosed in single quotes (' ') or double quotes (" "), e.g., 'Hello', "Python", '123'.
4. Boolean (**bool**): Represents True or False values, used for logical operations and comparisons, e.g., True, False.

Operators:

Operators are symbols or keywords that perform operations on items.

1. **Arithmetic** Operators:
    - Addition (+),
    - Subtraction (-),
    - Multiplication (*),
    - Division (/),
    - Modulus (%),
    - Exponentiation (**)
2. **Comparison** Operators:
    - Equal to (==),
    - Not equal to (!=),
    - Greater than (>),
    - Less than (<),
    - Greater than or equal to (>=),
    - Less than or equal to (<=)
3. **Logical** Operators:
    - AND (and),
    - OR (or),
    - NOT (not).
4. **Assignment** Operators:
    - Assignment (=),
    - Addition assignment (+=),
    - Subtraction assignment (-=),
    - Similar arithmetic assignments

5. **Bitwise** Operators: (to work with binary bits)
    - AND (&),
    - OR (|),
    - XOR (^),
    - NOT (~),
    - Left shift (<<),
    - Right shift (>>).

Exception handling:

- allows you to handle errors or exceptional situations that may occur during the execution of a program
- prevents the program from crashing
- provides a mechanism to catch and handle exceptions using **try, except, else,** and **finally** blocks.

try block: The code that might raise an exception is placed within the try block.

except block: If an exception occurs within the try block, the control is transferred to the corresponding except block.

else block (optional): The else block is executed if no exceptions occur in the try block. It is typically used to perform actions that should only occur if no exceptions were raised.

finally block (optional): The finally block is executed regardless of whether an exception occurs or not. It is often used to release resources or clean up operations.

Let's take a common example:

- Type conversion error

```
try:
    result = 'hello' + 5  # Raises TypeError:
except TypeError as e:
    print("Error:", e)
```

- Divide by zero error

```
try:
    # Code that might raise an exception
    x = int(input("Enter a number: "))
    result = 10 / x
    print("Result:", result)

except ZeroDivisionError:
    # Handle division by zero error
    print("Error: Cannot divide by zero!")

except ValueError:
    # Handle invalid input error
    print("Error: Please enter a valid number!")

else:
    # Executed if no exceptions occur
    print("No exceptions occurred!")

finally:
    # Executed regardless of exceptions
    print("Finally block executed!")
```

Simple Python Project (Homework):

- Create a program that takes inputs from a student (name, roll number, marks in 5 subjects) and then give result (Name, Total marks, average marks, percentage) as output
- Format the output **(optional)**

Tuple(..) vs List [..]:

- Ordered collections of items, separated by commas and enclosed in brackets
- Contain elements of different data types, and they can be nested.
- Elements can be accessed by their index, starting from 0 for the first element.
- Mutation: whether an object's state can be modified after it is created
- **List is Mutable Tuple is not. (Use Cases??)**

```python
my_tuple = (1, 2, 3, 'hello')
# Accessing elements in a tuple
print(my_tuple[0])  # Output: 1
print(my_tuple[-1])  # Output: 'hello'
# Slicing a tuple
print(my_tuple[1:3])  # Output: (2, 3)
print(my_tuple[::2])  #[start:end:step]
# Checking membership
print(2 in my_tuple)  # Output: True
print('world' in my_tuple)  # Output: False
```

```python
my_list = [1, 2, 3, 'hello']
# Adding elements to the end of the list
my_list.append(4)
print(my_list)  # Output: [1, 2, 3, 'hello', 4]
my_list.extend([5, 6])
print(my_list)  # Output: [1, 2, 3, 'hello', 4, 5, 6]
# Inserting/removing elements at a particular position
my_list.insert(2, 'world')
print(my_list)  # Output: [1, 2, 'world', 3, 'hello', 4, 5, 6]
my_list.remove('world')
print(my_list)  # Output: [1, 2, 3, 'hello', 4, 5, 6]
# Pop and Delete
popped_element = my_list.pop(2)
print(popped_element)  # Output: 3
del my_list[0]
print(my_list)  # Output: [2, 'hello', 4, 5, 6]
```

Dictionaries vs Sets:

- Dictionaries are used when you need to store and retrieve data based on keys,
- Sets are used when you need to store a collection of unique elements and perform set operations.

```
# Disctionary
my_dict = {'Name': 'My Name',
           'Some Key': [2, 4, 3]}
next_dict = {'name': 'John',
           'age': 30,
           'city': 'New York'}
print(next_dict['age'])  # Output: 30
```

```
# Sets
my_set = {1, 2, 3, 4, 5, 5, 5}
print(my_set) # Output: {1, 2, 3, 4, 5}
next_set = {1, 2, 3, 4, 5}
next_set.add(6)
print(next_set)  # Output: {1, 2, 3, 4, 5, 6}
```

- Explore operations on Sets and Dictionaries

pip install numpy
pip install pandas
pip install matplotlib
python -m pip install numpy

Conditional Statements:

‘if’ statement:

```
if condition:
    # Code block to execute if the condition is true
```

‘if...else’ statement:

```
if condition:
    # Code block to execute if the condition is true
else:
    # Code block to execute if the condition is false
```

‘if...elif...else’ statement:

```
if condition1:
    # Code block to execute if condition1 is true
elif condition2:
    # Code block to execute if condition2 is true
else:
    # Code block to execute if all conditions are false
```

```python
# If statement
x = input("Enter a number: ")
if x < 0:
    print('Negative')

# If else statement
x = input("Enter 0 or 1: ")
if x == '0':
    print('False')
else:
    print('True')
```

```python
# If elif else statement
x = input("Enter your percentage: ")
if x >= 90:
    print('A')
elif x >= 80:
    print('B')
elif x >= 70:
    print('C')
elif x >= 60:
    print('D')
else:
    print('Fail')
```

Day -3

Looping:

Why Loop? – Repeated task

For Loop:

for variable in sequence:

    # Code block to execute for each element in the sequence

```
# For loop
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
for i in range(5):
    print('i'*i)
```

While Loop:

while condition:

    # Code block to execute as long as the condition is true

```
# While loop
i = 1
while i <= 5:
    print(i)
    i += 1
z = input("Yes/no?")
while z != 'no':
    z = input("Yes/no?")
```

break statement:

- The **break** statement is used to exit a loop prematurely
- based on a certain condition (immediately terminates the loop)

- execution continues with the next statement after the loop.

continue statement:

- The **continue** statement is used to skip the rest of the code inside the loop for the current iteration and proceed to the next iteration.

```
# continue and break
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)

for i in range(10):
    print(i)
    if i == 5:
        break
```

Object-oriented programming (OOP) is a programming paradigm that revolves around the concept of objects, which can contain data in the form of fields (attributes or properties) and code in the form of procedures (methods or functions). Python supports OOP principles and provides features such as classes, objects, inheritance, and polymorphism. Here's an overview of OOP in Python:

1. **Classes and Objects:**
   - A class is a blueprint for creating objects. It defines the attributes (data) and methods (functions) that the objects of the class will have.
   - An object is an instance of a class. It represents a real-world entity that has attributes and behaviors defined by the class.
   - Classes and objects allow you to model real-world entities and encapsulate their data and behavior.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

# Create an object of the Person class
person1 = Person("Alice", 30)
person1.greet()  # Output: Hello, my name is Alice and I am 30 years old.
```

2. **Inheritance:**

- Inheritance allows a class (subclass) to inherit attributes and methods from another class (superclass).
- Subclasses can extend or override the behavior of the superclass.

```python
class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id

    def study(self):
        print(f"{self.name} is studying.")

# Create an object of the Student class
student1 = Student("Bob", 25, "S12345")
student1.greet()  # Output: Hello, my name is Bob and I am 25 years old.
student1.study()  # Output: Bob is studying.
```

3. **Encapsulation:**
   - Encapsulation is the bundling of data and methods that operate on the data within a single unit (class).
   - It allows you to hide the internal state of an object and only expose the necessary functionality through methods.
4. **Polymorphism:**
   - Polymorphism allows objects of different classes to be treated as objects of a common superclass.
   - It enables methods to be invoked on objects without knowing their specific class, as long as they implement the required interface.

NumPy, which stands for Numerical Python, is a powerful Python library for numerical computing. It provides support for multidimensional arrays, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy is widely used in various fields such as scientific computing, machine learning, data analysis, and more. Here's an introduction to some of the key features and functionalities of NumPy:

1. Arrays:
   - The core data structure in NumPy is the ndarray (n-dimensional array), which is a homogeneous collection of elements of the same data type.
   - Arrays can have any number of dimensions, allowing you to represent scalars, vectors, matrices, or even higher-dimensional data structures.
   - NumPy arrays are more efficient in terms of memory usage and computational performance compared to Python lists.
2. Array Creation:
   - You can create NumPy arrays using various methods such as `np.array()`, `np.zeros()`, `np.ones()`, `np.arange()`, `np.linspace()`, and more.
   - Arrays can be initialized from Python lists, tuples, or other array-like objects.
3. Array Operations:
   - NumPy provides a wide range of mathematical operations and functions to perform element-wise computations on arrays.
   - Arithmetic operations, trigonometric functions, exponential and logarithmic functions, statistical functions, etc., can be applied to arrays directly.
4. Indexing and Slicing:
   - NumPy supports advanced indexing and slicing operations to access and manipulate array elements efficiently.

    - You can use integer indexing, boolean indexing, and slicing with start, stop, and step parameters to extract or modify subarrays.
5. Broadcasting:
    - NumPy arrays support broadcasting, which allows you to perform arithmetic operations between arrays of different shapes and sizes.
    - Broadcasting automatically aligns the dimensions of arrays to perform element-wise operations efficiently.
6. Linear Algebra:
    - NumPy provides a comprehensive suite of linear algebra functions for matrix and vector operations.
    - You can perform matrix multiplication, matrix decomposition (e.g., LU decomposition, QR decomposition), eigenvalue and eigenvector calculations, and more.
7. Random Number Generation:
    - NumPy includes a powerful random number generation module (`np.random`) for generating random samples from various probability distributions.
    - Random numbers can be generated for arrays of any shape and size.

Overall, NumPy is a fundamental library for numerical computing in Python, providing efficient data structures and algorithms for performing a wide range of mathematical and scientific computations. It forms the foundation for many other libraries and frameworks in the Python ecosystem, such as SciPy, pandas, and scikit-learn.

In NumPy, one-dimensional arrays are represented using the `ndarray` class. These arrays can be created using various methods provided by NumPy. One-dimensional arrays are commonly used to represent vectors or sequences of data. Here are some common use cases of one-dimensional arrays in NumPy:

1. Data Storage: One-dimensional arrays are used to store and manipulate sequences of data, such as sensor readings, stock prices, or audio signals.
2. Mathematical Operations: NumPy provides a wide range of mathematical operations and functions that can be applied directly to one-dimensional arrays. These operations include arithmetic operations, trigonometric functions, exponential and logarithmic functions, statistical functions, and more.
3. Indexing and Slicing: One-dimensional arrays support advanced indexing and slicing operations, allowing you to access and modify specific elements or subarrays efficiently.
4. Vectorized Operations: NumPy arrays support vectorized operations, which enable efficient element-wise computations without the need for explicit looping. This allows for faster and more concise code compared to traditional Python lists.

5. Integration with Other Libraries: One-dimensional arrays are often used as inputs or outputs for functions in other libraries and frameworks, such as SciPy, pandas, scikit-learn, and matplotlib. These libraries seamlessly integrate with NumPy arrays, allowing for efficient data manipulation, analysis, and visualization.

This example demonstrates some common operations performed on one-dimensional arrays in NumPy, including creating arrays, indexing, modifying elements, performing mathematical operations, and slicing. One-dimensional arrays are versatile data structures that can be used in a wide range of applications in numerical computing and data analysis.

NumPy provides a vast collection of functions for performing various numerical operations efficiently on arrays. These functions are optimized for performance and allow you to work with arrays in a vectorized manner, eliminating the need for explicit loops in many cases. Here are some common categories of functions within NumPy:

1. Mathematical Functions:
    - NumPy includes a wide range of mathematical functions for performing arithmetic operations, trigonometric functions, exponential and logarithmic functions, and more.
    - Examples include `np.add()`, `np.subtract()`, `np.multiply()`, `np.divide()`, `np.sin()`, `np.cos()`, `np.exp()`, `np.log()`, etc.
2. Statistical Functions:
    - NumPy provides functions for calculating various statistical measures such as mean, median, standard deviation, variance, minimum, maximum, percentile, etc.
    - Examples include `np.mean()`, `np.median()`, `np.std()`, `np.var()`, `np.min()`, `np.max()`, `np.percentile()`, etc.
3. Linear Algebra Functions:
    - NumPy offers a comprehensive suite of functions for performing linear algebra operations on arrays, such as matrix multiplication, matrix inversion, eigenvalue decomposition, singular value decomposition, etc.
    - Examples include `np.dot()`, `np.linalg.inv()`, `np.linalg.eig()`, `np.linalg.svd()`, etc.
4. Array Manipulation Functions:

- NumPy provides functions for manipulating the shape, size, and contents of arrays, including reshaping, stacking, splitting, transposing, flipping, and more.
- Examples include `np.reshape()`, `np.concatenate()`, `np.split()`, `np.transpose()`, `np.flip()`, etc.

5. Random Number Generation:
    - NumPy includes a robust random number generation module (`np.random`) for generating random samples from various probability distributions, such as uniform, normal, binomial, etc.
    - Examples include `np.random.rand()`, `np.random.randn()`, `np.random.randint()`, `np.random.choice()`, etc.
6. Logical Functions:
    - NumPy provides functions for performing logical operations on arrays, such as element-wise logical AND, OR, NOT, and XOR operations.
    - Examples include `np.logical_and()`, `np.logical_or()`, `np.logical_not()`, `np.logical_xor()`, etc.

These are just a few examples of the extensive functionality provided by NumPy. The library also includes functions for interpolation, Fourier transforms, signal processing, image processing, and more. NumPy's rich collection of functions makes it a versatile tool for numerical computing and scientific analysis in Python.

One of the fundamental functions in NumPy is `np.array()`, which is used to create NumPy arrays from Python lists, tuples, or other array-like objects. Here's an example demonstrating the basic usage of `np.array()`:

https://colab.research.google.com/drive/1MW3flV7uz_4IrEcz7oK0mPotvabHNL2c?usp=sharing

raw.githubusercontent.com/moest-np/center-randomize/main/sample_data/centers_grade12_2081.tsv

Day -4

No Slides, Just projects

Clone the repo:

https://github.com/rohanrajpoudel/PythonWorkshop

Or Download files from Google Drive:

https://drive.google.com/drive/folders/1E71X5XkqSMrTMy30NktBIX7kGFuAa6CY?usp=sharing

- ☐ Revise Numpy and Pandas
- ☐ Machine Learning (with Data visualization using Matplotlib)
- ☐ Real world application Example (LP Recognition System)
- ☐ Web Scraping Basics
- ☐ Web Scraping Demo

Title of the Session
Session 1: Welcome and overview ● Introduction to the workshop, goals, and expectations ● Overview of Python and its applications
Session 2: Setting UP Python Environment ● Installation of Python and necessary tools (e.g. Jupyter Notebook) ● Introduction to an integrated development environment (IDE) ● Concept of Virtual Environment
Session 3: Python Basics ● Variables, data types, and basic operations ● I/O statements and comments
Session 4: Control Flow ● Introduction to conditional statements (if, else if, else) ● Looping (for and while loops)
Session 5: Functions, Parameters and arguments ● Defining functions , parameters, arguments ● Return statements and function calls
Session 6 : Exception handling ● Introduction to handling errors and exceptions
Session 7 : List and Tuples ● Introduction to basic data structures

- List manipulation and tuple usage

Session 8 : Dictionaries and Sets

- Understanding dictionaries and sets
- Operations on dictionaries and sets

Session 9: File handling

- Reading and writing to files
- Working with different file formats (e.g., text files, CSV, json)

Session 10: Hands on Project

- Small project to apply the concepts learned

Session 11: Introduction to modules

- Understanding Python modules and libraries
- Exploring commonly used modules (e.g., math, random)

Session 12: Object Oriented Programming

- Concept of OOP

Session 13:Introduction to NumPy and Pandas

- Basics of data manipulation using NumPy and Pandas

Session 14: Web Scraping

- Introduction to web scraping for data collection

Session 15: Introduction to Data Visualization

- Basic plotting using libraries like Matplotlib

Session 16: Final Project (Machine Learning) Q & A

Advanced Topics and Real-world applications

- Participants work on a comprehensive final project, incorporating various concepts covered
- Q&A session and review of key takeaways

Session 17: Closing Session

Green = First Day

Yellow = Second Day

Blue = Third Day

Pink = Fourth Day