BIYANI
GROUP OF COLLEGES

Basics of Python

Concept by :
Dr. Sanjay Biyani
Director (Acad.) Biyani Group of Colleges
www.sanjaybiyani.com

Written by :
Ms. Shbna Ali
Ms. Shruti Kumawat
Asso. Prof.
Biyani Girls College

1. Getting Started with IPython

Definition:

IPython (Interactive Python) is an enhanced interactive shell that provides an improved environment for writing, testing, and debugging Python code efficiently. It serves as the core interactive component of Jupyter Notebooks.

Key Features :

- Interactive execution of Python commands — execute code line by line and get instant feedback.
- Syntax highlighting and auto-completion to improve readability and speed up coding.
- Command history and "magic commands" (e.g., %time, %run, %edit) for quick and powerful operations.
- Inline plotting support when used with Jupyter Notebook, allowing direct visualization of graphs and data.

Launching IPython:

You can start IPython from the command line or inside Jupyter Notebook.

$ ipython

or simply use a Jupyter Notebook cell to access IPython functionalities.

Common Magic Commands :

Magic commands (prefixed with % or %%) enhance productivity by extending Python's basic functionality.

Command	Description
%time	Measures the execution time of a single statement.
%run filename.py	Executes an external Python script.
%history	Displays the list of previously executed commands.
%pwd	Prints the current working directory.%lsLists files in the current directory.

2. Using Plot Command Interactively

Introduction:

In data science and exploratory data analysis, visualization plays a key role in understanding patterns, trends, and relationships in data. The plot() command from the Matplotlib library is one of the most fundamental and commonly used functions for creating 2D line plots.

When combined with IPython or Jupyter Notebook, the plot() function becomes highly interactive—allowing users to quickly visualize and modify data without leaving the coding environment.

Definition:

The plot() command in Matplotlib is used to draw simple two-dimensional line plots of data points connected by straight lines.

Syntax:

matplotlib.pyplot.plot(x, y)

x → Sequence of x-coordinates (horizontal axis values)

y → Sequence of y-coordinates (vertical axis values)

Basic Example:

```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3, 4], [10, 20, 25, 30])  # Plotting data
plt.show()  # Displaying the plot
```

Explanation:

[1, 2, 3, 4] → X-axis values

[10, 20, 25, 30] → Y-axis values

plt.show() → Displays the plot window (required in scripts but optional in Jupyter)

Output:

A 2D line plot connecting the points (1,10), (2,20), (3,25), and (4,30).

Interactive Usage:

When using IPython or Jupyter Notebook, Matplotlib plots can be displayed inline — directly below the code cell.

To enable inline plotting in Jupyter:

```
%matplotlib inline
```

Then simply run:

```
plt.plot([1, 2, 3, 4], [10, 20, 25, 30])
```

The graph will appear immediately within the notebook, making it easy to explore and refine your data visualization interactively.

Additional Features of plot():

You can customize the plot with various optional parameters:

```
plt.plot(x, y, color='green', linestyle='--', marker='o')
plt.title("Sample Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.show()
```

Explanation of Parameters:

Parameter	Description	Example
color	Sets the color of the line	'r', 'blue', 'green'
linestyle	Defines the type of line	'-', '--', ':', '-.'
marker	Marks each data point	'o', 's', '^', 'x'
xlabel,ylabel	Label the axes	"Time", "Value"
title	Adds a title to the plot	"Sales Over Time"
grid(True)	Displays grid lines for better readability	—

Advantages of Interactive Plotting:

Instant visualization of data changes.

Easy debugging of numerical or analytical errors.

Quick experimentation with parameters (colors, markers, etc.).

Seamless integration with other libraries like NumPy and Pandas.

3. Embellishing a Plot

Introduction:

Creating a plot is the first step in visualizing data — but to make it informative, readable, and visually appealing, we need to embellish it with additional elements such as titles, labels, grids, legends, and colors. These additions help in communicating insights effectively and make graphs easier to interpret for both technical and non-technical audiences.

Definition:

Embellishing a plot refers to the process of enhancing a basic plot by adding descriptive and visual elements such as:

- Titles (to describe what the plot represents)
- Axis labels (to define what the data represents on each axis)
- Grid lines (to improve readability)
- Legends (to identify plotted data)
- Colors and styles (to visually differentiate data)

Example Code:

```
import matplotlib.pyplot as plt
# Sample data
x = [2018, 2019, 2020, 2021, 2022]
y = [50, 65, 80, 90, 120]
```

```
# Creating and embellishing the plot
plt.plot(x, y, color='r', linestyle='--', marker='o')  # Line style and color
plt.title('Sales Growth')                    # Title
plt.xlabel('Year')                           # X-axis label
plt.ylabel('Revenue (in Lakhs)')             # Y-axis label
plt.grid(True)                               # Grid lines
plt.legend(['Revenue'])                      # Legend
plt.show()
```

Explanation of Each Command:

Command	Purpose	Example / Description
plt.title()	Adds a title to the plot	plt.title('Sales Growth')
plt.xlabel()	Labels the x-axis	plt.xlabel('Year')
plt.ylabel()	Labels the y-axis	plt.ylabel('Revenue (in Lakhs)')
plt.grid(True)	Adds grid lines to the plot background	plt.grid(True)
plt.legend()	Displays the legend identifying data series	plt.legend(['Revenue'])
color	Sets the color of the line	'r' for red, 'b' for blue, 'g' for green
linestyle	Defines line style'--' (dashed),	'-' (solid), ':' (dotted)
marker	Adds symbols at data points	'o' (circle), '^' (triangle), 's' (square)

Visualization Output:

The output is a red dashed line plot with circular markers.

It includes:

- A title ("Sales Growth")
- X-axis labeled as "Year"
- Y-axis labeled as "Revenue (in Lakhs)"
- A legend indicating the line represents "Revenue"
- A grid that makes reading the values easier

Importance of Plot Embellishments:

- Enhances clarity and interpretability of data.
- Provides context to numerical trends.
- Improves presentation quality for reports and research papers.
- Makes plots self-explanatory and visually attractive.

4. Saving Plots

Introduction:

After creating and enhancing visualizations, it is often necessary to save plots for future use in reports, research papers, or presentations. Matplotlib allows saving plots in multiple file formats such as PNG, JPG, PDF, SVG, and more using the savefig() function.

Definition:

The savefig() function in Matplotlib is used to export the current figure to an image or document file.

Syntax:

plt.savefig(filename, dpi=None, bbox_inches=None)

Parameters:

ParameterDescriptionExamplefilenameName of the output file with extension'chart.png', 'output.pdf'dpiDots per inch — controls resolution/claritydpi=300 for high-quality imagesbbox_inchesAdjusts bounding box around plot'tight' trims extra whitespace

Example 1 – Save as PNG file:

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]
plt.plot(x, y, color='r', marker='o')
plt.title('Sales Chart')
plt.xlabel('Quarter')
plt.ylabel('Revenue (in Lakhs)')
plt.grid(True)
# Save the plot as an image
plt.savefig('sales_chart.png')
plt.show()
```

Explanation:

- plt.savefig('sales_chart.png') saves the figure in PNG format.
- The image file will be created in the current working directory.
- The plot is saved before plt.show() because plt.show() may sometimes clear the figure window.

Example 2 – Save as High-Resolution PDF:

```
plt.savefig('report.pdf', dpi=300, bbox_inches='tight')
```

Explanation:

- dpi=300 ensures high-quality output (ideal for printing or publication).
- bbox_inches='tight' removes extra white space around the plot.

Supported File Formats :

Matplotlib supports a wide range of formats:FormatExtensionUse CasePortable Network Graphics.pngDefault format, good for reportsJPEG Image.jpg or .jpegSuitable for web usePortable Document Format.pdfIdeal for reports and printingScalable Vector Graphics.svgBest for scalable illustrationsEncapsulated PostScript.epsUsed for high-quality

PublicationsTips for Saving Plots :

- Always use high DPI (300 or more) for publication-quality images.
- Use bbox_inches='tight' to remove unnecessary borders.
- Choose vector formats like PDF or SVG for scalable graphics.
- Save before calling plt.show() to ensure the plot is not cleared.

5. Multiple Plots

Introduction:

In many cases, it is useful to display multiple lines or datasets on the same graph to compare trends or analyze relationships between variables. Matplotlib's plot() function can be called multiple times to overlay several data series on a single figure.

Example:

```
import matplotlib.pyplot as plt
# Sample data
x = [1, 2, 3, 4, 5]
y1 = [10, 20, 25, 30, 40]
y2 = [15, 18, 22, 27, 35]
# Plotting multiple lines
plt.plot(x, y1, label='Product A', color='r', marker='o')
plt.plot(x, y2, label='Product B', color='b', marker='s')
# Adding embellishments
plt.title('Sales Comparison')
plt.xlabel('Year')
plt.ylabel('Revenue (in Lakhs)')
plt.legend()
plt.grid(True)
plt.show()
```

Explanation:

Two datasets (y1 and y2) are plotted against the same x values.

The legend identifies which line represents which dataset.

Each plot call adds a new line to the same figure.

Use Cases:

Comparing sales trends of multiple products.

Plotting actual vs predicted data.

Displaying multiple experimental results together.

Key Points:

Command	Description
plt.plot() (multiple times)	Plots several datasets on the same graph
label	Used for legend identification
plt.legend()	Displays dataset names
color, marker	Distinguish lines visually

6. Subplots

Introduction:

When you want to visualize different datasets side-by-side, or show different aspects of the same data, you can create multiple plots in a single figure using subplots.

This makes comparisons cleaner and avoids overlapping data.

Definition:

A subplot divides a figure into a grid of smaller plots, allowing multiple charts to appear in one window.

Syntax:

plt.subplot(nrows, ncols, index)

nrows → Number of rows in the grid

ncols → Number of columns in the grid

index → Position of the current plot (counted left to right, top to bottom)

Example:

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y1 = [2, 4, 6, 8, 10]
y2 = [1, 4, 9, 16, 25]
# First subplot
plt.subplot(1, 2, 1)
plt.plot(x, y1, 'r--o')
plt.title('Linear Growth')
```

```
# Second subplot
plt.subplot(1, 2, 2)
plt.plot(x, y2, 'b-^')
plt.title('Quadratic Growth')
plt.tight_layout()
plt.show()
```

Explanation:

subplot(1, 2, 1) → Creates the first of two plots in a single row.

subplot(1, 2, 2) → Creates the second plot beside the first.

plt.tight_layout() → Automatically adjusts spacing to prevent label overlap.

Advantages of Subplots:

- Compare multiple data relationships easily.
- Organize visualizations neatly in one figure.
- Ideal for dashboards or analytical reports.

Key Notes:

Command	Purpose
plt.subplot()	Creates multiple plots within one figure
plt.tight_layout()	Adjusts layout for readability
plt.figure()	Creates a new figure for separate subplot groups

7. Additional Features of IPython

Introduction:

IPython extends the standard Python interpreter with several features that enhance productivity, exploration, and debugging. It is widely used in data analysis, machine learning, and visualization due to its flexibility and interactive capabilities.

Advantages of IPython Environment:

- Speeds up experimentation and debugging.
- Integrates smoothly with NumPy, Pandas, and Matplotlib.
- Makes learning Python and data analysis more interactive and engaging.
- Foundation for Jupyter Notebook, the standard data science interface.

Summary:

Feature	Functionality	Example
%time,	%timeitMeasure execution time	%timeit x**2
%run	Run external Python scripts	%run file.py
%history	Show command history	%history -n 1-5
!command	Run shell commands	!dir, !ls
?, ??	Object introspection	len?, plt.plot??

8. Loading Data from Files

Introduction:

In data science, most datasets are stored in external files such as CSV, TXT, or Excel. Python provides multiple libraries to import data efficiently, such as NumPy and Pandas.

Using NumPy:

```
import numpy as np
data = np.loadtxt('data.txt', delimiter=',')
```

- np.loadtxt() reads numerical data from text files.
- The delimiter parameter specifies how values are separated (e.g., comma , or tab \t).

Example:

If data.txt contains:

```
1,10
2,20
3,30
```

then:

```
print(data)
```

will output:

```
[[ 1. 10.]
 [ 2. 20.]
 [ 3. 30.]]
```

Using Pandas:

```
import pandas as pd
data = pd.read_csv('data.csv')
```

- pd.read_csv() reads data from a CSV file into a DataFrame (a tabular structure).
- You can skip header rows or specify separators:

```
data = pd.read_csv('data.csv', header=1, delimiter=',')
```

Common Parameters:ParameterDescriptionExampledelimiterCharacter separating valuesdelimiter=','headerRow number(s) to use as column namesheader=0skiprowsNumber of rows to skip at the startskiprows=1usecolsLoad specific columnsusecols=[0,2]

9. Plotting Data

Introduction:

After loading data into Python, it can be visualized using Matplotlib to identify trends and patterns.

Example:

```
import matplotlib.pyplot as plt
import pandas as pd
data = pd.read_csv('sales.csv')
plt.plot(data['Year'], data['Sales'])
plt.title('Yearly Sales')
plt.xlabel('Year')
plt.ylabel('Sales')
plt.show()
```

Explanation:

- The x-axis represents Year, and the y-axis represents Sales.
- plt.plot() creates a line plot connecting the data points.

Common Plot Types:

10. Other Types of Plots

Introduction:

Matplotlib supports many types of visualizations beyond simple line plots. Each type highlights a specific aspect of data.

1. Scatter Plot

Used to show relationship between two variables.

```
plt.scatter(x, y)
```

2. Bar Chart

Used to compare categorical data.

```
plt.bar(categories, values)
```

3. Histogram

Used to show distribution of numerical data.

```
plt.hist(data, bins=10)
```

4. Pie Chart

Used to show proportional distribution of categories.

```
plt.pie(values, labels=categories, autopct='%1.1f%%')
```

5. Box Plot

Used to show data spread and outliers.

```
plt.boxplot(data)
```

11. Plotting Charts

Introduction:

Charts are powerful visual tools for summarizing and interpreting data at a glance. Matplotlib supports various chart types for different purposes.

Types of Charts:

Chart Type	Purpose
Line Chart	Shows trends over time or continuous data
Bar Chart	Compares categorical or discrete data
Pie Chart	Displays proportion or percentage share
Histogram	Illustrates frequency distribution

Example – Bar Chart:

```
import matplotlib.pyplot as plt
plt.bar(['A', 'B', 'C'], [20, 35, 30])
plt.title('Category Comparison')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.show()
```

Output:

Displays a bar chart comparing categories A, B, and C with respective values.

Tips for Effective Charting:

- Always label axes and add titles for clarity.
- Use different colors to distinguish data.
- Apply legends when multiple datasets are plotted.
- Choose the right chart type based on data nature (trend, comparison, distribution, etc.).

12. Getting Started with Lists

Definition:

A list in Python is an ordered, mutable (changeable) collection of elements, allowing storage of multiple values in a single variable.

Example:

```
fruits = ['apple', 'banana', 'mango']
```

Common Operations:

Operation	Command	Description
Appendfruits.	append('grape')	Adds a new element
Removefruits.	remove('banana')	Removes an element
Modify	fruits[1] = 'orange'	Changes an element
Access	print(fruits[0])	Access by index
Slice	print(fruits[0:2])	Extracts part of list

Iteration Example:

```
for item in fruits:
   print(item)
```

Output:

apple
orange
mango
grape

13. Getting Started with for

Used to iterate over sequences (lists, strings, etc.)

```
for i in range(5):
print(i)
```

Common uses:

- Looping through lists, files, or dictionaries.
- Nested loops for matrices.

14. Getting Started with Strings

- Definition: Sequence of Unicode characters.

name = "Python Programming”

- **Common Operations**:
- len(name)
- name.lower()
- name.upper()
- name.split()
- name.replace("Python", "Data Science")
- String Formatting:
- print(f"Hello, {name}")

15. Getting Started with Files

Reading and writing files:

```
file = open('data.txt', 'r')
content = file.read()
file.close()
```

Writing:

```
with open('output.txt', 'w') as f:
    f.write("Hello, world!")
```

16. Parsing Data

Extracting structured data from files or text.

```
with open('data.csv') as f:
for line in f:
fields = line.strip().split(',')
print(fields)
```

- Useful for cleaning raw data before analysis.
- Can be combined with regex or pandas.

17. Statistics

Basic descriptive statistics using NumPy or pandas.

```
import numpy as np
data = [10, 20, 30, 40, 50]
np.mean(data)      # Average
np.median(data)    # Median
np.std(data)       # Standard deviation
```

or using pandas:

```
df.describe()
```

18. Getting Started with Arrays

- Arrays are more efficient than lists for numerical computation.
- **NumPy Array :**
- import numpy as np
- arr = np.array([1, 2, 3, 4])
- **Operations:**
- arr * 2
- arr + 5
- np.sqrt(arr)

19. Accessing Parts of Arrays

- **Indexing:**
- arr[0], arr[-1]
- **Slicing:**
- arr[1:3]
- **Boolean indexing:**
- arr[arr > 2]
- **Multidimensional:**
- arr2D[1, 2]   # element at row 1, column 2
- arr2D[:, 0]   # all rows, first column

20. Image Manipulation using Arrays

Images can be represented as **NumPy arrays** (matrices of pixels).

```
from matplotlib import image
from matplotlib import pyplot as plt
img = image.imread('photo.png')
print(img.shape)
plt.imshow(img)
```

- Modify pixel values:
- img[:,:,0] = 0   # Remove red channel
- plt.imshow(img)
- plt.show()
- **Applications:**
  - Brightness/contrast adjustment
  - Cropping, resizing
  - Color filtering and transformations

Python Programming and Matrix Computation Notes

1. Basic Matrix Operations

What is a Matrix?

A **matrix** is a two-dimensional array of numbers arranged in rows and columns.

Creating Matrices using NumPy

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
```

Basic Operations

Operation	Code	Description
Addition	A + B	Element-wise addition
Subtraction	A - B	Element-wise subtraction
Multiplication	A * B	Element-wise multiplication
Division	A / B	Element-wise division
Transpose	A.T	Swap rows and columns
Dot Product	np.dot(A, B)	Matrix multiplication

Example:

```
C = np.dot(A, B)
print©
```

2. Advanced Matrix Operations

Determinant

np.linalg.det(A)

Inverse

np.linalg.inv(A)

Eigenvalues and Eigenvectors

vals, vecs = np.linalg.eig(A)

Rank of a Matrix

np.linalg.matrix_rank(A)

Solving Linear Equations

Solves Ax = b:

x = np.linalg.solve(A, b)

3. Least Square Fit

Used to find the best-fitting line to a given dataset (minimizing squared error).

Example:

```
import numpy as np
x = np.array([1,2,3,4,5])
y = np.array([2.2,2.8,4.5,3.7,5.5])
A = np.vstack([x, np.ones(len(x))]).T
m, c = np.linalg.lstsq(A, y, rcond=None)[0]
print("Slope:", m, "Intercept:", c)
```

Equation of line → y = mx + c

4. Basic Datatypes and Operators

Basic Types:

Type	Example	Description
int	x = 5	Integer number
floaty	= 5.7	Decimal number
str	"Hello"	String
bool	True or False	Boolean
complex	2 + 3j	Complex numbers

Operators:

- Arithmetic: +, -, *, /, %, **, //
- Comparison: ==, !=, <, >, <=, >=
- Logical: and, or, not
- Assignment: =, +=, -=, etc.

5. Sequence Datatypes

Sequences store ordered collections:

- List → mutable
- Tuple → immutable
- String → immutable text
- Range → arithmetic progression

```
nums = [1, 2, 3]
tup = (4, 5, 6)
s = "Python"
r = range(1, 5)
```

6. Input-Output

Input:

```
name = input("Enter your name: ")
```

Output:

```
print("Welcome", name)
print(f"Hello {name}, nice to meet you!")
```

7. Conditional Statements

Used to make decisions:

```
age = 18
if age >= 18:
print("Adult")
elif age > 13:
print("Teenager")
else:
print("Child")
```

Nested ifs and ternary operators can also be used.

8. Loops

For Loop:

```
for i in range(5):
    print(i)
```

While Loop:

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

Loop Control:

- break – exit loop
- continue – skip iteration
- pass – placeholder

9. Manipulating Lists

```
lst = [10, 20, 30]
lst.append(40)
lst.remove(20)
lst.insert(1, 15)
lst.sort()
```

Slicing:

```
lst[1:3]
```

List Comprehension:

```
squares = [x**2 for x in range(5)]
```

10. Manipulating Strings

```
s = "Python Programming"
print(s.lower())
print(s.upper())
print(s.replace("Python", "Data"))
print(s.split())
```

String concatenation:

```
"Hello" + " " + "World"
```

Slicing:

```
s[0:6]  # 'Python'
```

11. Getting Started with Tuples

Immutable sequences:

```
tup = (10, 20, 30)
print(tup[1])
```

Tuples can be unpacked:

```
a, b, c = tup
```

12. Dictionaries

Stores data as **key–value pairs**.

```
student = {'name': 'John', 'age': 20}
student['age'] = 21
student['grade'] = 'A'
```

Methods:

student.keys()
student.values()
student.items()

13. Sets in Python

Unordered collection of unique items.

A = {1, 2, 3}
B = {3, 4, 5}
print(A | B) # Union
print(A & B) # Intersection
print(A - B) # Difference

14. Getting Started with Functions

def greet(name):
return f"Hello, {name}"

Functions make code reusable and organized.

Default and Keyword Arguments:

def add(a, b=10):
return a + b

15. Advanced Features of Functions

- Variable arguments:
- def sum_all(*args):
- return sum(args)
- Lambda functions:
- sq = lambda x: x*x
- Recursion:
- def fact(n):
-     return 1 if n == 0 else n * fact(n-1)

16. Using Python Modules

Modules are files containing Python code.

import math
print(math.sqrt(16))

Custom Module:

# mymodule.py
def hello():
print("Hello World")

Then import:

import mymodule
mymodule.hello()

17. Writing Python Scripts

- Write Python code in a .py file.
- Execute:
- python script.py

Scripts automate tasks, perform computations, or process files.

18. Testing and Debugging

- Syntax Errors: fixed before execution.
- Runtime Errors: occur during execution.
- Logical Errors: give wrong results.

Debugging Tools:

```
import pdb
pdb.set_trace()
```

Unit Testing:

```
import unittest
class TestMath(unittest.TestCase):
def test_add(self):
self.assertEqual(2 + 2, 4)
```

19. Handling Errors and Exceptions

Used to prevent program crashes.

```
try:
x = 10 / 0
except ZeroDivisionError:
   print("Cannot divide by zero!")
finally:
   print("Execution complete.")
```

Raise exceptions manually:

```
raise ValueError("Invalid input!")
```

Summary

Topic	Concept	Key Example
Matrix Operations	NumPyarray	snp.dot(A,B)
Least Squares	Regression fit	np.linalg.lstsq()
Control Flow	if/else, loops	for, while
Data Types	list, dict, tuple, set	append(), keys()
Functions	modular code	def add(a,b)
Modules	reuse code	import math
Error Handling	try-except	ZeroDivisionError

OUR GROUP OF COLLEGES

Girls Colleges

Biyani Girls College
www.biyanicolleges.org
Affiliated to University of Rajasthan
B.B.A.| B.Com. (Pass Course/Hons.) | CA/CS
BCA
B.A.| BVA (Visual Arts)
B.Sc. (Biology/Maths/Biotech.)
M.Com. (ABST) | M.Sc. (Biotech/Physics/Maths/Chemistry)
Zoology (Botany/Environmental Science) | M.A. (Geography)
English Literature/Economics)
IAS/RAS

Biyani Institute of Science & Management for Girls
www.bisma.in
Affiliated to Rajasthan Technical University, Kota
MBA | Ph.D

Biyani Girls B.Ed. College
www.biyanigirlscollege.com
Affiliated to University of Rajasthan
Approved by National Council of Teachers' Education
B.Ed. | M.Ed. | B.Sc.-B.Ed | M.Ed | D.El. Ed.

Biyani School of Nursing & Paramedical Sciences
Biyani Institute of Science & Management (Nurs.)
www.biyaninursingcollege.com
Affiliated to RUHS, Jaipur
Approved by INC and RNC
G.N.M. | B.Sc.Nursing

Biyani Institute of Skill Development for Girls
www.bisd.in
Affiliated to Rajasthan ILD Skill Development University
B.Voc. - Fashion Designing
B. Voc. - Journalism & Mask Communication

Biyani Institute of Yoga & Naturopathy for Girls
www.biyanicolleges.org
Affiliated to Jagadguru Ramanandacharya
Rajasthan Sanskrit University, Jaipur
PGDYT

Co-Ed Colleges

Biyani College of Science & Mgmt. (Co-Ed.)
www.bcsmjaipur.com/edu
Affiliated to University of Rajasthan
B.A. | B.Sc. | B. Com. | BCA-B.Ed. | B.Sc.- B.Ed.

Biyani Law College (Co-Ed.)
www.biyanilawcollege.com
Affiliated to Dr. Bhimrao Ambedkar Law University, Jaipur
Approved by Bar Council of India, New Delhi
BA-LL.B. | LLB. | LLM. | PGD.LL.

Biyani Institute of Pharmaceutical Sciences (Co.Ed.)
www.biyanipharmacycollege.com
Affiliated to RUHS, Jaipur
Approved by Pharmacy Council of India, New Delhi
D. Pharma | B. Pharma

Biyani Institute of Architecture & Design
www.biyanicolleges.org
Affiliated to Rajasthan Technical University, Kota
Approved by AICTE, New Delhi
B. Arch.

Biyani Private ITI (Co-Ed.)
www.biyaniiti.com
Approved by Quality Council of India, New Delhi
ITI Trade- Elecrician

Biyani Institute of Physical Education (Co-Ed.)
www.bcsmjaipur.com
Affiliated to University of Rajasthan
B.P. Ed.

Jaipur Institute of Yoga & Naturopathy
www.biyanicolleges.org
Affiliated to Jagadguru Ramanandacharya
Rajasthan Sanskrit University, Jaipur
PGDYT

Department of Information Technology
Biyani Girls College
Sector-3, Vidhyadhar Nagar, Jaipur,Rajasthan
+91-8696218218,+91-8290638942
acad@biyanicolleges.org