Introduction

1. **About the work book:**
   This workbook is intended to be used by S.Y.B.Sc. (Computer Science) students for Mathematics practical course (Python Programming) in Semester–IV. This workbook is designed by considering all the practical concepts / topics mentioned in syllabus.

2. **The objectives of this workbook are:**
   1) Defining the scope of the course.
   2) To have continuous assessment of the course and students.
   3) Providing ready reference for the students during practical implementation.
   4) Provide more options to students so that they can have good practice before facing the examination.
   5) Catering to the demand of slow and fast learners and accordingly providing the practice assignments to them.

3. Instructions to the students

Please read the following instructions carefully and follow them.

1. Students are expected to carry this book every time they come to the lab for practical.
2. Students should prepare oneself beforehand for the Assignment by reading the relevant material.
3. Instructor will specify which problems to solve in the lab during the allotted slot and student should complete them and get verified by the instructor. However student should spend additional hours in Lab and at home to cover as many problems as possible.
4. Students will be assessed for each exercise on a scale from 0 to 5.

Not Done	0
Incomplete	1
Late Complete	2
Needs Improvement	3
Complete	4
Well Done	5

4. Instruction to the Instructors

1) Explain the assignment and related concepts in around ten minutes using white board if required or by demonstrating the software.
2) You should evaluate each assignment carried out by a student on a scale of 5 as specified above by ticking appropriate box.
3) The value should also be entered on assignment completion page of the respective Lab course.

S.Y.B.Sc.(Comp.Sc.) Mathematics Practical

Certificate

This is to certify that Mr./Ms.____ ,
Roll No. ___________ of S.Y.B.Sc.(Comp.Sc.) has satisfactorily completed all the assignments of Mathematics practical in Semester IV of the academic year 20 .

Date: Batch In-charge

Internal Examiner External Examiner

Assignment Completion Sheet

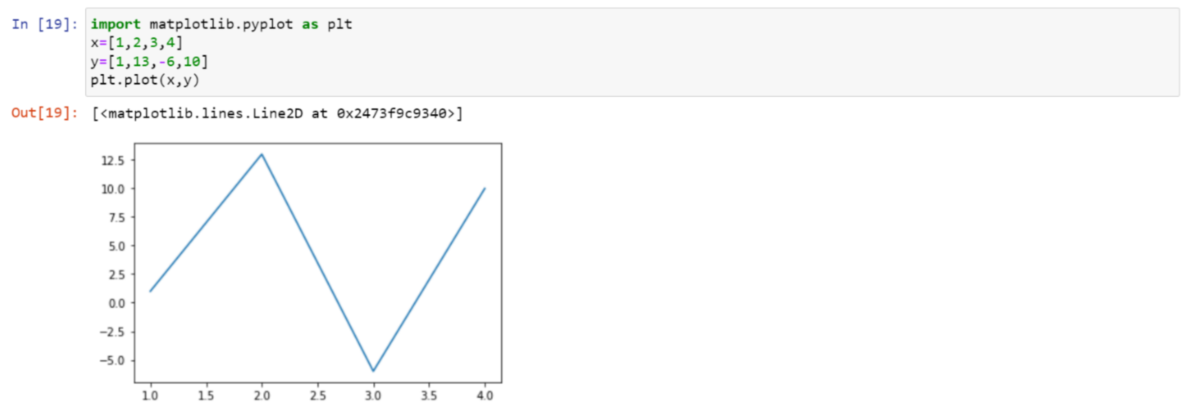
Assignment No	**Description**	**Marks (out of 2)**
1	Drawing 2D curves, 3D surfaces	
2	Drawing bar graphs	
3	Polygon, line segment, ray	
4	2D Transformations –Reflection, rotation, scaling	
5	Concatenated transformation matrix – 2D	
6	Concatenated transformation matrix – 3D	
7	Solving LPP – Pulp module	
8	Solving LPP - Simplex	

Assignment 1

2D-plotting in matplotlib

matplotlib is the workhorse of visualization in Python. To see how plotting with matplotlib works, let's start with a simple example of 2D curve plotting. matplotlib.pyplot is a collection of command-style functions and methods that have been intentionally made to work similar to MATLAB plotting functions. Each pyplot function makes some changes to a figure. For example, it can create a figure, like plt.plot() in the above example, or decorates the plot with labels, texts, etc, as will be seen below.

First import matplotlib.pyplot and then use plot command for plotting 2D curves.



```
In [19]: import matplotlib.pyplot as plt
         x=[1,2,3,4]
         y=[1,13,-6,10]
         plt.plot(x,y)

Out[19]: [<matplotlib.lines.Line2D at 0x2473f9c9340>]
```

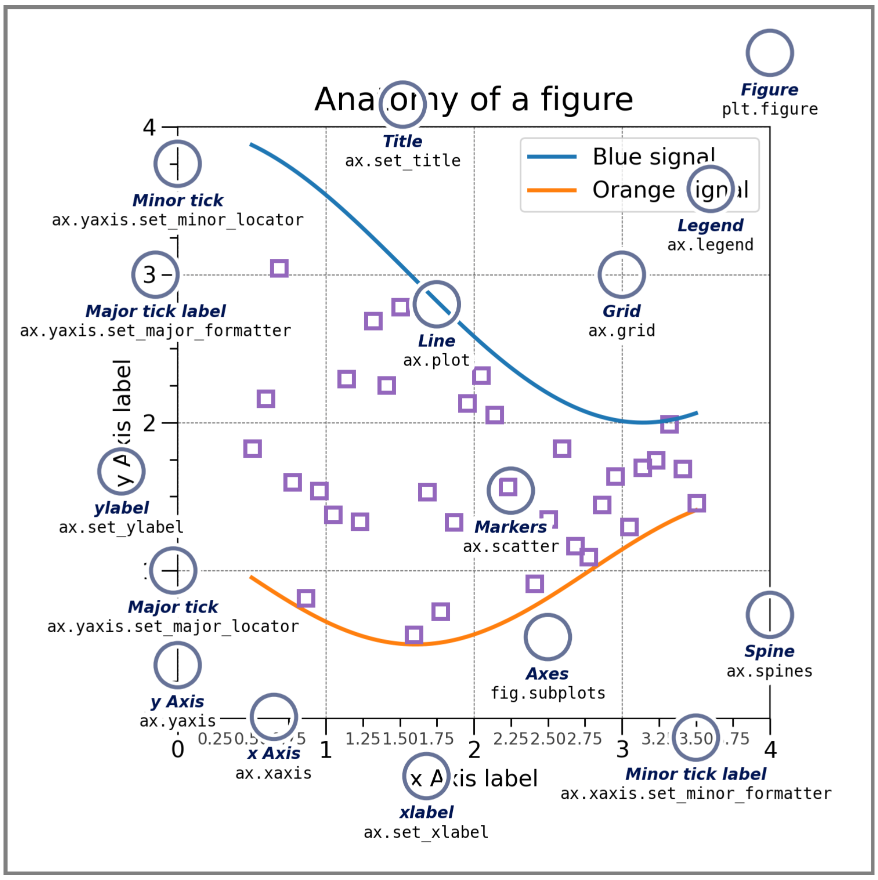
Matplotlib graphs your data on **Figure**s (e.g., windows, Jupyter widgets, etc.), each of which can contain one or more **Axes**, an area where points can be specified in terms of x-y coordinates (or theta-r in a polar plot, x-y-z in a 3D plot, etc.). The simplest way of creating a Figure with an Axes is using **pyplot.subplots**. We can then use **Axes.plot** to draw some data on the Axes:

```
fig, ax = plt.subplots()  # Create a figure containing a single axes.
ax.plot([1, 2, 3, 4], [1, 4, 2, 3])  # Plot some data on the axes.
```

Note that to get this Figure to display, you may have to call plt.show(), depending on your backend. For more details of Figures and backends, see Introduction to Figures.

Parts of a Figure

Here are the components of a Matplotlib Figure.



Figure

The **whole** figure. The Figure keeps track of all the child **Axes**, a group of 'special' Artists (titles, figure legends, colorbars, etc), and even nested subfigures.

The easiest way to create a new Figure is with pyplot:

```
fig = plt.figure()  # an empty figure with no Axes
fig, ax = plt.subplots()  # a figure with a single Axes
fig, axs = plt.subplots(2, 2)  # a figure with a 2x2 grid of Axes
# a figure with one axes on the left, and two on the right:
fig, axs = plt.subplot_mosaic([['left', 'right_top'],
                               ['left', 'right_bottom']])
```

It is often convenient to create the Axes together with the Figure, but you can also manually add Axes later on. Note that many Matplotlib backends support zooming and panning on figure windows.

Axes

An Axes is an Artist attached to a Figure that contains a region for plotting data, and usually includes two (or three in the case of 3D) **Axis** objects (be aware of the difference between **Axes** and **Axis**) that provide ticks and tick labels to provide scales for the data in the Axes. Each **Axes** also has a title (set via **set_title()**), an x-label (set via **set_xlabel()**), and a y-label set via **set_ylabel()**).

The **Axes** class and its member functions are the primary entry point to working with the OOP interface, and have most of the plotting methods defined on them (e.g. ax.plot(), shown above, uses the **plot** method)

Axis

These objects set the scale and limits and generate ticks (the marks on the Axis) and ticklabels (strings labeling the ticks). The location of the ticks is determined by a **Locator** object and the ticklabel strings are formatted by a **Formatter**. The combination of the correct **Locator** and **Formatter** gives very fine control over the tick locations and labels.

Artist

Basically, everything visible on the Figure is an Artist (even **Figure**, **Axes**, and **Axis** objects). This includes **Text** objects, **Line2D** objects, **collections** objects, **Patch** objects, etc. When the Figure is rendered, all of the Artists are drawn to the **canvas**. Most Artists are tied to an Axes; such an Artist cannot be shared by multiple Axes, or moved from one to another.

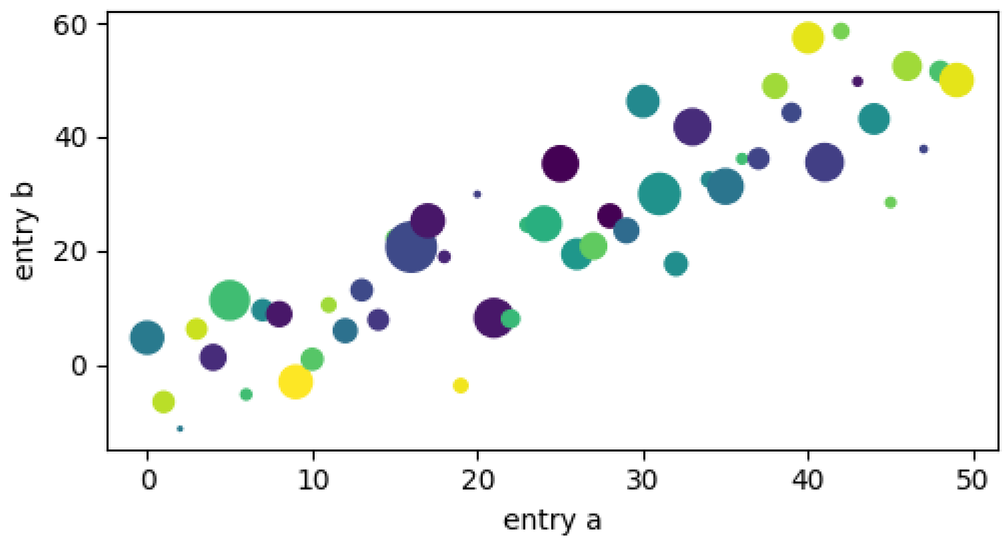
Types of inputs to plotting functions

Plotting functions expect **numpy.array** or **numpy.ma.masked_array** as input, or objects that can be passed to **numpy.asarray**. Classes that are similar to arrays ('array-like') such as **pandas** data objects and **numpy.matrix** may not work as intended. Common convention is to convert these to **numpy.array** objects prior to plotting. For example, to convert a **numpy.matrix**

```
b = np.matrix([[1, 2], [3, 4]])
b_asarray = np.asarray(b)
```

Most methods will also parse a string-indexable object like a *dict*, a structured **numpy array**, or a **pandas.DataFrame**. Matplotlib allows you to provide the data keyword argument and generate plots passing the strings corresponding to the *x* and *y* variables.

```
np.random.seed(19680801)  # seed the random number generator.
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
ax.scatter('a', 'b', c='c', s='d', data=data)
ax.set_xlabel('entry a')
ax.set_ylabel('entry b')
```



Coding styles

The explicit and the implicit interfaces

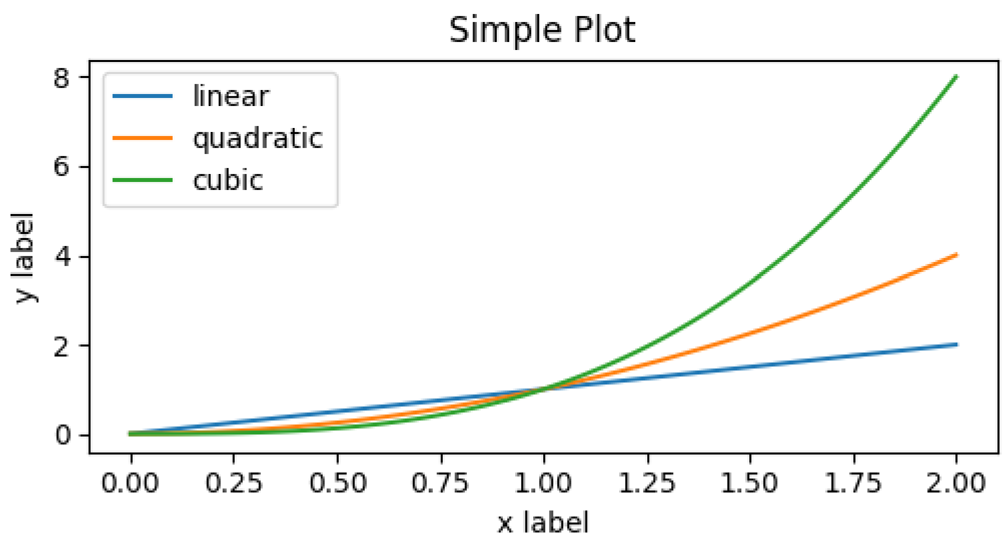
As noted above, there are essentially two ways to use Matplotlib:

- Explicitly create Figures and Axes, and call methods on them (the "object-oriented (OO) style").
- Rely on pyplot to implicitly create and manage the Figures and Axes, and use pyplot functions for plotting.

See Matplotlib Application Interfaces (APIs) for an explanation of the tradeoffs between the implicit and explicit interfaces.

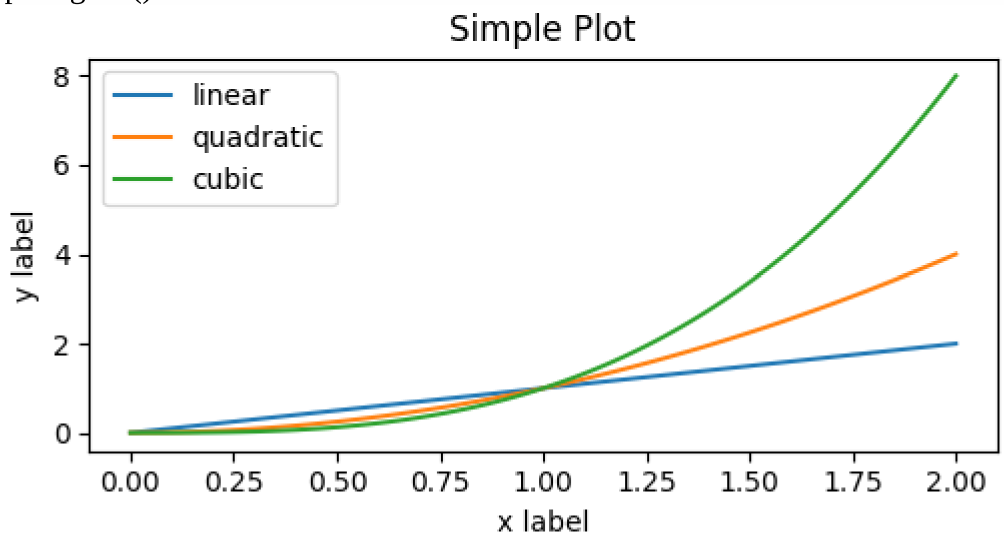
So one can use the OO-style

```
x = np.linspace(0, 2, 100)  # Sample data.

# Note that even in the OO-style, we use `.pyplot.figure` to create the Figure.
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
ax.plot(x, x, label='linear')  # Plot some data on the axes.
ax.plot(x, x**2, label='quadratic')  # Plot more data on the axes...
ax.plot(x, x**3, label='cubic')  # ... and some more.
ax.set_xlabel('x label')  # Add an x-label to the axes.
ax.set_ylabel('y label')  # Add a y-label to the axes.
ax.set_title("Simple Plot")  # Add a title to the axes.
ax.legend()  # Add a legend.
```



or the pyplot-style:

```
x = np.linspace(0, 2, 100)  # Sample data.

plt.figure(figsize=(5, 2.7), layout='constrained')
plt.plot(x, x, label='linear')  # Plot some data on the (implicit) axes.
plt.plot(x, x**2, label='quadratic')  # etc.
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()
```



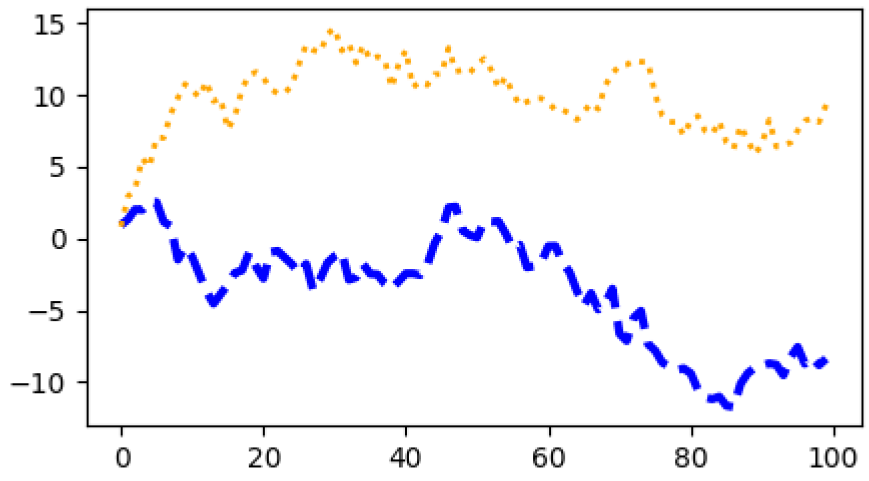
(In addition, there is a third approach, for the case when embedding Matplotlib in a GUI application, which completely drops pyplot, even for figure creation. See the corresponding section in the gallery for more info: Embedding Matplotlib in graphical user interfaces.)

Matplotlib's documentation and examples use both the OO and the pyplot styles. In general, we suggest using the OO style, particularly for complicated plots, and functions and scripts that are intended to be reused as part of a larger project. However, the pyplot style can be very convenient for quick interactive work.

Styling Artists

Most plotting methods have styling options for the Artists, accessible either when a plotting method is called, or from a "setter" on the Artist. In the plot below we manually set the *color*, *linewidth*, and *linestyle* of the Artists created by **plot**, and we set the linestyle of the second line after the fact with **set_linestyle**.

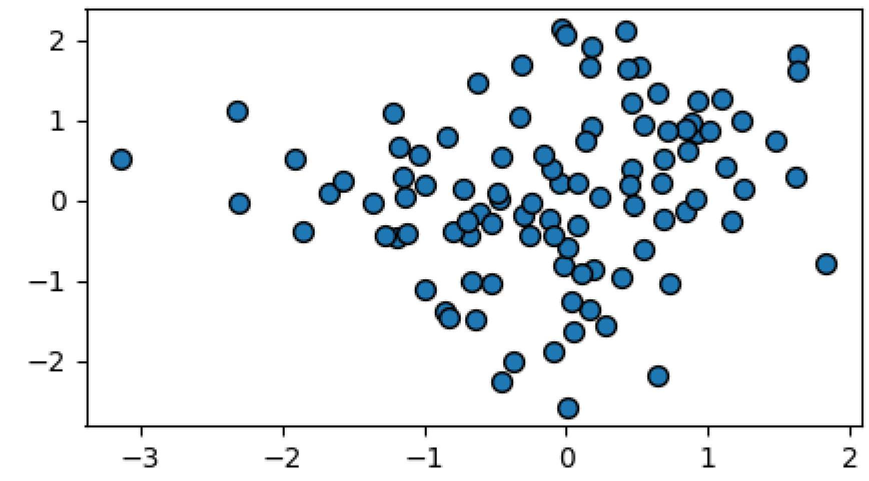
```
fig, ax = plt.subplots(figsize=(5, 2.7))
x = np.arange(len(data1))
ax.plot(x, np.cumsum(data1), color='blue', linewidth=3, linestyle='--')
l, = ax.plot(x, np.cumsum(data2), color='orange', linewidth=2)
l.set_linestyle(':')
```



Colors

Matplotlib has a very flexible array of colors that are accepted for most Artists; see allowable color definitions for a list of specifications. Some Artists will take multiple colors. i.e. for a **scatter** plot, the edge of the markers can be different colors from the interior:

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.scatter(data1, data2, s=50, facecolor='C0', edgecolor='k')
```

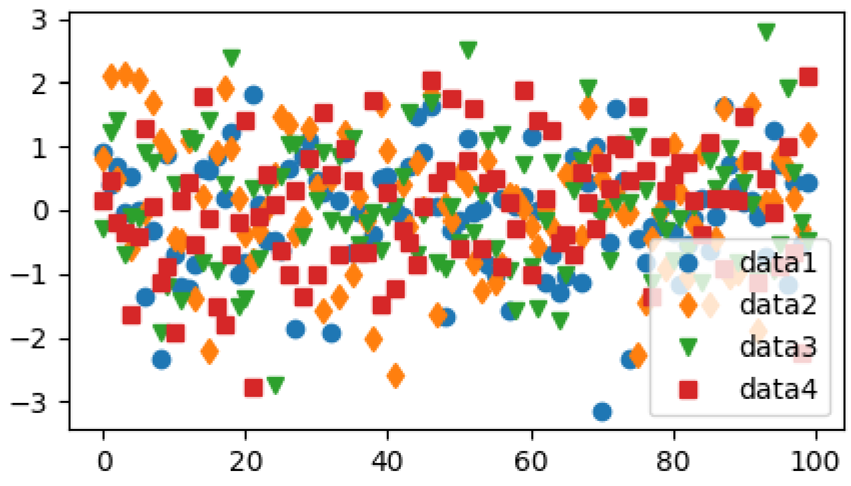


Linewidths, linestyles, and markersizes

Line widths are typically in typographic points (1 pt = 1/72 inch) and available for Artists that have stroked lines. Similarly, stroked lines can have a linestyle. See the linestyles example.

Marker size depends on the method being used. **plot** specifies markersize in points, and is generally the "diameter" or width of the marker. **scatter** specifies markersize as approximately proportional to the visual area of the marker. There is an array of markerstyles available as string codes (see **markers**), or users can define their own **MarkerStyle** (see Marker reference):

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.plot(data1, 'o', label='data1')
ax.plot(data2, 'd', label='data2')
ax.plot(data3, 'v', label='data3')
ax.plot(data4, 's', label='data4')
ax.legend()
```

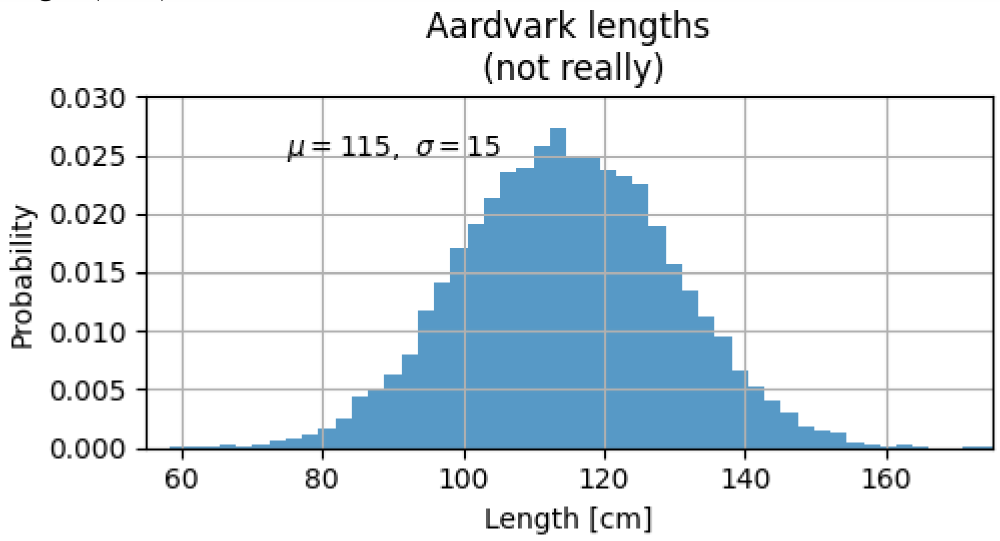


Labelling plots

Axes labels and text

set_xlabel, **set_ylabel**, and **set_title** are used to add text in the indicated locations (see Text in Matplotlib for more discussion). Text can also be directly added to plots using **text**:

```
mu, sigma = 115, 15
x = mu + sigma * np.random.randn(10000)
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
# the histogram of the data
n, bins, patches = ax.hist(x, 50, density=True, facecolor='C0', alpha=0.75)

ax.set_xlabel('Length [cm]')
ax.set_ylabel('Probability')
ax.set_title('Aardvark lengths\n (not really)')
ax.text(75, .025, r'$\mu=115,\ \sigma=15$')
ax.axis([55, 175, 0, 0.03])
ax.grid(True)
```



All of the **text** functions return a **matplotlib.text.Text** instance. Just as with lines above, you can customize the properties by passing keyword arguments into the text functions:

```
t = ax.set_xlabel('my data', fontsize=14, color='red')
```

These properties are covered in more detail in Text properties and layout.

Using mathematical expressions in text

Matplotlib accepts TeX equation expressions in any text expression. For example to write the expression ��=15 in the title, you can write a TeX expression surrounded by dollar signs:

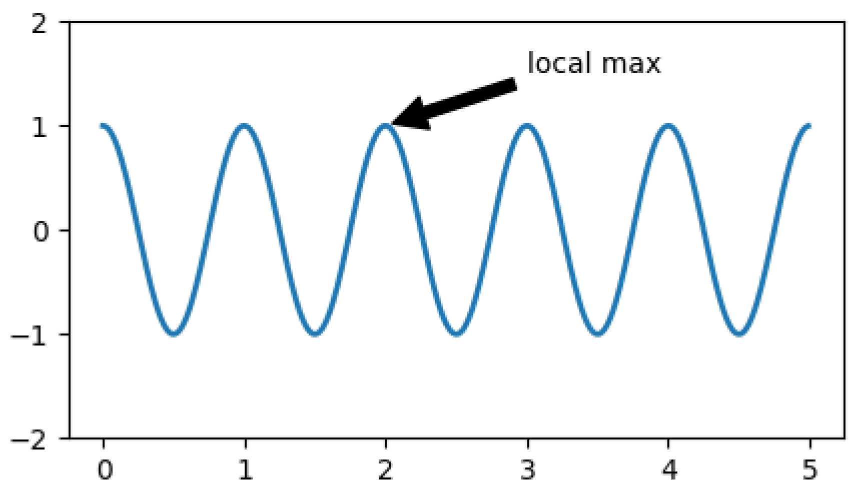
```
ax.set_title(r'$\sigma_i=15$')
```

where the r preceding the title string signifies that the string is a *raw* string and not to treat backslashes as python escapes. Matplotlib has a built-in TeX expression parser and layout engine, and ships its own math fonts – for details see Writing mathematical expressions. You can also use LaTeX directly to format your text and incorporate the output directly into your display figures or saved postscript – see Text rendering with LaTeX.

Annotations

We can also annotate points on a plot, often by connecting an arrow pointing to *xy*, to a piece of text at *xytext*:

```
fig, ax = plt.subplots(figsize=(5, 2.7))

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2 * np.pi * t)
line, = ax.plot(t, s, lw=2)

ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05))

ax.set_ylim(-2, 2)
```

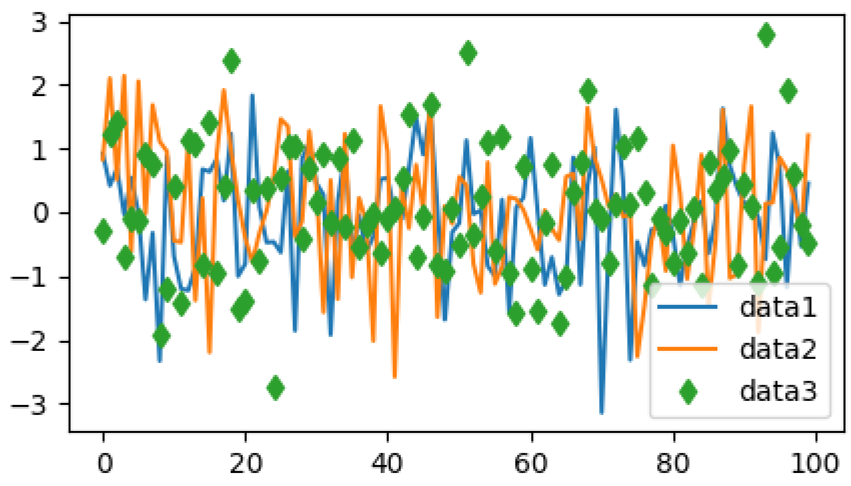


In this basic example, both *xy* and *xytext* are in data coordinates. There are a variety of other coordinate systems one can choose -- see Basic annotation and Advanced annotation for details. More examples also can be found in Annotating Plots.

Legends

Often we want to identify lines or markers with a **Axes.legend**:

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.plot(np.arange(len(data1)), data1, label='data1')
ax.plot(np.arange(len(data2)), data2, label='data2')
ax.plot(np.arange(len(data3)), data3, 'd', label='data3')
ax.legend()
```



Legends in Matplotlib are quite flexible in layout, placement, and what Artists they can represent. They are discussed in detail in Legend guide.

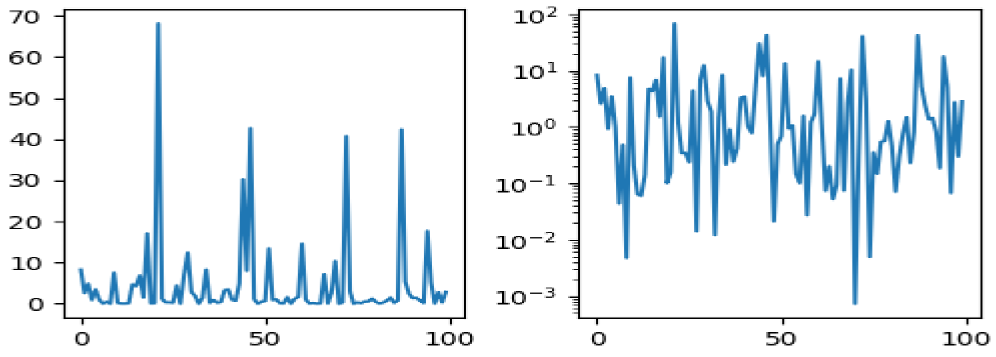
Axis scales and ticks

Each Axes has two (or three) **Axis** objects representing the x- and y-axis. These control the *scale* of the Axis, the tick *locators* and the tick *formatters*. Additional Axes can be attached to display further Axis objects.

Scales

In addition to the linear scale, Matplotlib supplies non-linear scales, such as a log-scale. Since log-scales are used so much there are also direct methods like **loglog**, **semilogx**, and **semilogy**. There are a number of scales (see Scales for other examples). Here we set the scale manually:

```
fig, axs = plt.subplots(1, 2, figsize=(5, 2.7), layout='constrained')
xdata = np.arange(len(data1))  # make an ordinal for this
data = 10**data1
axs[0].plot(xdata, data)

axs[1].set_yscale('log')
axs[1].plot(xdata, data)
```

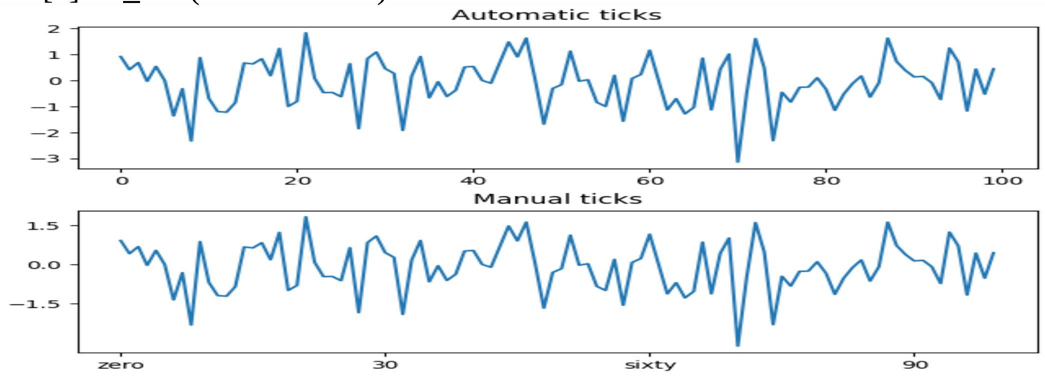


The scale sets the mapping from data values to spacing along the Axis. This happens in both directions, and gets combined into a *transform*, which is the way that Matplotlib maps from data coordinates to Axes, Figure, or screen coordinates. See Transformations Tutorial.

Tick locators and formatters

Each Axis has a tick *locator* and *formatter* that choose where along the Axis objects to put tick marks. A simple interface to this is **set_xticks**:

```
fig, axs = plt.subplots(2, 1, layout='constrained')
axs[0].plot(xdata, data1)
axs[0].set_title('Automatic ticks')

axs[1].plot(xdata, data1)
axs[1].set_xticks(np.arange(0, 100, 30), ['zero', '30', 'sixty', '90'])
axs[1].set_yticks([-1.5, 0, 1.5])  # note that we don't need to specify labels
axs[1].set_title('Manual ticks')
```

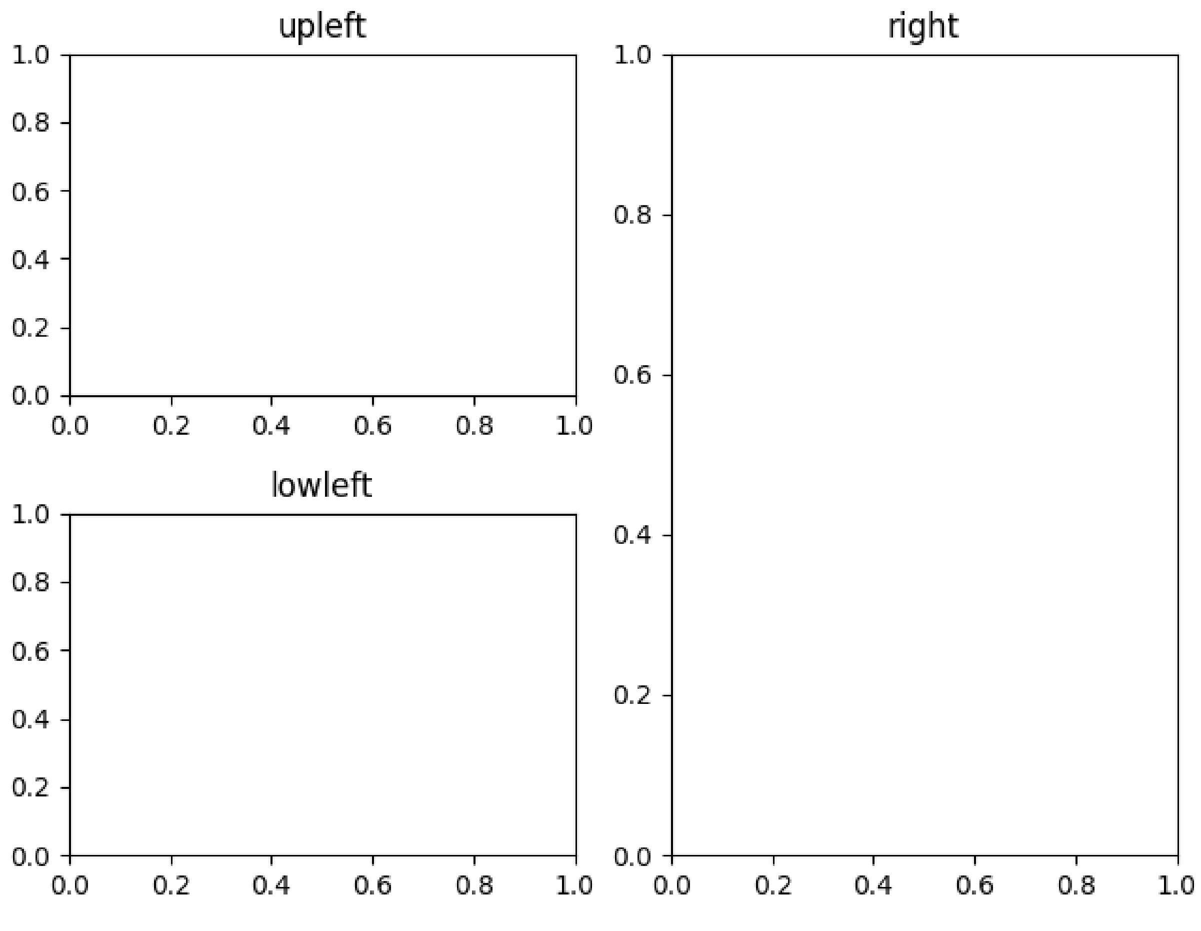


Working with multiple Figures and Axes

You can open multiple Figures with multiple calls to fig = plt.figure() or fig2, ax = plt.subplots(). By keeping the object references you can add Artists to either Figure.

Multiple Axes can be added a number of ways, but the most basic is plt.subplots() as used above. One can achieve more complex layouts, with Axes objects spanning columns or rows, using **subplot_mosaic**.

```
fig, axd = plt.subplot_mosaic([['upleft', 'right'],

                               ['lowleft', 'right']], layout='constrained')

axd['upleft'].set_title('upleft')

axd['lowleft'].set_title('lowleft')

axd['right'].set_title('right')
```

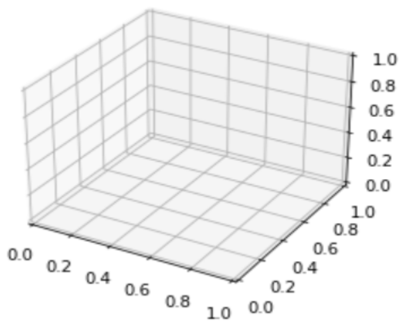


Matplotlib has quite sophisticated tools for arranging Axes: See Arranging multiple Axes in a Figure and Complex and semantic figure composition (subplot_mosaic).

3D Plotting

Getting started An Axes3D object is created just like any other axes using the projection=‘3d’ keyword. Create a new matplotlib.figure.Figure and add a new axes to it of type Axes3D:

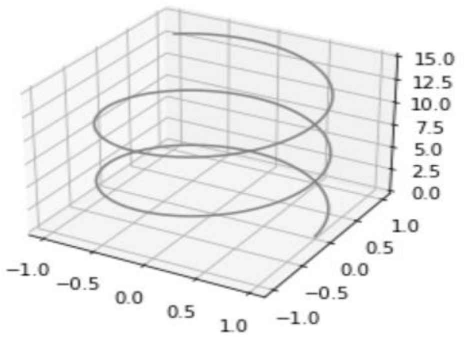
```
In [14]: import matplotlib.pyplot as plt
         from mpl_toolkits.mplot3d import Axes3D
         fig = plt.figure()
         ax = fig.add_subplot(111, projection='3d')
```



Line plots Axes3D.plot(xs, ys, *args,* *kwargs) Plot 2D or 3D data.

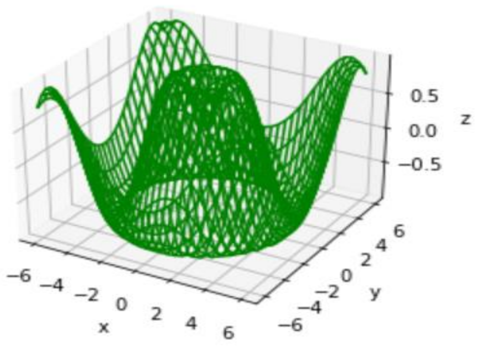
Argument Description xs, ys x, y coordinates of vertices zs z value(s), either one for all points or one for each point. zdir Which direction to use as z (‘x’, ‘y’ or ‘z’) when plotting a 2D set.

```
In [15]: fig = plt.figure()
         ax = plt.axes(projection="3d")

         z_line = np.linspace(0, 15, 1000)
         x_line = np.cos(z_line)
         y_line = np.sin(z_line)
         ax.plot3D(x_line, y_line, z_line, 'gray')


         plt.show()
```

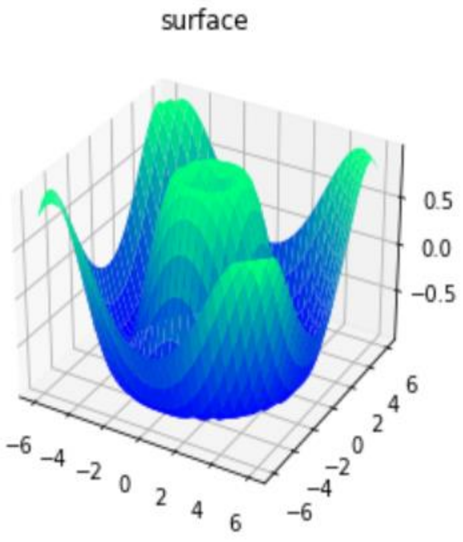


Surface Plots Surface plots can be great for visualizing the relationships among 3 variables across the entire 3D landscape. They give a full structure and view as to how the value of each variable changes across the axes of the 2 others. Constructing a surface plot in Matplotlib is a 3-step process. (1) First we need to generate the actual points that will make up the surface plot. Now, generating all the points of the 3D surface is impossible since there are an infinite number of them! So instead, we’ll generate just enough to be able to estimate the surface and then extrapolate the rest of the points. We’ll define the x and y points and then compute the z points using a function.

```
In [17]: fig = plt.figure()
         ax = plt.axes(projection="3d")
         ax.plot_wireframe(X, Y, Z, color='green')
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         ax.set_zlabel('z')

         plt.show()
```



```
In [18]: ax = plt.axes(projection='3d')
         ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                         cmap='winter', edgecolor='none')
         ax.set_title('surface');
```



Assignment 1

1. Plot 2D graph of the function $f(x)=x^2$ and $g(x)=x^3$ in [-1,1].
2. Write a Python program to plot 2D graph of the functions f(x) = log10(x) in the interval [0, 10].
3. Plot graph of $f(x)= e^{(-x2)}$ in [-5,5] with green dashed points upward pointing triangles.
4. Using Python plot the graph of function f(x) = cos(x) on the interval [0, 2π].
5. Using Python plot the graph of function $f(x) = \sin^{-1}(x)$ on the interval [−1, 1].
6. Plot the graph of $f(x) = x^4$ in [0, 5] with red dashed line with circle markers
7. Plot the graphs of sin x, cos x, e^x and x^2 in [0, 5] in one figure with (2 × 2) subplots.
8. Write a python program to Plot 2DX-axis and Y-axis black color and in the same diagram plot green triangle with vertices [5, 4], [7, 4], [6, 6].
9. Write a Python program to generate 3D plot of the functions z=sin x + cos y
10. Using python, generate 3D surface Plot for the function $f(x) = \sin(x^2 + y^2)$ in the interval [0, 10].
11. Write a Python program to generate 3D plot of the functions z = sin x+ cos y in $-10 < x, y < 10$.
12. Using Python plot the surface plot of function z = cos(x2 + y2 − 0.5) in the interval from -1<x, y < 1.
13. Write a Python program to generate 3D plot of the surface z=x^2 + y^2
14. Plot 3D axes with labels and plot (70, -25, 15)

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of Instructor

Assignment No. 2 : Drawing Bar graph

1. Represent the given information using a bar graph (in green color).

Item	clothing	food	rent	fuel	misc
Expenditure	600	4000	2000	1500	700

2. Plot bar graph from given information with bar color brown.
   City=['pune', 'mumbai', 'nashik', 'nagpur', 'thane'],
   Air quality index = [168,190,170,178, 195].
3. Using python, draw a bar graph in green colour to represent the data below:

Subject:	Maths,	Science,	English,	Marathi,	Hindi.
Percentage of passing:	68,	90,	70,	85,	91.

4. Following is the information of students participating in various games in a school. Represent it by a Bar graph with bar width of 0.7 inches.

Game:	Cricket,	Football,	Hockey,	Chess,	Tennis.
Number of students:	65,	30,	54,	10,	20

Assignment Evaluation

0: Not Done [ ]　　1: Incomplete [ ]　　2: Late Complete [ ]

3: Needs Improvement [ ]　　4: Complete [ ]　　5: Well Done [ ]

Signature of Instructor

Assignment No. 3 : Polygon, line segment, ray

Introduction

The geometry module for SymPy allows one to create two-dimensional geometrical entities, such as lines and circles, and query for information about these entities. This could include asking the area of an ellipse, checking for collinearity of a set of points, or finding the intersection between two lines. The primary use case of the module involves entities with numerical values, but it is possible to also use symbolic representations.

Available Entities

The following entities are currently available in the geometry module:

- Point
- Line, Segment, Ray
- Ellipse, Circle
- Polygon, RegularPolygon, Triangle

Most of the work one will do will be through the properties and methods of these entities, but several global methods exist:

- intersection(entity1, entity2)
- are_similar(entity1, entity2)
- convex_hull(points)

For a full API listing and an explanation of the methods and their return values please see the list of classes at the end of this document.

Example

The following Python session gives one an idea of how to work with some of the geometry module.

```
>>> from sympy import *
>>> from sympy.geometry import *
>>> x = Point(0, 0)
>>> y = Point(1, 1)
>>> z = Point(2, 2)
>>> zp = Point(1, 0)
>>> Point.is_collinear(x, y, z)
True
>>> Point.is_collinear(x, y, zp)
False
>>> t = Polygon(zp, y, x)
>>> t.area
1/2
>>> t.medians[x]
Segment2D(Point2D(0, 0), Point2D(1, 1/2))
>>> m = t.medians
>>> intersection(m[x], m[y], m[zp])
[Point2D(2/3, 1/3)]
>>> c = Circle(x, 5)
>>> l = Line(Point(5, -5), Point(5, 5))
>>> c.is_tangent(l) # is l tangent to c?
True
>>> l = Line(x, y)
>>> c.is_tangent(l) # is l tangent to c?
False
>>> intersection(c, l)
[Point2D(-5*sqrt(2)/2, -5*sqrt(2)/2), Point2D(5*sqrt(2)/2, 5*sqrt(2)/2)]
```

Intersection of medians

```
>>> from sympy import symbols
>>> from sympy.geometry import Point, Triangle, intersection

>>> a, b = symbols("a,b", positive=True)

>>> x = Point(0, 0)
>>> y = Point(a, 0)
>>> z = Point(2*a, b)
>>> t = Triangle(x, y, z)

>>> t.area
a*b/2

>>> t.medians[x]
Segment2D(Point2D(0, 0), Point2D(3*a/2, b/2))

>>> intersection(t.medians[x], t.medians[y], t.medians[z])
[Point2D(a, b/3)]
```

Command/ attribute/ method	**Used for**
p.area	Area of polygon p
p.perimeter	Perimeter of polygon p
s.midpoint	Midpoint of segment s
s1.angle_between(s2)	Angle between segments s1 and s2
l.distance(a)	Distance of a point a from line l
s.slope	Slope of segment/ ray/ line s
s.length	Length of segment s

Notes

- The area property of Polygon and Triangle may return a positive or negative value, depending on whether or not the points are oriented counter-clockwise or clockwise, respectively. If you always want a positive value be sure to use the abs function.
- Although Polygon can refer to any type of polygon, the code has been written for simple polygons. Hence, expect potential problems if dealing with complex polygons (overlapping sides).
- Since SymPy is still in its infancy some things may not simplify properly and hence some things that should return True (e.g., Point.is_collinear) may not actually do so. Similarly, attempting to find the intersection of entities that do intersect may result in an empty result.

Assignment 3

1. Find area and perimeter of a triangle with vertices (0,0), (5,0), (3,3).
2. Write a Python code to find the area and perimeter of the ∆ABC, where A[0, 0], B[6, 0], C[4, 4].
3. Write a python code to find the area and perimeter of the ∆XY Z, where X(1, 2), Y (2,−2), Z(−1, 2).
4. Using sympy declare the points A(0, 2),B(5, 2),C(3, 0) check whether these points are collinear. Declare the line passing through the points A and B, find the distance of this line from point C.
5. Draw a regular polygon with 6 sides and radius 1 centered at (1, 2) and find its area and perimeter.
6. Draw a polygon with 8 sides having radius 5 centered at origin and find its area and perimeter.
7. Generate line segment having endpoints (0, 0) and (10, 10) find midpoint of line segment.
8. Generate line passing through points (2, 3) and (4, 3) and find equation of the line.
9. Using sympy, declare the points A(0, 7), B(5, 2). Declare the line segment passing through them. Find the length and midpoint of the line segment passing through points A and B.
10. Using python, generate triangle with vertices (0, 0), (4, 0), (4, 3) and check whether the triangle is right angle triangle.
11. Generate triangle with vertices (0, 0), (4, 0), (2, 4), check whether the triangle is isosceles triangle.
12. Using sympy declare the points P(5, 2),Q(5,−2),R(5, 0), check whether these points are collinear. Declare the ray passing through the points P and Q, find the length of this ray between P and Q. Also find slope of this ray.

Assignment Evaluation

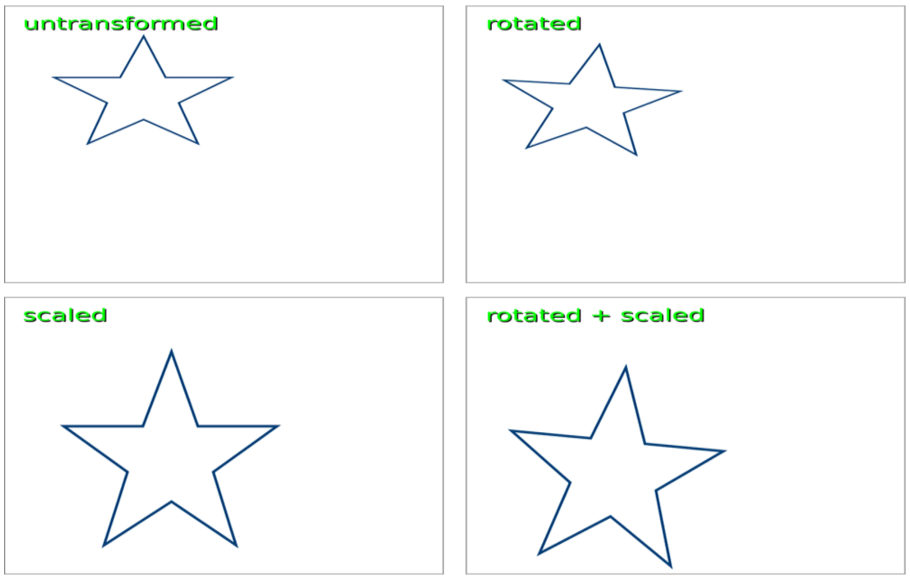
0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of Instructor

Assignment No. 4 : 2D transformations

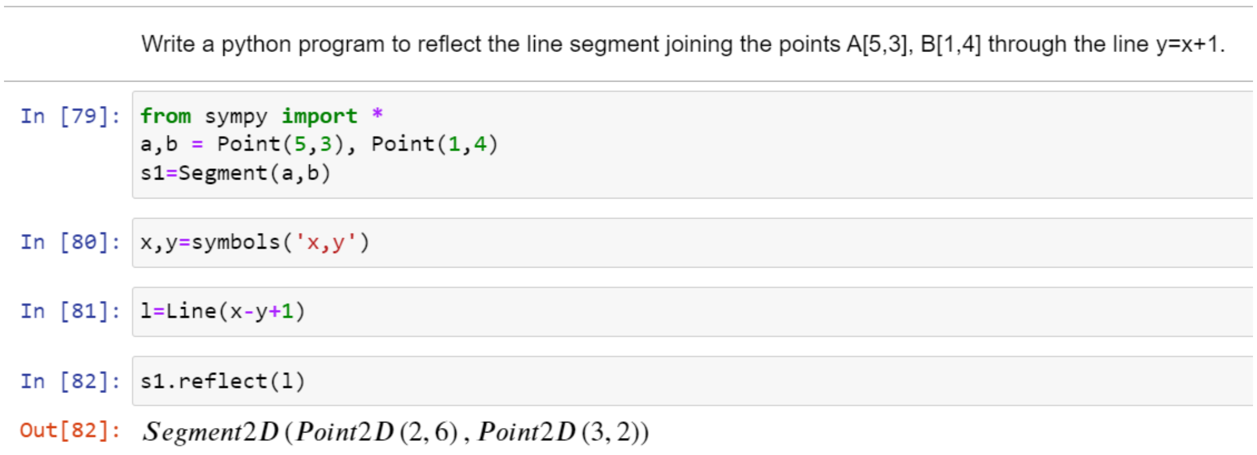
Transformations

Processing’s transform functions provide a convenient means of dealing with complex geometric transformations. Such operations include scaling, rotation, shearing, and translation. Suppose that you wish to rotate and scale the star shape depicted at the top-left, such that it results in the bottom-right:



Attributes of some objects in sympy module have attributes reflect, rotate and scale.

Example to reflect:



Write a python program to reflect the line segment joining the points A[5,3], B[1,4] through the line y=x+1.

```
In [79]: from sympy import *
         a,b = Point(5,3), Point(1,4)
         s1=Segment(a,b)

In [80]: x,y=symbols('x,y')

In [81]: l=Line(x-y+1)

In [82]: s1.reflect(l)
Out[82]: Segment2D(Point2D(2,6), Point2D(3,2))
```

Example to rotate:

Write a Python program to a polygon with vertices (0,0), (2,0), (2,3), (1,6). Rotate it by 180 deg.

```
In [90]: from sympy import *
```

```
In [91]: a,b,c,d=Point(0,0), Point(2,0), Point(2,3), Point(1,6)
```

```
In [92]: poly=Polygon(a,b,c,d)
```

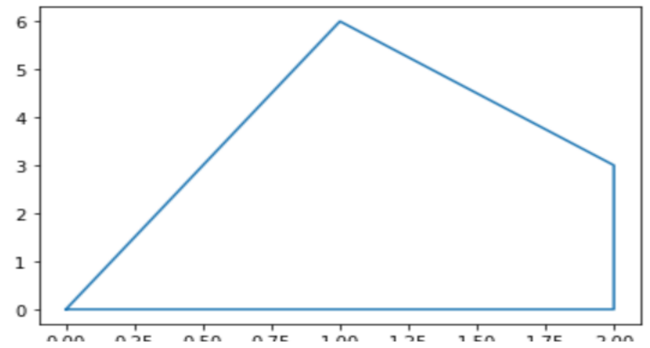
```
In [93]: from math import *
         poly.rotate(pi)
```

Out[93]:

$$Polygon\left(Point2D\left(0,0\right),Point2D\left(-2,\frac{244929359829471}{1000000000000000000000000000000}\right),Point2D\left(-2,-3\right),Point2D\left(-1,-6\right)\right)$$

```
In [94]: x1=[0,2,2,1,0]
         y1=[0,0,3,6,0]
         from matplotlib.pyplot import *
         plot(x1,y1,label="Polygon ABCD")
```

```
Out[94]: [<matplotlib.lines.Line2D at 0x16c7d842d30>]
```



```
In [95]: p1=poly.rotate(pi)
```

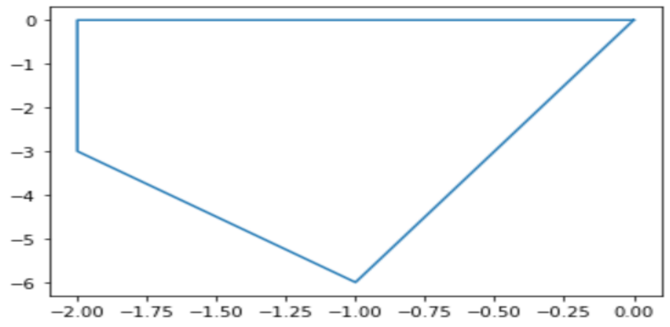
```
In [96]: v=p1.vertices
```

```
In [97]: x1=[]
         y1=[]
```

```
In [98]: for i in range(0,4):
             x1.append(v[i].x)
             y1.append(v[i].y)
         x1.append(x1[0])
         y1.append(y1[0])
```

```
In [99]: from matplotlib.pyplot import *
         plot(x1,y1,label="Rotated Polygon ABCD")
```

```
Out[99]: [<matplotlib.lines.Line2D at 0x16c7e8f6f40>]
```



Example of scaling: Scaling in X and Y direction by 2 and 1/3 units respectively.

```
In [1]: from sympy import *

In [2]: a=Point([1,2])

In [6]: b=Point([3,-2])

In [7]: s=Segment(a,b)

In [8]: s1=s.scale(2,1/3)

In [9]: s1
```

Out[9]: $$Segment2D\left(Point2D\left(2, \frac{2}{3}\right), Point2D\left(6, -\frac{2}{3}\right)\right)$$

Assignment 4

1. Reflect the line segment joining the points A[5, 3] and B[1, 4] through the line y = x + 1.
2. Reflect the line segment joining the points A[-1, 3] and B[9, 1] through the line y = 2x-3.
3. Reflect the line segment joining the points A[4, 2] and B[3,-1] through the line y = 3x+5.
4. Draw a polygon with vertices (0, 0), (2, 0), (2, 3) and (1, 6) and reflect it through the line y = 2x-3.
5. Draw a polygon with vertices (0, 0), (2, 0), (2, 3) and (1, 6) and rotate it by 180◦.
6. Rotate the line segment by 180◦ having end points (1, 0) and (2,−1).
7. Draw rectangle with vertices [1, 0], [2, 1], [1, 2] and [0, 1], its rotation about the origin by $\pi/2$ radians.
8. Scale the line segment between the points A[5,−2] & B[4, 3] in X−coordinate by factor 2.
9. Scale the triangle ABC, where A(2, 0), B(2, 3) and C(1, 6), in Y-coordinate by factor 3.
10. Uniform scaling the triangle ABC, where A(2, 0), B(2, 3) and C(1, 6), by factor 2/3.

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of Instructor

Assignment No. 5 – Concatenated transformation matrix

Write Python code to find concatenated transformation matrix for the following sequence transformations and apply it on the given object.

1. X= point P[3, 8]
   (I) Refection through X−axis.
   (II) Scaling in X−coordinate by factor 6.
   (III) Rotation about origin through an angle 30◦.
   (IV) Reflection through the line y = −x.

2. X= the line segment between the points A[4,−1] & B[3, 0]

   (I) Shearing in X direction by 9 units.
   (II) Rotation about origin through an angle π.
   (III) Scaling in X− coordinate by 2 units.
   (IV) Reflection through the line y = x.

3. X= the line segment between the points A[2,−1] & B[5, 4]

   (I) Rotation about origin through an angle π.
   (II) Scaling in X−coordinate by 3 units.
   (III) Shearing in X direction by 6 units.
   (IV) Reflection through the line y = x.

4. X= the line segment between the points A[2,−1] & B[5, 4]

   (I) Rotation about origin through an angle π.
   (II) Scaling in X−coordinate by 3 units.
   (III) Translation in X-direction and Y-directions by -3 and 2 units respectively.

5. X = the point P[4,−2]

   (I) Refection through Y−axis.
   (II) Scaling in X−coordinate by factor 7.
   (III) Shearing in Y direction by 3 units
   (IV) Reflection through the X-axis.

6. X = the segment between the points A[-1,5] and B[4, 3]

   (I) Rotation about origin through an angle 60◦.
   (II) Scaling in X−coordinate by 7 units.
   (III) Uniform scaling by 4 units.
   (IV) Translation in Y-direction by 5 units.

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of Instructor

Solving Linear Programming Problems using PuLP method

PuLP modeling process has the following steps for solving LP problems:

1. Initialize Model.
2. Define Decision Variable.
3. Define Objective Function.
4. Define the Constraints.
5. Solve Model.

The first step is to initialize an instance of LpProblem to represent your model:

```
# Create the model
model = LpProblem(name="small-problem", sense=LpMaximize)
```

You use the sense parameter to choose whether to perform minimization (`LpMinimize` or 1, which is the default) or maximization (LpMaximize or -1). This choice will affect the result of your problem.

Once that you have the model, you can define the decision variables as instances of the `LpVariable` class:

```
Python
# Initialize the decision variables
x = LpVariable(name="x", lowBound=0)
y = LpVariable(name="y", lowBound=0)
```

You need to provide a lower bound with lowBound=0 because the default value is negative infinity. The parameter upBound defines the upper bound, but you can omit it here because it defaults to positive infinity.

The optional parameter cat defines the category of a decision variable. If you're working with continuous variables, then you can use the default value "Continuous".

You can use the variables x and y to create other PuLP objects that represent linear expressions and constraints:

```
>>> expression = 2 * x + 4 * y
>>> type(expression)
<class 'pulp.pulp.LpAffineExpression'>

>>> constraint = 2 * x + 4 * y >= 8
>>> type(constraint)
<class 'pulp.pulp.LpConstraint'>
```

Having this in mind, the next step is to create the constraints and objective function as well as to assign them to your model. You don't need to create lists or matrices. Just write Python expressions and use the += operator to append them to the model:

```
# Add the constraints to the model
model += (2 * x + y <= 20, "red_constraint")
model += (4 * x - 5 * y >= -10, "blue_constraint")
model += (-x + 2 * y >= -2, "yellow_constraint")
model += (-x + 5 * y == 15, "green_constraint")
```

In the above code, you define tuples that hold the constraints and their names. LpProblem allows you to add constraints to a model by specifying them as tuples. The first element is a LpConstraint instance. The second element is a human-readable name for that constraint.

Setting the objective function is very similar:

```
# Add the objective function to the model
obj_func = x + 2 * y
model += obj_func
```

Alternatively, you can use a shorter notation:

```
# Add the objective function to the model
model += x + 2 * y
```

Now you have the objective function added and the model defined.

Note: You can append a constraint or objective to the model with the operator += because its class, LpProblem, implements the special method .__iadd__(), which is used to specify the behavior of +=.

You can now see the full definition of this model:

```
>>> model
small-problem:
MAXIMIZE
1*x + 2*y + 0
SUBJECT TO
red_constraint: 2 x + y <= 20

blue_constraint: 4 x - 5 y >= -10

yellow_constraint: - x + 2 y >= -2

green_constraint: - x + 5 y = 15

VARIABLES
x Continuous
y Continuous
```

Finally, you're ready to solve the problem. You can do that by calling .solve() on your model object. If you want to use the default solver (CBC), then you don't need to pass any arguments:

```
# Solve the problem
status = model.solve()
```

.solve() calls the underlying solver, modifies the model object, and returns the integer status of the solution, which will be 1 if the optimum is found. For the rest of the status codes, see `LpStatus[]`.

You can get the optimization results as the attributes of `model`. The function `value()` and the corresponding method `.value()` return the actual values of the attributes:

```
>>> print(f"status: {model.status}, {LpStatus[model.status]}")
status: 1, Optimal

>>> print(f"objective: {model.objective.value()}")
objective: 16.8181817

>>> for var in model.variables():
```

```
...     print(f"{var.name}: {var.value()}")
...
x: 7.7272727
y: 4.5454545
```

model.objective holds the value of the objective function, model.constraints contains the values of the slack variables, and the objects x and y have the optimal values of the decision variables. model.variables() returns a list with the decision variables.

<u>Assignment 6 – Solving LPP using PuLP module</u>

Write a python code to solve the following LPP by using pulp module.

1. Max $Z = 3x1 + 5x2 + 4x3$
   subject to $2x1 + 3x2 \leq 8$
   $2x2 + 5x3 \leq 10$
   $3x1 + 2x2 + 4x3 \leq 15$
   $x1 \geq 0, x2 \geq 0, x3 \geq 0$.
2. Min $Z = 3.5x + 2y$
   subject to $x + y \geq 5$
   $x \geq 4$
   $y \leq 2$
   $x \geq 0, y \geq 0$.
3. Max $Z = 150x + 75y$
   subject to $4x + 6y \leq 24$
   $5x + 3y \leq 15$
   $x \geq 0, y \geq 0$.
4. Max $Z = 4x + y + 3z + 5w$
   subject to $4x + 6y - 5z - 4w \geq -20$
   $-8x - 3y + 3z + 2w \leq 20$
   $x + y \leq 11$
   $x \geq 0, y \geq 0, z \geq 0, w \geq 0$.
5. Max $Z = 5x + 3y$
   subject to $x + y \leq 7$
   $2x + 5y \leq 1$
   $x \geq 0, y \geq 0$.

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of Instructor

Assignment No. 7 : Solving LPP using optimize package

SciPy is an open-source Python library dedicated to scientific computation.
The **optimize package** in SciPy provides several common optimization algorithms such as least squares, minimization, curve fitting, etc.

optimize.linprog() function

The **optimize.linprog()** function is from the domain of linear programming, which minimizes a linear objective function subject to linear equality and inequality constraints. Inequalities are of ≤ type.

Example

Linear programming in two variables

Syntax

```
scipy.optimize.linprog(c, A_ub = None, b_ub = None, A_eq = None, b_eq = None,
                        bounds = None, method = 'interior-point', callback = None,
                       options = None, x0 = None)
```

Parameters

optimize.linprog() accepts the following parameters:

- **c**: This is a one-dimensional array representing the coefficients of the linear objective function.
- **A_ub**: This is a two-dimensional array representing the inequality constraint matrix. Each row of the matrix represents the coefficients of a linear inequality.
- **b_ub**: This is a one-dimensional array representing the inequality constraint vector.
- **A_eq**: This is a two-dimensional array representing the equality constraint matrix. Each row of the matrix represents the coefficients of a linear equality.
- **b_eq**: This is a one-dimensional array representing the equality constraint vector.
- **bounds**: This is a sequence of the (min,max) pair defining the minimum and maximum value of the decision variable.
- **callback**: This is an optional function argument. It is invoked on every iteration.
- **method**: This is the algorithm used to solve the standard form problem.
- **options**: This is a dictionary of solver options.
- **x0**: This is a one-dimensional array that represents the guess values of the decision variables.

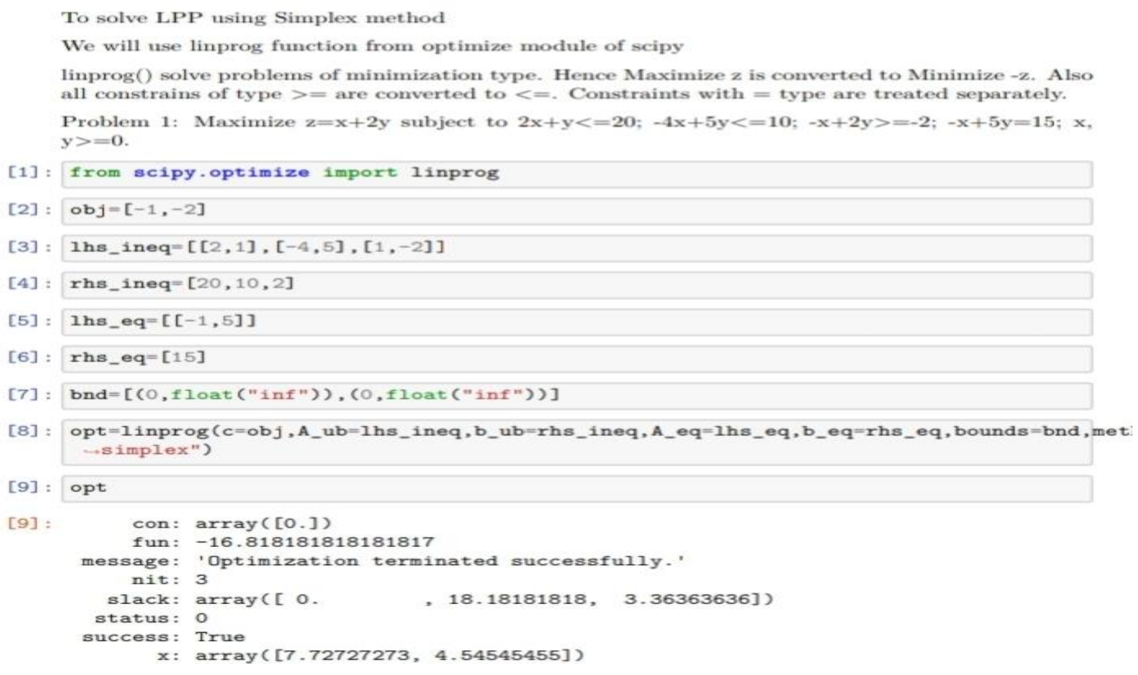
Return value

This function returns a solution represented by the OptimizeResult object. As part of the object, the following components are returned.

- **x**: This contains the values of the decision variables that minimize the objective function while meeting the defined constraints.
- **fun**: This tells the optimal value of the objective function.
- **slack**: This tells the (nominally positive) values of the slack variables. **Slack variables** are the differences between the values of the left and right sides of the constraints.
- **con**: This represents the equality constraints residuals.
- **status**: This is a value between 0-4 that represents the exit status of the algorithm.
- **success**: This tells us that the algorithm has found an optimal solution.
- **nit**: This tells us the total number of iterations in all phases.
- **message**: This displays the message produced on the algorithm's termination.

Code

Let's see how we can use the optimize.linprog() function to solve the following minimization problem.



To solve LPP using Simplex method

We will use linprog function from optimize module of scipy

linprog() solve problems of minimization type. Hence Maximize z is converted to Minimize -z. Also all constrains of type >= are converted to <=. Constraints with = type are treated separately.

Problem 1: Maximize z=x+2y subject to 2x+y<=20; -4x+5y<=10; -x+2y>=-2; -x+5y=15; x, y>=0.

```
[1]: from scipy.optimize import linprog
[2]: obj=[-1,-2]
[3]: lhs_ineq=[[2,1],[-4,5],[1,-2]]
[4]: rhs_ineq=[20,10,2]
[5]: lhs_eq=[[-1,5]]
[6]: rhs_eq=[15]
[7]: bnd=[(0,float("inf")),(0,float("inf"))]
[8]: opt=linprog(c=obj,A_ub=lhs_ineq,b_ub=rhs_ineq,A_eq=lhs_eq,b_eq=rhs_eq,bounds=bnd,met
      ↪simplex")
[9]: opt
[9]:      con: array([0.])
          fun: -16.818181818181817
      message: 'Optimization terminated successfully.'
          nit: 3
        slack: array([ 0.        , 18.18181818,  3.36363636])
       status: 0
      success: True
            x: array([7.72727273, 4.54545455])
```

Assignment 7

Write a python code to solve the following LPP by using optimize module.

1. Max $Z = 3x1 + 5x2 + 4x3$
   subject to $2x1 + 3x2 \leq 8$
   $2x2 + 5x3 \leq 10$
   $3x1 + 2x2 + 4x3 \leq 15$
   $x1 \geq 0, x2 \geq 0, x3 \geq 0$.

2. Min $Z = 3.5x + 2y$
   subject to $x + y \geq 5$
   $x \geq 4$
   $y \leq 2$
   $x \geq 0, y \geq 0$.

3. Max $Z = 150x + 75y$
   subject to $4x + 6y \leq 24$
   $5x + 3y \leq 15$
   $x \geq 0, y \geq 0$.

4. Max $Z = 4x + y + 3z + 5w$
   subject to $4x + 6y - 5z - 4w \geq -20$
   $-8x - 3y + 3z + 2w \leq 20$
   $x + y \leq 11$
   $x \geq 0, y \geq 0, z \geq 0, w \geq 0$.

5. Max $Z = 5x + 3y$
   subject to $x + y \leq 7$
   $2x + 5y \leq 1$
   $x \geq 0, y \geq 0$.

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of Instructor

Assignment No. 8 – Concatenated transformation matrix (3D)

Write Python code to find concatenated transformation matrix for the following sequence transformations and apply it on the given object.

1. X= point P[3, 8,-1]
   (I) Rotation about X−axis through an angle 30◦.
   (II) Scaling in X−coordinate by factor 6.
   (III) Shearing in y direction proportional to z-coordinate by 3 units
   (IV) Reflection through the plane y = 0.

2. X= the line segment between the points A[4,−1,1] & B[3,5, 0]

   (I) Shearing in X direction proportional to y coordinate by 9 units.
   (II) Rotation about Y-axis through an angle π.
   (III) Scaling in X and Y coordinate by 2 units.
   (IV) Reflection through the plane z=0.

3. X= the line segment between the points A[7, 2,−1] & B[5, 4, -1]

   (I) Rotation about Z-axis through an angle π.
   (II) Uniform scaling by 3 units.
   (III) Shearing in X direction proportional to Y and Z coordinates by 2, 6 units.

4. X= the line segment between the points A[2,−1,0] & B[5, 4,6]. Reflect the object X through (i) the plane z=5. (ii) x=3 (iii) y=10

5. X = the segment between the points A[2,−1,0] & B[5, 4,6]. Rotate X about (i) local X-axis at (1,2,3) (ii) local Y-axis through (5,0,2), (iii) local Z-axis passing through (1,4,7)

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2: Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]