APSSDC

Andhra Pradesh State Skill Development Corporation

(https://apssdc.in)

Data Analysis Using Python Day10

Day Objectives:

Data Preprocessing

- Normalizing Data
- Data Imputation

Introduction to Visualization and Python packages

- Matplotlib history
- Introduction to plotting
- Line Plot
- Scatter Plot
- Bar Graph
- Histogram
- Pie Chart
- Box Plot

[Advertisments Dataset Link (https://raw.githubusercontent.com/AP-State-Skill-Development-Corporation/Datasets/master/Advertising.csv)](https://raw.githubusercontent.com/AP-State-Skill-Development-Corporation/Datasets/master/Advertising.csv)

In [3]:

```
import pandas as pd
import numpy as np
```

In [49]:

```
df = pd.read_csv("https://raw.githubusercontent.com/AP-State-Skill-Development-Corporation/

df.head()
```

Out[49]:

	TV	radio	newspaper	sales
1	230.1	37.8	69.2	22.1
2	44.5	39.3	45.1	10.4
3	17.2	45.9	69.3	9.3
4	151.5	41.3	58.5	18.5
5	180.8	10.8	58.4	12.9

In [50]:

```
df.columns
```

Out[50]:

```
Index(['TV', 'radio', 'newspaper', 'sales'], dtype='object')
```

In [4]:

```
from sklearn.preprocessing import Normalizerzer
```

In [5]:

```
norm = Normalizer()
```

In [6]:

```
norm = norm.fit_transform(df)

norm[:5,:]
```

Out[6]:

```
array([[0.94211621, 0.15476746, 0.28333091, 0.09048574],
       [0.59113524, 0.52205877, 0.59910561, 0.13815296],
       [0.20142628, 0.53752711, 0.81156054, 0.10891072],
       [0.89863215, 0.24497365, 0.34699657, 0.10973396],
       [0.94788063, 0.05662119, 0.30617383, 0.06763086]])
```

Data Imputation

ffill, bfill, interpolation

df.fillna()

- mean
- median
- most Frequent
- constan

In [8]:

```
arr = np.array([[1,2,3,4,np.nan], [1,3,8,np.nan,15], [np.nan, 5, 15, 66, 25], [5, 6, 8, 9,5
```

In [9]:

```
arr
```

Out[9]:

```
array([[ 1.,  2.,  3.,  4., nan],
       [ 1.,  3.,  8., nan, 15.],
       [nan,  5., 15., 66., 25.],
       [ 5.,  6.,  8.,  9.,  5.]])
```

In [10]:

```
from sklearn.impute import SimpleImputer
```

In [12]:

```
mean = SimpleImputer(strategy = 'mean')
```

In [13]:

```
mean.fit_transform(arr)
```

Out[13]:

```
array([[ 1.        ,  2.        ,  3.        ,  4.        , 15.        ],
       [ 1.        ,  3.        ,  8.        , 26.33333333, 15.        ],
       [ 2.33333333,  5.        , 15.        , 66.        , 25.        ],
       [ 5.        ,  6.        ,  8.        ,  9.        ,  5.        ]])
```

In [14]:

```
median = SimpleImputer(strategy = 'median')

median.fit(arr)
```

Out[14]:

```
SimpleImputer(strategy='median')
```

In [15]:

```
median.n_features_in_
```

Out[15]:

```
5
```

In [19]:

```
median.get_params()
```

Out[19]:

```
{'add_indicator': False,
 'copy': True,
 'fill_value': None,
 'missing_values': nan,
 'strategy': 'median',
 'verbose': 0}
```

In [20]:

```
median.transform(arr)
```

Out[20]:

```
array([[ 1.,  2.,  3.,  4., 15.],
       [ 1.,  3.,  8.,  9., 15.],
       [ 1.,  5., 15., 66., 25.],
       [ 5.,  6.,  8.,  9.,  5.]])
```

In [21]:

```
mode = SimpleImputer(strategy = 'most_frequent')
```

In [22]:

```
mode.fit_transform(arr)
```

Out[22]:

```
array([[ 1.,  2.,  3.,  4.,  5.],
       [ 1.,  3.,  8.,  4., 15.],
       [ 1.,  5., 15., 66., 25.],
       [ 5.,  6.,  8.,  9.,  5.]])
```

In [23]:

```
con = SimpleImputer(strategy = 'constant', fill_value=0)

con.fit_transform(arr)
```

Out[23]:

```
array([[ 1.,  2.,  3.,  4.,  0.],
       [ 1.,  3.,  8.,  0., 15.],
       [ 0.,  5., 15., 66., 25.],
       [ 5.,  6.,  8.,  9.,  5.]])
```

In [24]:

```
df = pd.DataFrame(arr)

df.dropna()
```

Out[24]:

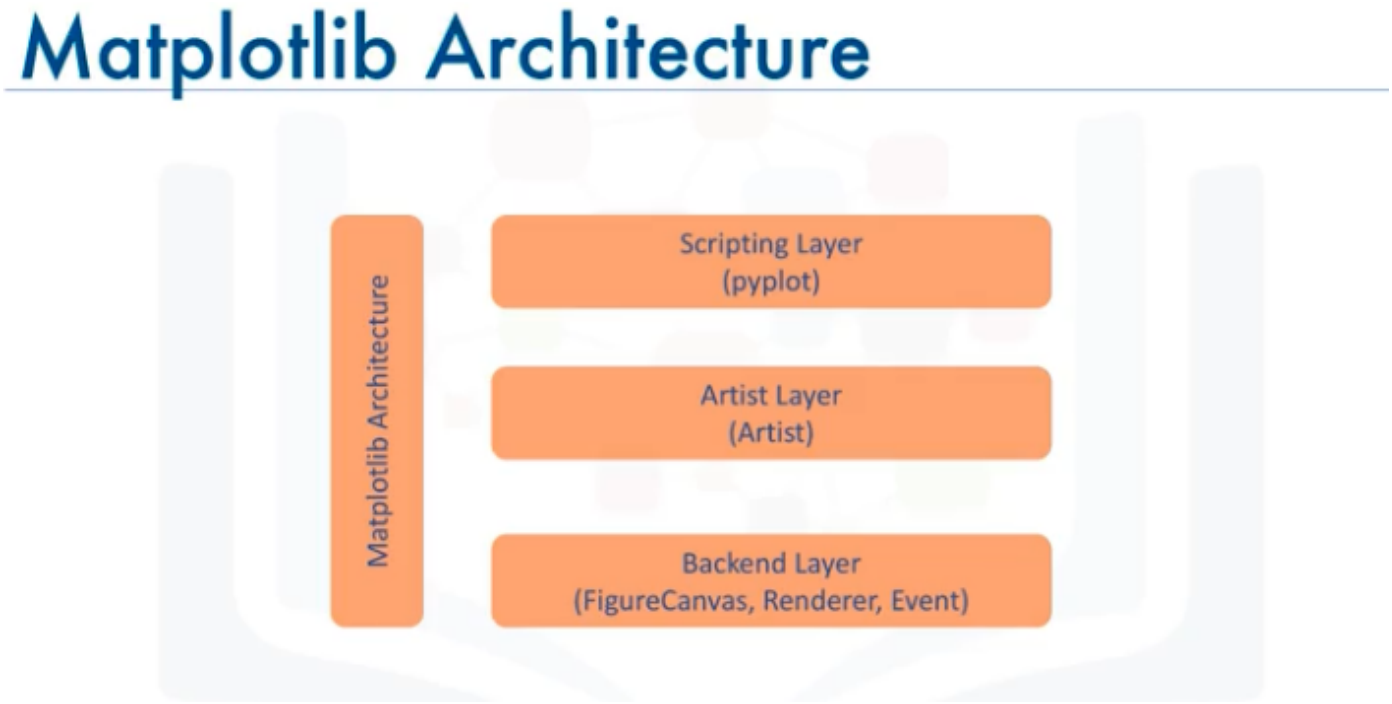
	0	1	2	3	4
3	5.0	6.0	8.0	9.0	5.0

Data Visualization

Matplotlib

Seaborn

John Hunter



In [25]:

```
import matplotlib.pyplot as plt
```

Line Plot

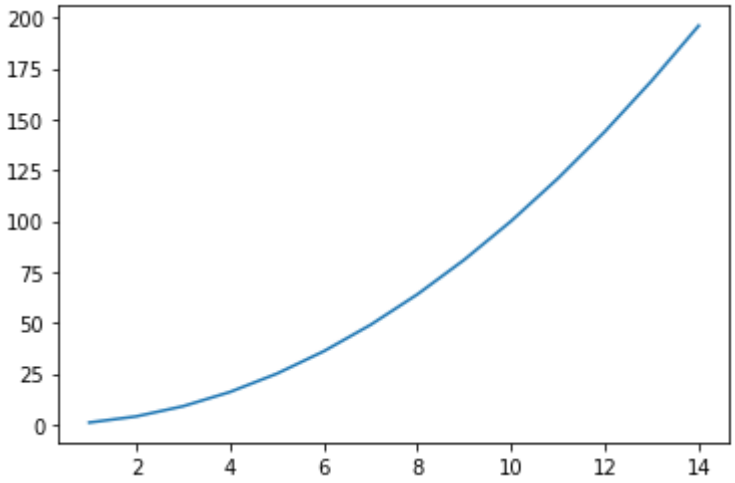
In [27]:

```
x = np.arange(1, 15)
y = x ** 2

plt.plot(x, y)
```

Out[27]:

```
[<matplotlib.lines.Line2D at 0x2aa4ea06c70>]
```

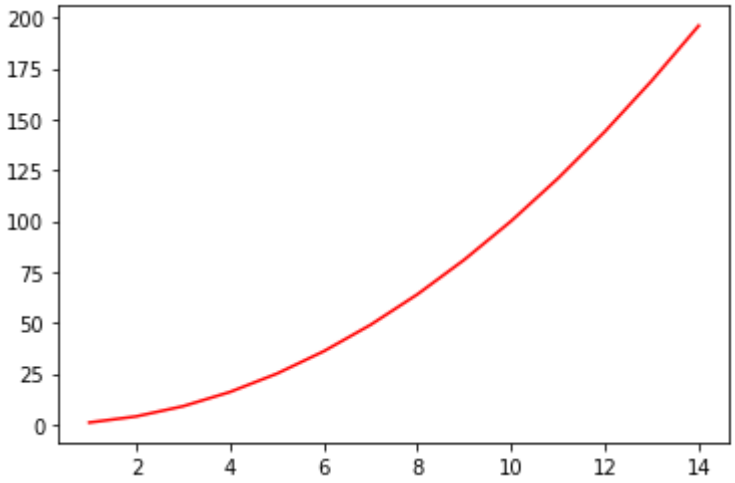


In [28]:

```
plt.plot(x, y, c = 'r')
```

Out[28]:

```
[<matplotlib.lines.Line2D at 0x2aa4f631ac0>]
```



In [29]:

```
help(plt.plot)
```

```
Help on function plot in module matplotlib.pyplot:

plot(*args, scalex=True, scaley=True, data=None, **kwargs)
    Plot y versus x as lines and/or markers.

    Call signatures::

        plot([x], y, [fmt], *, data=None, **kwargs)
        plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)

    The coordinates of the points or line nodes are given by *x*, *y*.

    The optional parameter *fmt* is a convenient way for defining basic
    formatting like color, marker and linestyle. It's a shortcut string
    notation described in the *Notes* section below.

    >>> plot(x, y)        # plot x and y using default line style and color
    >>> plot(x, y, 'bo')  # plot x and y using blue circle markers
    >>> plot(y)           # plot y using x as index array 0..N-1
    >>> plot(y, 'r+')     # ditto, but with red plusses

    You can use `.Line2D` properties as keyword arguments for more
    control on the appearance. Line properties and *fmt* can be mixed.
    The following two calls yield identical results:

    >>> plot(x, y, 'go--', linewidth=2, markersize=12)
    >>> plot(x, y, color='green', marker='o', linestyle='dashed',
    ...      linewidth=2, markersize=12)

    When conflicting with *fmt*, keyword arguments take precedence.


    **Plotting labelled data**

    There's a convenient way for plotting objects with labelled data (i.e.
    data that can be accessed by index ``obj['y']``). Instead of giving
    the data in *x* and *y*, you can provide the object in the *data*
    parameter and just give the labels for *x* and *y*::

    >>> plot('xlabel', 'ylabel', data=obj)

    All indexable objects are supported. This could e.g. be a `dict`, a
    `pandas.DataFame` or a structured numpy array.


    **Plotting multiple sets of data**

    There are various ways to plot multiple sets of data.

    - The most straight forward way is just to call `plot` multiple times.
      Example:

      >>> plot(x1, y1, 'bo')
      >>> plot(x2, y2, 'go')
```

```
- Alternatively, if your data is already a 2d array, you can pass it
  directly to *x*, *y*. A separate data set will be drawn for every
  column.

  Example: an array ``a`` where the first column represents the *x*
  values and the other columns are the *y* columns::

  >>> plot(a[0], a[1:])

- The third way is to specify multiple sets of *[x]*, *y*, *[fmt]*
  groups::

  >>> plot(x1, y1, 'g^', x2, y2, 'g-')

  In this case, any additional keyword argument applies to all
  datasets. Also this syntax cannot be combined with the *data*
  parameter.

By default, each line is assigned a different style specified by a
'style cycle'. The *fmt* and line property parameters are only
necessary if you want explicit deviations from these defaults.
Alternatively, you can also change the style cycle using
:rc:`axes.prop_cycle`.


Parameters
----------
x, y : array-like or scalar
    The horizontal / vertical coordinates of the data points.
    *x* values are optional and default to `range(len(y))`.

    Commonly, these parameters are 1D arrays.

    They can also be scalars, or two-dimensional (in that case, the
    columns represent separate data sets).

    These arguments cannot be passed as keywords.

fmt : str, optional
    A format string, e.g. 'ro' for red circles. See the *Notes*
    section for a full description of the format strings.

    Format strings are just an abbreviation for quickly setting
    basic line properties. All of these and more can also be
    controlled by keyword arguments.

    This argument cannot be passed as keyword.

data : indexable object, optional
    An object with labelled data. If given, provide the label names to
    plot in *x* and *y*.

    .. note::
        Technically there's a slight ambiguity in calls where the
        second label is a valid *fmt*. `plot('n', 'o', data=obj)`
        could be `plt(x, y)` or `plt(y, fmt)`. In such cases,
        the former interpretation is chosen, but a warning is issued.
        You may suppress the warning by adding an empty format string
        `plot('n', 'o', '', data=obj)`.

Other Parameters
```

```
    ----------------
    scalex, scaley : bool, optional, default: True
        These parameters determined if the view limits are adapted to
        the data limits. The values are passed on to `autoscale_view`.

    **kwargs : `.Line2D` properties, optional
        *kwargs* are used to specify properties like a line label (for
        auto legends), linewidth, antialiasing, marker face color.
        Example::

        >>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1', linewidth=2)
        >>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')

        If you make multiple lines with one plot command, the kwargs
        apply to all those lines.

        Here is a list of available `.Line2D` properties:

        Properties:
        agg_filter: a filter function, which takes a (m, n, 3) float array a
nd a dpi value, and returns a (m, n, 3) array
        alpha: float or None
        animated: bool
        antialiased or aa: bool
        clip_box: `.Bbox`
        clip_on: bool
        clip_path: Patch or (Path, Transform) or None
        color or c: color
        contains: callable
        dash_capstyle: {'butt', 'round', 'projecting'}
        dash_joinstyle: {'miter', 'round', 'bevel'}
        dashes: sequence of floats (on/off ink in points) or (None, None)
        data: (2, N) array or two 1D arrays
        drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'ste
ps-post'}, default: 'default'
        figure: `.Figure`
        fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
        gid: str
        in_layout: bool
        label: object
        linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq),
...}
        linewidth or lw: float
        marker: marker style
        markeredgecolor or mec: color
        markeredgewidth or mew: float
        markerfacecolor or mfc: color
        markerfacecoloralt or mfcalt: color
        markersize or ms: float
        markevery: None or int or (int, int) or slice or List[int] or float
or (float, float)
        path_effects: `.AbstractPathEffect`
        picker: float or callable[[Artist, Event], Tuple[bool, dict]]
        pickradius: float
        rasterized: bool or None
        sketch_params: (scale: float, length: float, randomness: float)
        snap: bool or None
        solid_capstyle: {'butt', 'round', 'projecting'}
        solid_joinstyle: {'miter', 'round', 'bevel'}
        transform: `matplotlib.transforms.Transform`
        url: str
```

```
    visible: bool
    xdata: 1D array
    ydata: 1D array
    zorder: float

Returns
-------
lines
    A list of `.Line2D` objects representing the plotted data.

See Also
--------
scatter : XY scatter plot with markers of varying size and/or color (
    sometimes also called bubble chart).

Notes
-----
**Format Strings**

A format string consists of a part for color, marker and line::

    fmt = '[marker][line][color]'

Each of them is optional. If not provided, the value from the style
cycle is used. Exception: If ``line`` is given, but no ``marker``,
the data will be a line without markers.

Other combinations such as ``[color][marker][line]`` are also
supported, but note that their parsing may be ambiguous.

**Markers**

=============    ===============================
character        description
=============    ===============================
``'.'``          point marker
``','``          pixel marker
``'o'``          circle marker
``'v'``          triangle_down marker
``'^'``          triangle_up marker
``'<'``          triangle_left marker
``'>'``          triangle_right marker
``'1'``          tri_down marker
``'2'``          tri_up marker
``'3'``          tri_left marker
``'4'``          tri_right marker
``'s'``          square marker
``'p'``          pentagon marker
``'*'``          star marker
``'h'``          hexagon1 marker
``'H'``          hexagon2 marker
``'+'``          plus marker
``'x'``          x marker
``'D'``          diamond marker
``'d'``          thin_diamond marker
``'|'``          vline marker
``'_'``          hline marker
=============    ===============================

**Line Styles**
```

=============   ================================
character   description
=============   ================================
``'-'``   solid line style
``'--'``   dashed line style
``'-.'``   dash-dot line style
``':'``   dotted line style
=============   ================================

Example format strings::

    'b'   # blue markers with default shape
    'or'   # red circles
    '-g'   # green solid line
    '--'   # dashed line with default color
    '^k:' # black triangle_up markers connected by a dotted line

Colors

The supported color abbreviations are the single letter codes

=============   ================================
character   color
=============   ================================
``'b'``   blue
``'g'``   green
``'r'``   red
``'c'``   cyan
``'m'``   magenta
``'y'``   yellow
``'k'``   black
``'w'``   white
=============   ================================

and the ``'CN'`` colors that index into the default property cycle.

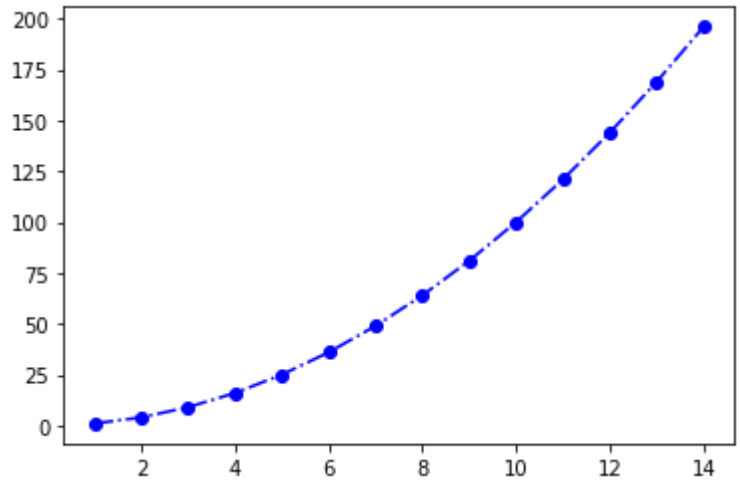
If the color is the only part of the format string, you can
additionally use any `matplotlib.colors` spec, e.g. full names
(``'green'``) or hex strings (``'#008000'``).

In [37]:

```
plt.plot(x, y, c = 'b',linewidth = 1.5, marker = 'o', linestyle = '-.' )
```

Out[37]:

```
[<matplotlib.lines.Line2D at 0x2aa51dcdc70>]
```



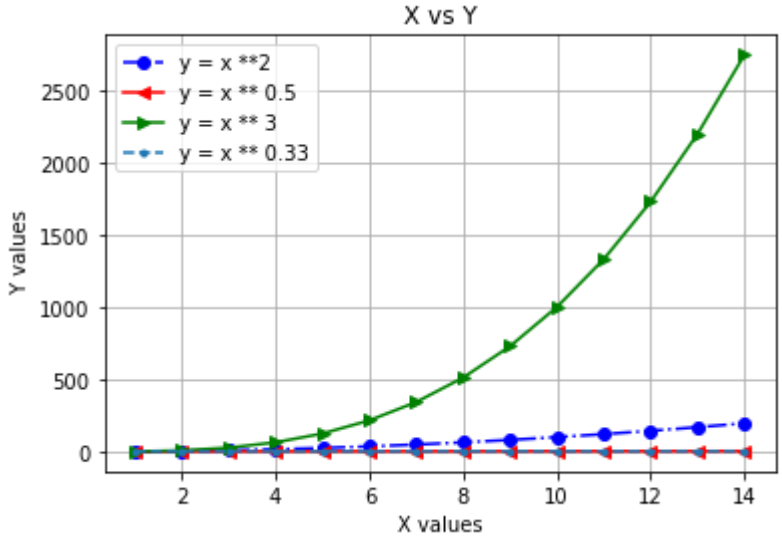
In [39]:

```
plt.plot(x, y, c = 'b',linewidth = 1.5, marker = 'o', linestyle = '-.' )
plt.plot(x, x ** 0.5, c = 'r',linewidth = 1.5, marker = '<', linestyle = '-' )
plt.plot(x, x ** 3, c = 'g',linewidth = 1.5, marker = '>' )
plt.plot(x, x ** 0.33,linewidth = 1.5, marker = '.', linestyle = '--' )

plt.legend(['y = x **2', 'y = x ** 0.5', 'y = x ** 3', 'y = x ** 0.33'])
plt.grid()
plt.xlabel('X values')
plt.ylabel(' Y values')
plt.title('X vs Y')
```

Out[39]:

```
Text(0.5, 1.0, 'X vs Y')
```



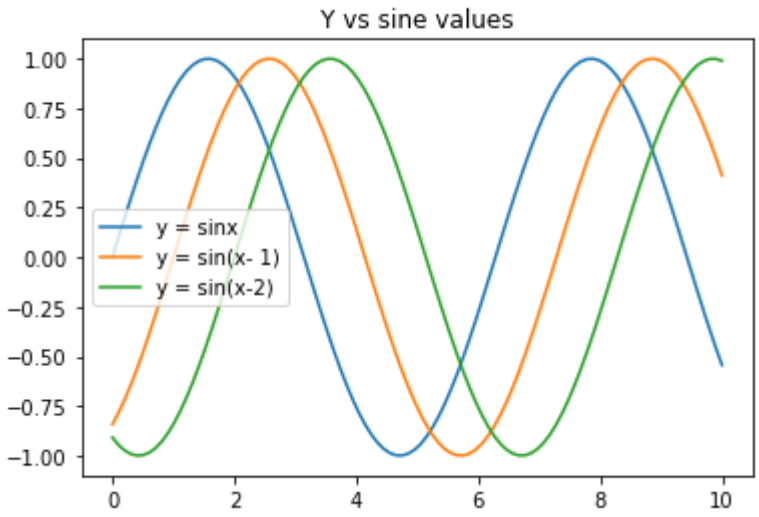
In [42]:

```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), label ='y = sinx')
plt.plot(x, np.sin(x - 1), label = 'y = sin(x- 1)')
plt.plot(x, np.sin(x - 2), label = 'y = sin(x-2)')
plt.title('Y vs sine values')
plt.legend(loc = 6)
```

Out[42]:

```
<matplotlib.legend.Legend at 0x2aa51f5f880>
```



In [41]:

```
help(plt.legend)
```

```
Help on function legend in module matplotlib.pyplot:

legend(*args, **kwargs)
    Place a legend on the axes.

    Call signatures::

        legend()
        legend(labels)
        legend(handles, labels)

    The call signatures correspond to three different ways how to use
    this method.

    **1. Automatic detection of elements to be shown in the legend**

    The elements to be added to the legend are automatically determined,
    when you do not pass in any extra arguments.

    In this case, the labels are taken from the artist. You can specify
    them either at artist creation or by calling the
    :meth:`~.Artist.set_label` method on the artist::

        line, = ax.plot([1, 2, 3], label='Inline label')
        ax.legend()

    or::

        line, = ax.plot([1, 2, 3])
        line.set_label('Label via method')
        ax.legend()

    Specific lines can be excluded from the automatic legend element
    selection by defining a label starting with an underscore.
    This is default for all artists, so calling `Axes.legend` without
    any arguments and without setting the labels manually will result in
    no legend being drawn.


    **2. Labeling existing plot elements**

    To make a legend for lines which already exist on the axes
    (via plot for instance), simply call this function with an iterable
    of strings, one for each legend item. For example::

        ax.plot([1, 2, 3])
        ax.legend(['A simple line'])

    Note: This way of using is discouraged, because the relation between
    plot elements and labels is only implicit by their order and can
    easily be mixed up.


    **3. Explicitly defining the elements in the legend**

    For full control of which artists have a legend entry, it is possible
```

```
    to pass an iterable of legend artists followed by an iterable of
    legend labels respectively::

        legend((line1, line2, line3), ('label1', 'label2', 'label3'))

    Parameters
    ----------
    handles : sequence of `.Artist`, optional
        A list of Artists (lines, patches) to be added to the legend.
        Use this together with *labels*, if you need full control on what
        is shown in the legend and the automatic mechanism described above
        is not sufficient.

        The length of handles and labels should be the same in this
        case. If they are not, they are truncated to the smaller length.

    labels : list of str, optional
        A list of labels to show next to the artists.
        Use this together with *handles*, if you need full control on what
        is shown in the legend and the automatic mechanism described above
        is not sufficient.

    Other Parameters
    ----------------

    loc : str or pair of floats, default: :rc:`legend.loc` ('best' for axe
s, 'upper right' for figures)
        The location of the legend.

        The strings
        ``'upper left', 'upper right', 'lower left', 'lower right'``
        place the legend at the corresponding corner of the axes/figure.

        The strings
        ``'upper center', 'lower center', 'center left', 'center right'``
        place the legend at the center of the corresponding edge of the
        axes/figure.

        The string ``'center'`` places the legend at the center of the axe
s/figure.

        The string ``'best'`` places the legend at the location, among the
nine
        locations defined so far, with the minimum overlap with other draw
n
        artists.  This option can be quite slow for plots with large amoun
ts of
        data; your plotting speed may benefit from providing a specific lo
cation.

        The location can also be a 2-tuple giving the coordinates of the l
ower-left
        corner of the legend in axes coordinates (in which case *bbox_to_a
nchor*
        will be ignored).

        For back-compatibility, ``'center right'`` (but no other location)
can also
        be spelled ``'right'``, and each "string" locations can also be gi
ven as a
        numeric value:
```

```
            ===============   =============
            Location String   Location Code
            ===============   =============
            'best'            0
            'upper right'     1
            'upper left'      2
            'lower left'      3
            'lower right'     4
            'right'           5
            'center left'     6
            'center right'    7
            'lower center'    8
            'upper center'    9
            'center'          10
            ===============   =============

    bbox_to_anchor : `.BboxBase`, 2-tuple, or 4-tuple of floats
        Box that is used to position the legend in conjunction with *loc*.
        Defaults to `axes.bbox` (if called as a method to `.Axes.legend`)
or
        `figure.bbox` (if `.Figure.legend`).  This argument allows arbitra
ry
        placement of the legend.

        Bbox coordinates are interpreted in the coordinate system given by
        `bbox_transform`, with the default transform
        Axes or Figure coordinates, depending on which ``legend`` is calle
d.

        If a 4-tuple or `.BboxBase` is given, then it specifies the bbox
        ``(x, y, width, height)`` that the legend is placed in.
        To put the legend in the best location in the bottom right
        quadrant of the axes (or figure)::

            loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)

        A 2-tuple ``(x, y)`` places the corner of the legend specified by
*loc* at
        x, y.  For example, to put the legend's upper right-hand corner in
the
        center of the axes (or figure) the following keywords can be use
d::

            loc='upper right', bbox_to_anchor=(0.5, 0.5)

    ncol : integer
        The number of columns that the legend has. Default is 1.

    prop : None or :class:`matplotlib.font_manager.FontProperties` or dict
        The font properties of the legend. If None (default), the current
        :data:`matplotlib.rcParams` will be used.

    fontsize : int or float or {'xx-small', 'x-small', 'small', 'medium',
'large', 'x-large', 'xx-large'}
        The font size of the legend. If the value is numeric the size will
be the
        absolute font size in points. String values are relative to the cu
rrent
        default font size. This argument is only used if *prop* is not spe
cified.
```

```
    numpoints : None or int
        The number of marker points in the legend when creating a legend
        entry for a `.Line2D` (line).
        Default is ``None``, which means using :rc:`legend.numpoints`.

    scatterpoints : None or int
        The number of marker points in the legend when creating
        a legend entry for a `.PathCollection` (scatter plot).
        Default is ``None``, which means using :rc:`legend.scatterpoints`.

    scatteryoffsets : iterable of floats
        The vertical offset (relative to the font size) for the markers
        created for a scatter plot legend entry. 0.0 is at the base the
        legend text, and 1.0 is at the top. To draw all markers at the
        same height, set to ``[0.5]``. Default is ``[0.375, 0.5, 0.3125]`
`.

    markerscale : None or int or float
        The relative size of legend markers compared with the originally
        drawn ones.
        Default is ``None``, which means using :rc:`legend.markerscale`.

    markerfirst : bool
        If *True*, legend marker is placed to the left of the legend labe
l.
        If *False*, legend marker is placed to the right of the legend
        label.
        Default is *True*.

    frameon : None or bool
        Whether the legend should be drawn on a patch (frame).
        Default is ``None``, which means using :rc:`legend.frameon`.

    fancybox : None or bool
        Whether round edges should be enabled around the `~.FancyBboxPatch
` which
        makes up the legend's background.
        Default is ``None``, which means using :rc:`legend.fancybox`.

    shadow : None or bool
        Whether to draw a shadow behind the legend.
        Default is ``None``, which means using :rc:`legend.shadow`.

    framealpha : None or float
        The alpha transparency of the legend's background.
        Default is ``None``, which means using :rc:`legend.framealpha`.
        If *shadow* is activated and *framealpha* is ``None``, the default
value is
        ignored.

    facecolor : None or "inherit" or color
        The legend's background color.
        Default is ``None``, which means using :rc:`legend.facecolor`.
        If ``"inherit"``, use :rc:`axes.facecolor`.

    edgecolor : None or "inherit" or color
        The legend's background patch edge color.
        Default is ``None``, which means using :rc:`legend.edgecolor`.
        If ``"inherit"``, use take :rc:`axes.edgecolor`.
```

```
    mode : {"expand", None}
        If *mode* is set to ``"expand"`` the legend will be horizontally
        expanded to fill the axes area (or `bbox_to_anchor` if defines
        the legend's size).

    bbox_transform : None or :class:`matplotlib.transforms.Transform`
        The transform for the bounding box (`bbox_to_anchor`). For a value
        of ``None`` (default) the Axes'
        :data:`~matplotlib.axes.Axes.transAxes` transform will be used.

    title : str or None
        The legend's title. Default is no title (``None``).

    title_fontsize: str or None
        The fontsize of the legend's title.  Default is the default fontsi
ze.

    borderpad : float or None
        The fractional whitespace inside the legend border, in font-size u
nits.
        Default is ``None``, which means using :rc:`legend.borderpad`.

    labelspacing : float or None
        The vertical space between the legend entries, in font-size units.
        Default is ``None``, which means using :rc:`legend.labelspacing`.

    handlelength : float or None
        The length of the legend handles, in font-size units.
        Default is ``None``, which means using :rc:`legend.handlelength`.

    handletextpad : float or None
        The pad between the legend handle and text, in font-size units.
        Default is ``None``, which means using :rc:`legend.handletextpad`.

    borderaxespad : float or None
        The pad between the axes and legend border, in font-size units.
        Default is ``None``, which means using :rc:`legend.borderaxespad`.

    columnspacing : float or None
        The spacing between columns, in font-size units.
        Default is ``None``, which means using :rc:`legend.columnspacing`.

    handler_map : dict or None
        The custom dictionary mapping instances or types to a legend
        handler. This `handler_map` updates the default handler map
        found at :func:`matplotlib.legend.Legend.get_legend_handler_map`.



    Returns
    -------
    legend : `~matplotlib.legend.Legend`

    Notes
    -----
    Not all kinds of artist are supported by the legend command. See
    :doc:`/tutorials/intermediate/legend_guide` for details.

    Examples
    --------
```

```
.. plot:: gallery/text_labels_and_annotations/legend.py
```

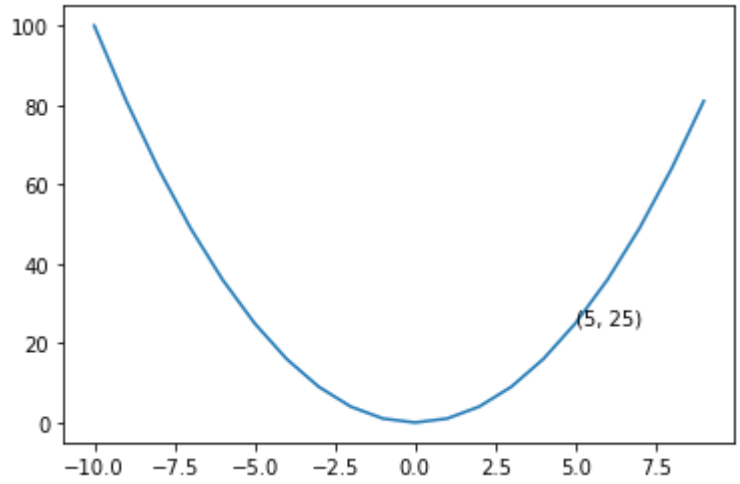
In [43]:

```
x = np.arange(-10, 10)
y = x ** 2

plt.plot(x, y)
plt.text(5, 25, "(5, 25)")
```

Out[43]:

```
Text(5, 25, '(5, 25)')
```



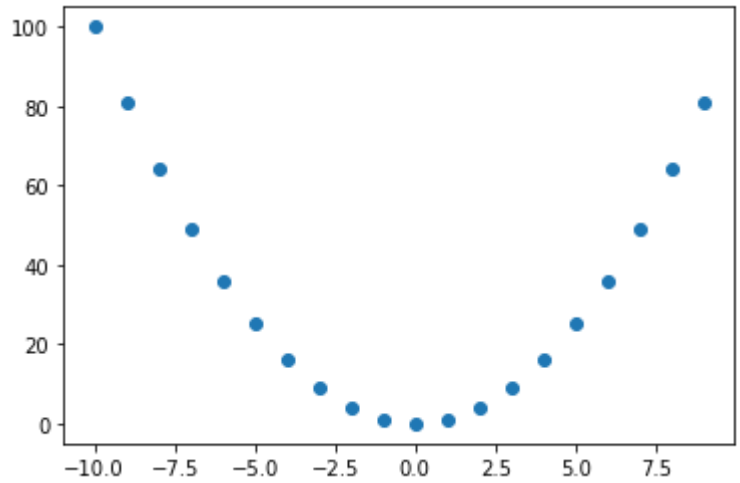
scatter plot

In [44]:

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

Out[44]:

```
<matplotlib.collections.PathCollection at 0x2aa51bdb460>
```



In [51]:

```
df.columns
```

Out[51]:

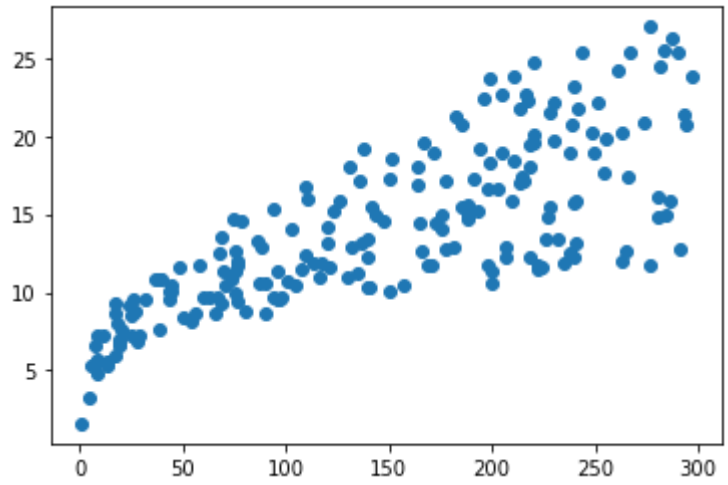
```
Index(['TV', 'radio', 'newspaper', 'sales'], dtype='object')
```

In [53]:

```
plt.scatter(df['TV'], df['sales'])
```

Out[53]:

```
<matplotlib.collections.PathCollection at 0x2aa51c53af0>
```



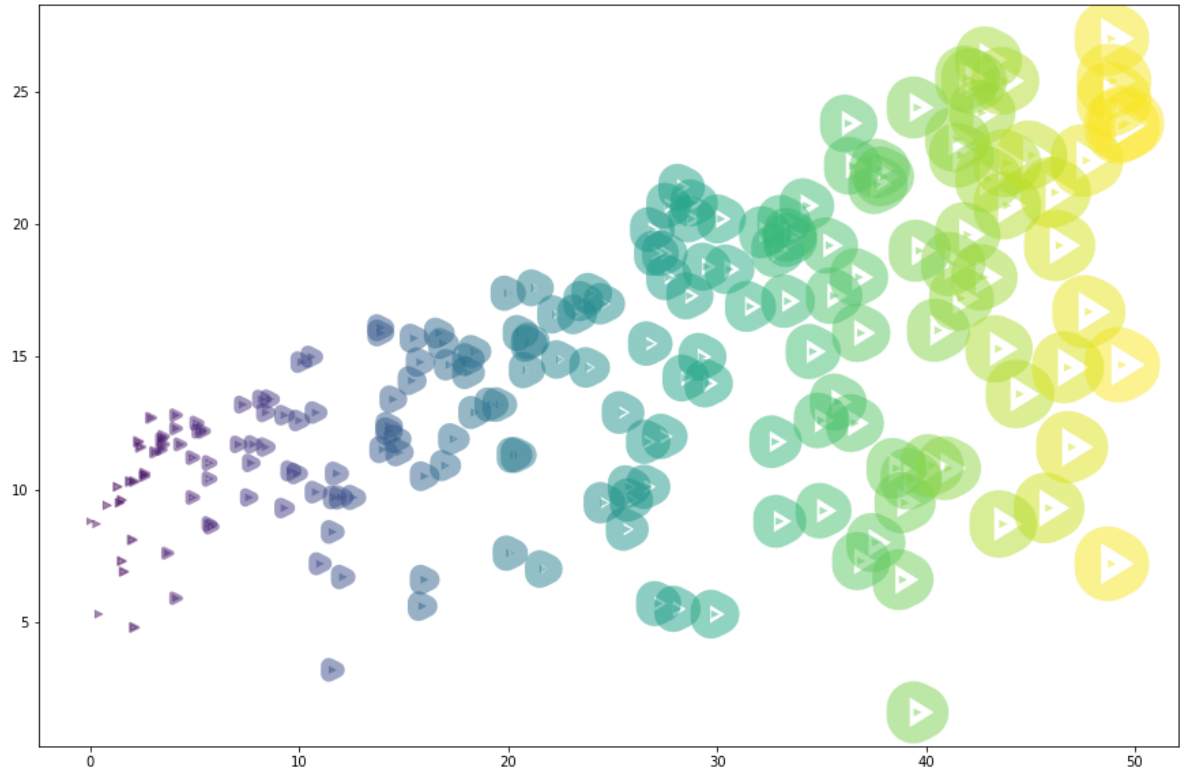
In [64]:

```
plt.figure(figsize = (15, 10))

plt.scatter(df['radio'], df['sales'], marker = '>', c = df['radio'].values, alpha = 0.5, li
```

Out[64]:

```
<matplotlib.collections.PathCollection at 0x2aa5365ef70>
```



In [59]:

```
help(plt.scatter)
```

```
Help on function scatter in module matplotlib.pyplot:

scatter(x, y, s=None, c=None, marker=None, cmap=None, norm=None, vmin=None,
vmax=None, alpha=None, linewidths=None, verts=<deprecated parameter>, edgeco
lors=None, *, plotnonfinite=False, data=None, **kwargs)
    A scatter plot of *y* vs. *x* with varying marker size and/or color.

    Parameters
    ----------
    x, y : scalar or array-like, shape (n, )
        The data positions.

    s : scalar or array-like, shape (n, ), optional
        The marker size in points**2.
        Default is ``rcParams['lines.markersize'] ** 2``.

    c : array-like or list of colors or color, optional
        The marker colors. Possible values:

        - A scalar or sequence of n numbers to be mapped to colors using
          *cmap* and *norm*.
        - A 2-D array in which the rows are RGB or RGBA.
        - A sequence of colors of length n.
        - A single color format string.

        Note that *c* should not be a single numeric RGB or RGBA sequence
        because that is indistinguishable from an array of values to be
        colormapped. If you want to specify the same RGB or RGBA value for
        all points, use a 2-D array with a single row.  Otherwise, value-
        matching will have precedence in case of a size matching with *x*
        and *y*.

        If you wish to specify a single color for all points
        prefer the *color* keyword argument.

        Defaults to `None`. In that case the marker color is determined
        by the value of *color*, *facecolor* or *facecolors*. In case
        those are not specified or `None`, the marker color is determined
        by the next color of the ``Axes`` current "shape and fill" color
        cycle. This cycle defaults to :rc:`axes.prop_cycle`.

    marker : `~matplotlib.markers.MarkerStyle`, optional
        The marker style. *marker* can be either an instance of the class
        or the text shorthand for a particular marker.
        Defaults to ``None``, in which case it takes the value of
        :rc:`scatter.marker` = 'o'.
        See `~matplotlib.markers` for more information about marker styles.

    cmap : `~matplotlib.colors.Colormap`, optional, default: None
        A `.Colormap` instance or registered colormap name. *cmap* is only
        used if *c* is an array of floats. If ``None``, defaults to rc
        ``image.cmap``.

    norm : `~matplotlib.colors.Normalize`, optional, default: None
        A `.Normalize` instance is used to scale luminance data to 0, 1.
        *norm* is only used if *c* is an array of floats. If *None*, use
```

```
        the default `.colors.Normalize`.

    vmin, vmax : scalar, optional, default: None
        *vmin* and *vmax* are used in conjunction with *norm* to normalize
        luminance data. If None, the respective min and max of the color
        array is used. *vmin* and *vmax* are ignored if you pass a *norm*
        instance.

    alpha : scalar, optional, default: None
        The alpha blending value, between 0 (transparent) and 1 (opaque).

    linewidths : scalar or array-like, optional, default: None
        The linewidth of the marker edges. Note: The default *edgecolors*
        is 'face'. You may want to change this as well.
        If *None*, defaults to :rc:`lines.linewidth`.

    edgecolors : {'face', 'none', *None*} or color or sequence of color, opt
ional.
        The edge color of the marker. Possible values:

        - 'face': The edge color will always be the same as the face color.
        - 'none': No patch boundary will be drawn.
        - A Matplotlib color or sequence of color.

        Defaults to ``None``, in which case it takes the value of
        :rc:`scatter.edgecolors` = 'face'.

        For non-filled markers, the *edgecolors* kwarg is ignored and
        forced to 'face' internally.

    plotnonfinite : boolean, optional, default: False
        Set to plot points with nonfinite *c*, in conjunction with
        `~matplotlib.colors.Colormap.set_bad`.

    Returns
    -------
    paths : `~matplotlib.collections.PathCollection`

    Other Parameters
    ----------------
    **kwargs : `~matplotlib.collections.Collection` properties

    See Also
    --------
    plot : To plot scatter plots when markers are identical in size and
        color.

    Notes
    -----
    * The `.plot` function will be faster for scatterplots where markers
      don't vary in size or color.

    * Any or all of *x*, *y*, *s*, and *c* may be masked arrays, in which
      case all masks will be combined and only unmasked points will be
      plotted.

    * Fundamentally, scatter works with 1-D arrays; *x*, *y*, *s*, and *c*
      may be input as N-D arrays, but within scatter they will be
      flattened. The exception is *c*, which will be flattened only if its
      size matches the size of *x* and *y*.
```

```
    .. note::
        In addition to the above described arguments, this function can take
a
        **data** keyword argument. If such a **data** argument is given, the
        following arguments are replaced by **data[<arg>]**:

        * All arguments with the following names: 'c', 'color', 'edgecolor
s', 'facecolor', 'facecolors', 'linewidths', 's', 'x', 'y'.

        Objects passed as **data** must support item access (``data[<arg>]`
`) and
        membership test (``<arg> in data``).
```

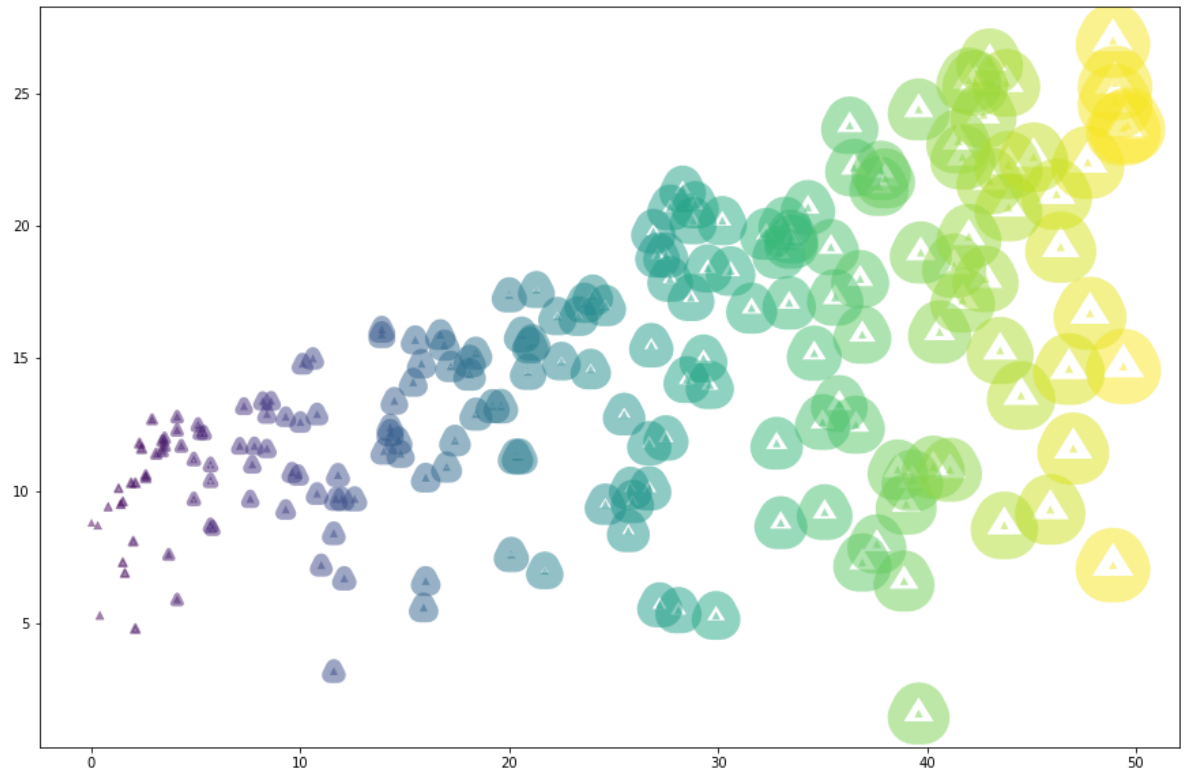
In [68]:

```
plt.figure(figsize = (15, 10))

plt.scatter(df['radio'], df['sales'], marker = '^', c = df['radio'].values, alpha = 0.5, li
```

Out[68]:

```
<matplotlib.collections.PathCollection at 0x2aa537d2940>
```



Bar Graph

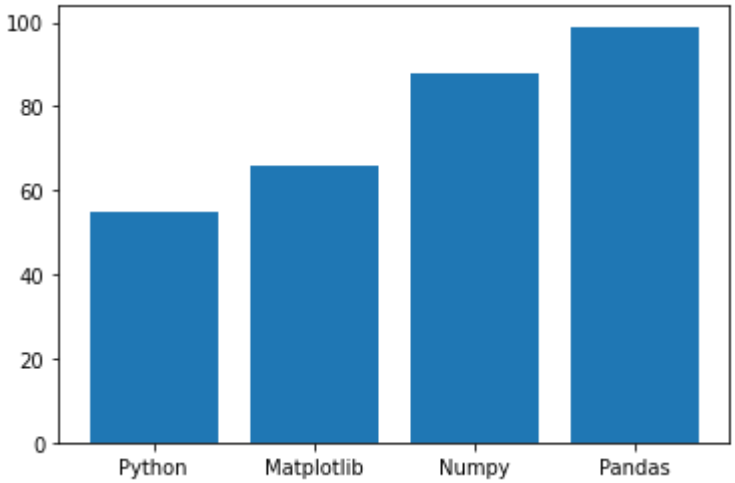
In [69]:

```
s = ['Python', 'Matplotlib', 'Numpy', 'Pandas']
marks = [55, 66, 88, 99]

plt.bar(s, marks)
```

Out[69]:

```
<BarContainer object of 4 artists>
```

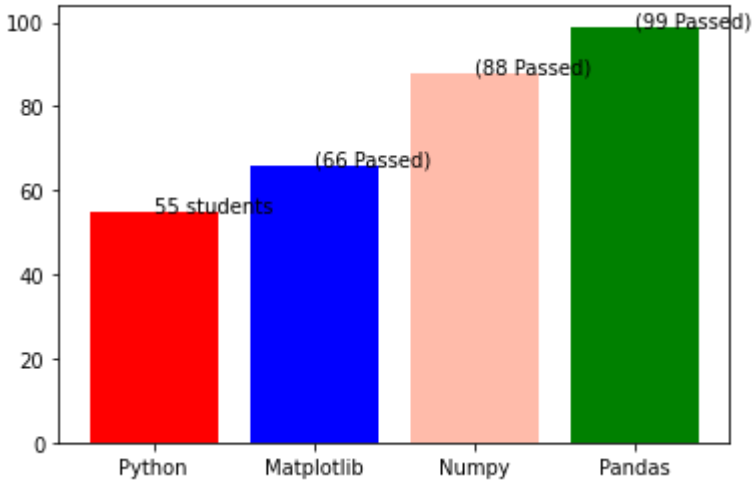


In [73]:

```
plt.bar(s, marks, color= ['r', 'b','#ffbbaa', 'green'])
plt.text(0, 55, '55 students')
plt.text(1, 66, '(66 Passed)')
plt.text(2, 88, '(88 Passed)')
plt.text(3, 99, '(99 Passed)')
```

Out[73]:

```
Text(3, 99, '(99 Passed)')
```



In [71]:

```
help(plt.bar)
```

```
Help on function bar in module matplotlib.pyplot:

bar(x, height, width=0.8, bottom=None, *, align='center', data=None, **kwarg
s)
    Make a bar plot.

    The bars are positioned at *x* with the given *align*\ment. Their
    dimensions are given by *width* and *height*. The vertical baseline
    is *bottom* (default 0).

    Each of *x*, *height*, *width*, and *bottom* may either be a scalar
    applying to all bars, or it may be a sequence of length N providing a
    separate value for each bar.

    Parameters
    ----------
    x : sequence of scalars
        The x coordinates of the bars. See also *align* for the
        alignment of the bars to the coordinates.

    height : scalar or sequence of scalars
        The height(s) of the bars.

    width : scalar or array-like, optional
        The width(s) of the bars (default: 0.8).

    bottom : scalar or array-like, optional
        The y coordinate(s) of the bars bases (default: 0).

    align : {'center', 'edge'}, optional, default: 'center'
        Alignment of the bars to the *x* coordinates:

        - 'center': Center the base on the *x* positions.
        - 'edge': Align the left edges of the bars with the *x* positions.

        To align the bars on the right edge pass a negative *width* and
        ``align='edge'``.

    Returns
    -------
    container : `.BarContainer`
        Container with all the bars and optionally errorbars.

    Other Parameters
    ----------------
    color : scalar or array-like, optional
        The colors of the bar faces.

    edgecolor : scalar or array-like, optional
        The colors of the bar edges.

    linewidth : scalar or array-like, optional
        Width of the bar edge(s). If 0, don't draw edges.

    tick_label : str or array-like, optional
        The tick labels of the bars.
```

```
        Default: None (Use default numeric labels.)

    xerr, yerr : scalar or array-like of shape(N,) or shape(2, N), optional
        If not *None*, add horizontal / vertical errorbars to the bar tips.
        The values are +/- sizes relative to the data:

        - scalar: symmetric +/- values for all bars
        - shape(N,): symmetric +/- values for each bar
        - shape(2, N): Separate - and + values for each bar. First row
          contains the lower errors, the second row contains the upper
          errors.
        - *None*: No errorbar. (Default)

        See :doc:`/gallery/statistics/errorbar_features`
        for an example on the usage of ``xerr`` and ``yerr``.

    ecolor : scalar or array-like, optional, default: 'black'
        The line color of the errorbars.

    capsize : scalar, optional
       The length of the error bar caps in points.
       Default: None, which will take the value from
       :rc:`errorbar.capsize`.

    error_kw : dict, optional
        Dictionary of kwargs to be passed to the `~.Axes.errorbar`
        method. Values of *ecolor* or *capsize* defined here take
        precedence over the independent kwargs.

    log : bool, optional, default: False
        If *True*, set the y-axis to be log scale.

    orientation : {'vertical',  'horizontal'}, optional
        *This is for internal use only.* Please use `barh` for
        horizontal bar plots. Default: 'vertical'.

    See also
    --------
    barh: Plot a horizontal bar plot.

    Notes
    -----
    The optional arguments *color*, *edgecolor*, *linewidth*,
    *xerr*, and *yerr* can be either scalars or sequences of
    length equal to the number of bars.  This enables you to use
    bar as the basis for stacked bar charts, or candlestick plots.
    Detail: *xerr* and *yerr* are passed directly to
    :meth:`errorbar`, so they can also have shape 2xN for
    independent specification of lower and upper errors.

    Other optional kwargs:

    Properties:
        agg_filter: a filter function, which takes a (m, n, 3) float array a
nd a dpi value, and returns a (m, n, 3) array
        alpha: float or None
        animated: bool
        antialiased or aa: unknown
        capstyle: {'butt', 'round', 'projecting'}
        clip_box: `.Bbox`
        clip_on: bool
```

```
        clip_path: Patch or (Path, Transform) or None
        color: color
        contains: callable
        edgecolor or ec: color or None or 'auto'
        facecolor or fc: color or None
        figure: `.Figure`
        fill: bool
        gid: str
        hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
        in_layout: bool
        joinstyle: {'miter', 'round', 'bevel'}
        label: object
        linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq),
...}
        linewidth or lw: float or None
        path_effects: `.AbstractPathEffect`
        picker: None or bool or float or callable
        rasterized: bool or None
        sketch_params: (scale: float, length: float, randomness: float)
        snap: bool or None
        transform: `.Transform`
        url: str
        visible: bool
        zorder: float

    .. note::
        In addition to the above described arguments, this function can take
a
        **data** keyword argument. If such a **data** argument is given, the
        following arguments are replaced by **data[<arg>]**:

        * All positional and all keyword arguments.

        Objects passed as **data** must support item access (``data[<arg>]`
`) and
        membership test (``<arg> in data``).
```