06 - Matplotlib

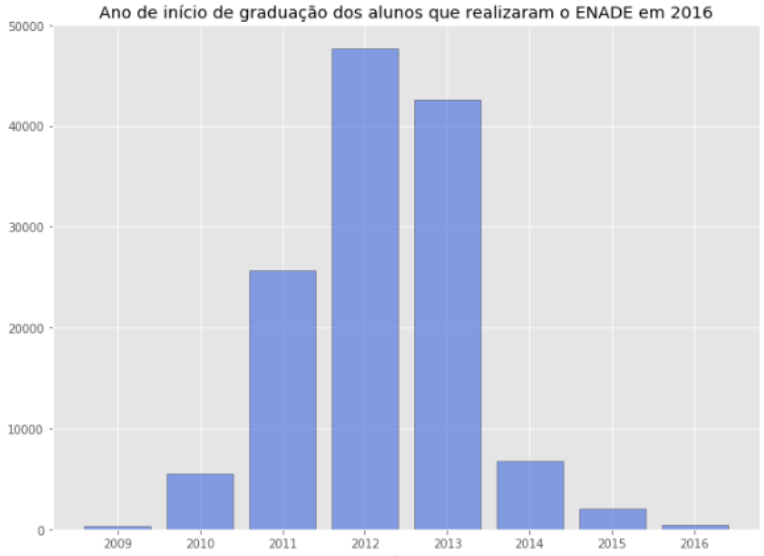
Docupedia Export

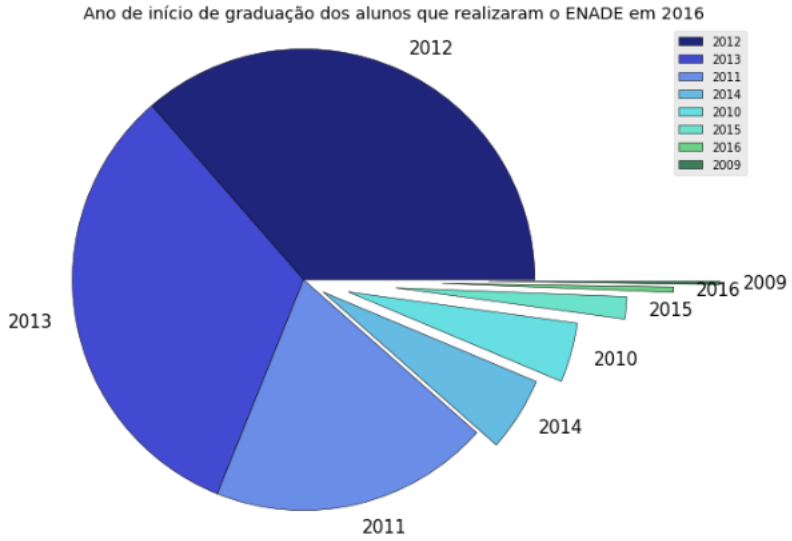
Author:Cardoso Cristian (CtP/ETS)
Date:10-Feb-2026 15:45

Table of Contents

1 TREINAMENTO DE PYTHON (MATPLOTLIB)

O matplotlib http://matplotlib.org/ org/) é uma biblioteca do Python para criação de gráficos em 2D, muito utilizado para visualização de dados e que apresenta uma série de possibilidades gráficas, como gráficos de barra, linha, pizza, histogramas, entre muitos outros.





Vamos começar importando o dataset.

```
import pandas as pd
```

```
df = pd.read_csv('respiradores.csv')
df
```

	MES	ACRE	ALAGOAS	AMAPA	AMAZONAS	BAHIA	...	RORAIMA	SERGIPE	SÃO PAULO	TOCANTINS	TOTAL
0	2020/04	0.0	0.0	45.0	90.0	0.0	...	0.0	0.0	0.0	0.0	387
1	2020/05	30.0	30.0	60.0	88.0	60.0	...	75.0	70.0	460.0	20.0	2284
2	2020/06	120.0	105.0	20.0	44.0	246.0	...	50.0	60.0	296.0	50.0	3741
3	2020/07	20.0	87.0	0.0	0.0	355.0	...	37.0	56.0	134.0	55.0	3317
4	2020/08	0.0	3.0	0.0	0.0	126.0	...	0.0	0.0	220.0	0.0	1408
5	2020/09	0.0	0.0	0.0	0.0	7.0	...	0.0	0.0	0.0	72.0	354
6	2020/10	0.0	11.0	0.0	80.0	15.0	...	0.0	0.0	39.0	26.0	683
7	2020/11	0.0	0.0	0.0	0.0	5.0	...	0.0	0.0	0.0	0.0	461
8	2020/12	0.0	1.0	0.0	90.0	143.0	...	0.0	0.0	9.0	3.0	1308
9	2021/01	20.0	50.0	8.0	204.0	32.0	...	50.0	0.0	6.0	4.0	941
10	2021/02	40.0	0.0	4.0	77.0	72.0	...	0.0	0.0	58.0	0.0	675

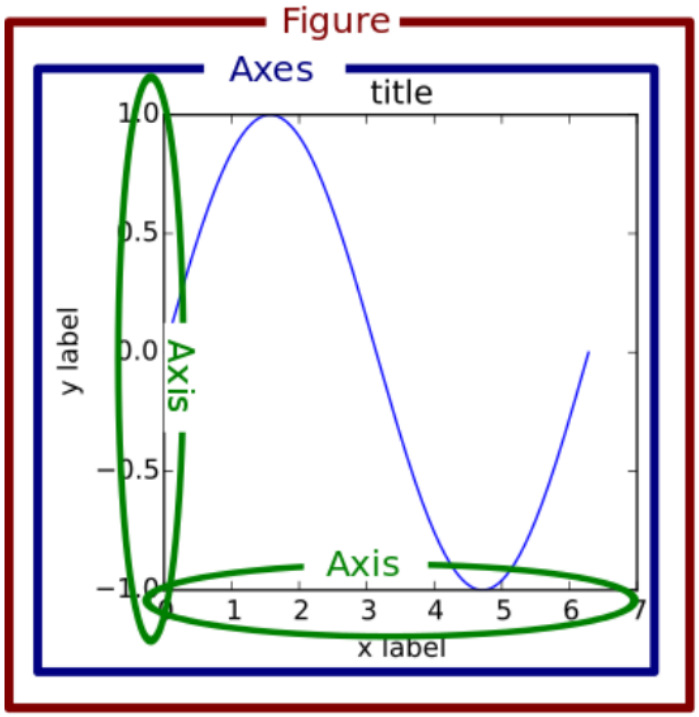
11 rows × 32 columns

O dataset mostra a quantidade de respiradores comprados por cada estado brasileiro em cada mês da pandemia.

1.1 Importando a biblioteca

O modo mais comum de chamar a biblioteca é usando o alias **plt**.

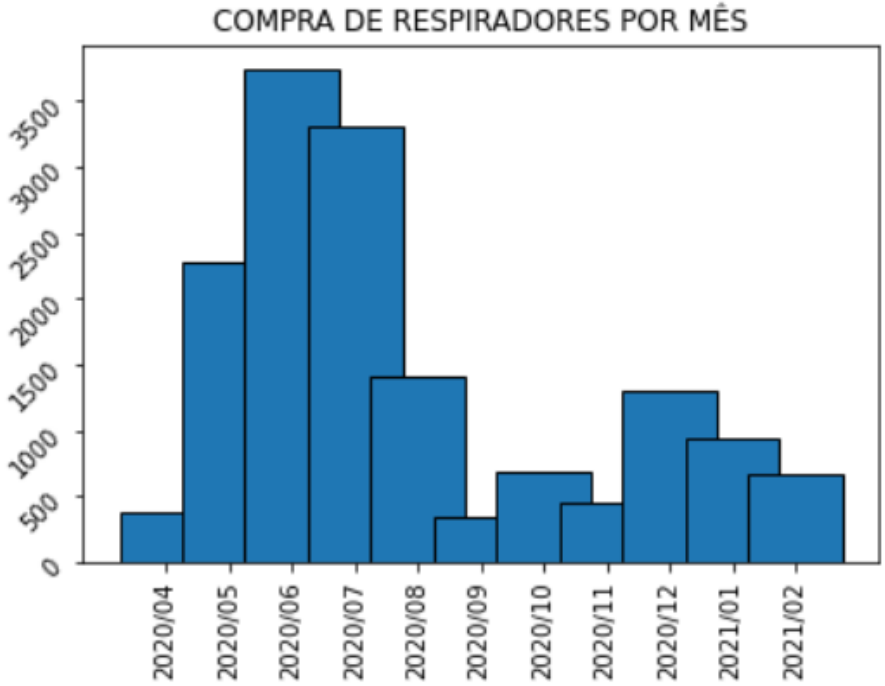
```
import matplotlib.pyplot as plt
```



1.2 Gráfico de barras

Vamos começar fazendo um gráfico de barras, para ver a quantidade total de respiradores comprados em cada mês no Brasil.
O gráfico de barras é um tipo de gráfico categórico, um eixo apresenta as categorias e o outro os valores
ax.bar(x, height, width, bottom, align)

```
x=df.MES
altura=df.TOTAL
plt.bar(x,altura, 1.5,align= "center", edgecolor="black" )
plt.title("COMPRA DE RESPIRADORES POR MÊS")
plt.xticks(rotation = '90')
plt.yticks(rotation = '45')
plt.show()

help(plt.bar)
```



Help on function bar in module matplotlib.pyplot:

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

    The bars are positioned at *x* with the given *align*\ment. Their

dimensions are given by *height* and *width*. The vertical baseline
is *bottom* (default 0).

Many parameters can take either a single value applying to all bars
or a sequence of values, one for each bar.

Parameters
----
x : float or array-like
    The x coordinates of the bars. See also *align* for the
    alignment of the bars to the coordinates.

height : float or array-like
    The height(s) of the bars.

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

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

align : {'center', 'edge'}, 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

`.BarContainer`
    Container with all the bars and optionally errorbars.

Other Parameters

```
----------------
color : color or list of color, optional
    The colors of the bar faces.

edgecolor : color or list of color, optional
    The colors of the bar edges.

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

tick_label : str or list of str, optional
    The tick labels of the bars.
    Default: None (Use default numeric labels.)

xerr, yerr : float 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 : color or list of color, default: 'black'
    The line color of the errorbars.

capsize : float, default: :rc:`errorbar.capsize`
   The length of the error bar caps in points.

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

**kwargs : `.Rectangle` properties

Properties:
agg_filter: a filter function, which takes a (m, n, 3) float array and 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: unknown
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 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

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

Notes
-----
Stacked bars can be achieved by passing individual *bottom* values per
bar. See :doc:`/gallery/lines_bars_and_markers/bar_stacked`.

.. note::
    In addition to the above described arguments, this function can take
    a *data* keyword argument. If such a *data* argument is given,
    every other argument can also be string ``s``, which is
    interpreted as ``data[s]`` (unless this raises an exception).

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

```
total_por_estado = df.sum()[1:-1]
total_por_estado
```

```
ACRE              230
ALAGOAS           287
AMAPA             137
AMAZONAS          673
BAHIA            1061
CEARA             364
DISTRITO FEDERAL  273
ESPIRITO SANTO    449
```

```
GOIAS                939
HAITI                100
LIBANO               300
MARANHÃO             328
MATO GROSSO          272
MATO GROSSO DO SUL   360
MINAS GERAIS        1139
PARA                 535
PARAIBA              462
PARANA               993
PERNAMBUCO           303
PERU                 330
PIAUI                260
RIO DE JANEIRO      1844
RIO GRANDE DO NORTE  330
RIO GRANDE DO SUL   1028
RONDONIA             349
RORAIMA              212
SANTA CATARINA       363
SERGIPE              186
SÃO PAULO           1222
TOCANTINS            230
TOTAL               2366
dtype: object
```

```
x=df.columns[1:-1]
plt.bar(x, total_por_estado, color = 'black', zorder=3)
plt.title('COMPRA DE RESPIRADORES POR ESTADO')
plt.xticks(rotation='vertical')
plt.grid(zorder=0)
plt.show()
```

Podemos fazer o gráfico de barras na horizontal também:

```
plt.barh(df.MES,df.TOTAL, color = 'purple')
plt.title('COMPRA DE RESPIRADORES POR MÊS')
plt.show()
```

1.3 Gráfico de linha

Usamos a função **plot** para fazer gráficos de linha:

```
plt.plot(df.MES,df.PARANA)
plt.xticks(rotation='45')
plt.grid(linestyle='dashed')
plt.title("COMPRA DE RESPIRADORES POR MÊS NO PARANÁ")
plt.show()
```

1.3.1 Podemos passar argumentos na função para personalizar nosso gráfico

```
plt.plot(df.MES,df.PARANA, color = 'purple', marker='x', linewidth = 3, markersize=10)
plt.xticks(rotation='45')
plt.title("COMPRA DE RESPIRADORES POR MÊS NO PARANÁ")
plt.grid(linestyle='dashed')
plt.show()
```

Vamos ver alguns argumentos usando o **help**.

```
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 the
'axes.prop_cycle' rcParam.

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:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array
alpha: float
animated: bool
antialiased or aa: bool
clip_box: `.Bbox`
clip_on: bool
clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | 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)
drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-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
```

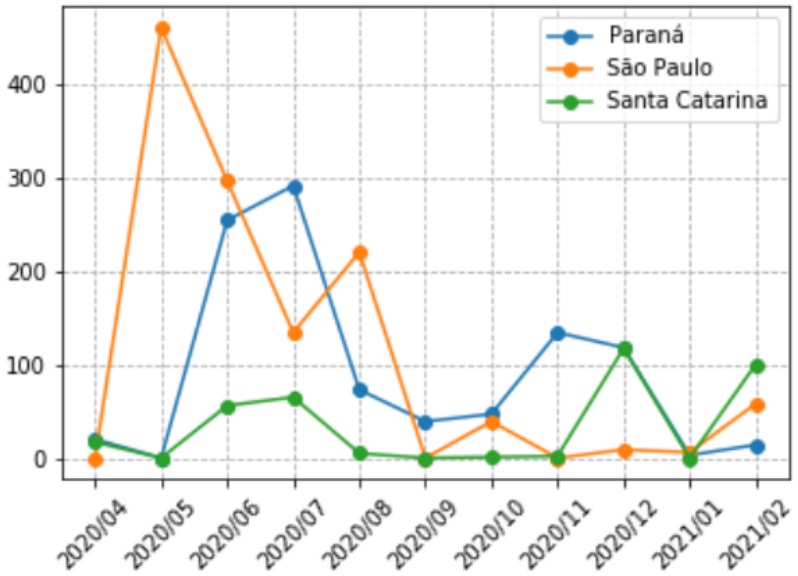
```
=============   ================================
```

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'``).

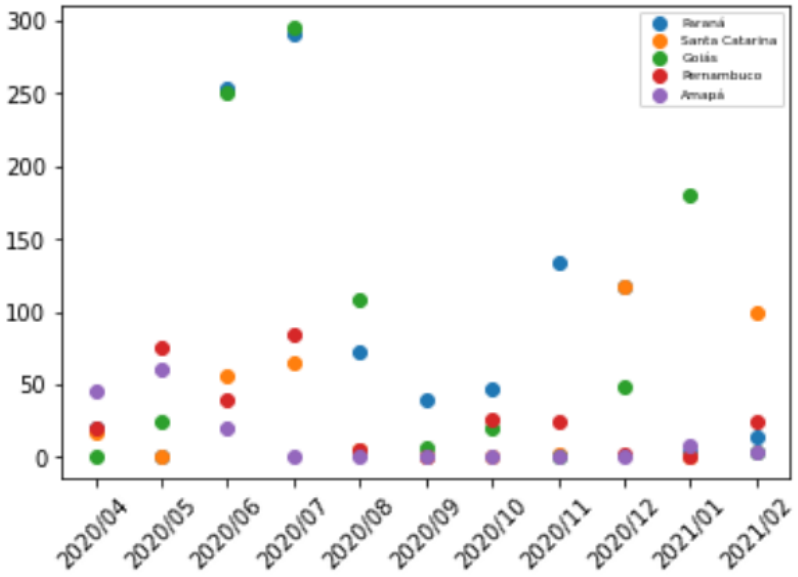
1.3.2 Para adicionar mais curvas no mesmo gráfico:

```
plt.plot(df.MES,df.PARANA,marker='o', label = 'Paraná')
plt.plot(df.MES,df['SÃO PAULO'],marker='o', label = 'São Paulo')
plt.plot(df.MES,df['SANTA CATARINA'],marker='o', label = 'Santa Catarina')
plt.legend()
plt.xticks(rotation='45')
plt.grid(linestyle='dashed')
plt.show()
```



1.4 Gráfico de dispersão (Scatter plot)

```
plt.scatter(df['MES'],df['PARANA'], label = 'Paraná')
plt.scatter(df['MES'],df['SANTA CATARINA'], label = 'Santa Catarina')
plt.scatter(df['MES'],df['GOIAS'], label = 'Goiás')
plt.scatter(df['MES'],df['PERNAMBUCO'], label = 'Pernambuco')
plt.scatter(df['MES'],df['AMAPA'], label = 'Amapá')
plt.legend() #fontsize=10 / prop={"size":10}
plt.title("")
plt.xticks(rotation=45)
plt.show()

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.

Returns

`~matplotlib.legend.Legend`

Other Parameters
----

loc : str or pair of floats, default: :rc:`legend.loc` ('best' for axes, '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 axes/figure.

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

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

For back-compatibility, ```'center right'`` (but no other location) can also
be spelled ```'right'``, and each "string" locations can also be given 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 arbitrary
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 called.

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 used::

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

ncol : int, default: 1
The number of columns that the legend has.

prop : None or `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 {'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 current

default font size. This argument is only used if *prop* is not specified.

labelcolor : str or list
Sets the color of the text in the legend. Can be a valid color string (for example, 'red'), or a list of color strings. The labelcolor can also be made to match the color of the line or marker using 'linecolor', 'markerfacecolor' (or 'mfc'), or 'markeredgecolor' (or 'mec').

numpoints : int, default: :rc:`legend.numpoints`
The number of marker points in the legend when creating a legend entry for a `.Line2D` (line).

scatterpoints : int, default: :rc:`legend.scatterpoints`
The number of marker points in the legend when creating a legend entry for a `.PathCollection` (scatter plot).

scatteryoffsets : iterable of floats, default: ``[0.375, 0.5, 0.3125]``
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]``.

markerscale : float, default: :rc:`legend.markerscale`
The relative size of legend markers compared with the originally drawn ones.

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

frameon : bool, default: :rc:`legend.frameon`
Whether the legend should be drawn on a patch (frame).

fancybox : bool, default: :rc:`legend.fancybox`
Whether round edges should be enabled around the `~.FancyBboxPatch` which makes up the legend's background.

shadow : bool, default: :rc:`legend.shadow`
    Whether to draw a shadow behind the legend.

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

facecolor : "inherit" or color, default: :rc:`legend.facecolor`
    The legend's background color.
    If ``"inherit"``, use :rc:`axes.facecolor`.

edgecolor : "inherit" or color, default: :rc:`legend.edgecolor`
    The legend's background patch edge color.
    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 `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 : int or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}, default: :rc:`legend.title_fontsize`
    The font size of the legend's title.

borderpad : float, default: :rc:`legend.borderpad`
    The fractional whitespace inside the legend border, in font-size units.

labelspacing : float, default: :rc:`legend.labelspacing`
    The vertical space between the legend entries, in font-size units.

handlelength : float, default: :rc:`legend.handlelength`
    The length of the legend handles, in font-size units.

handletextpad : float, default: :rc:`legend.handletextpad`
    The pad between the legend handle and text, in font-size units.

borderaxespad : float, default: :rc:`legend.borderaxespad`
    The pad between the axes and legend border, in font-size units.

columnspacing : float, default: :rc:`legend.columnspacing`
    The spacing between columns, in font-size units.

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 `matplotlib.legend.Legend.get_legend_handler_map`.

Notes

Some artists are not supported by this function. See
:doc:`/tutorials/intermediate/legend_guide` for details.

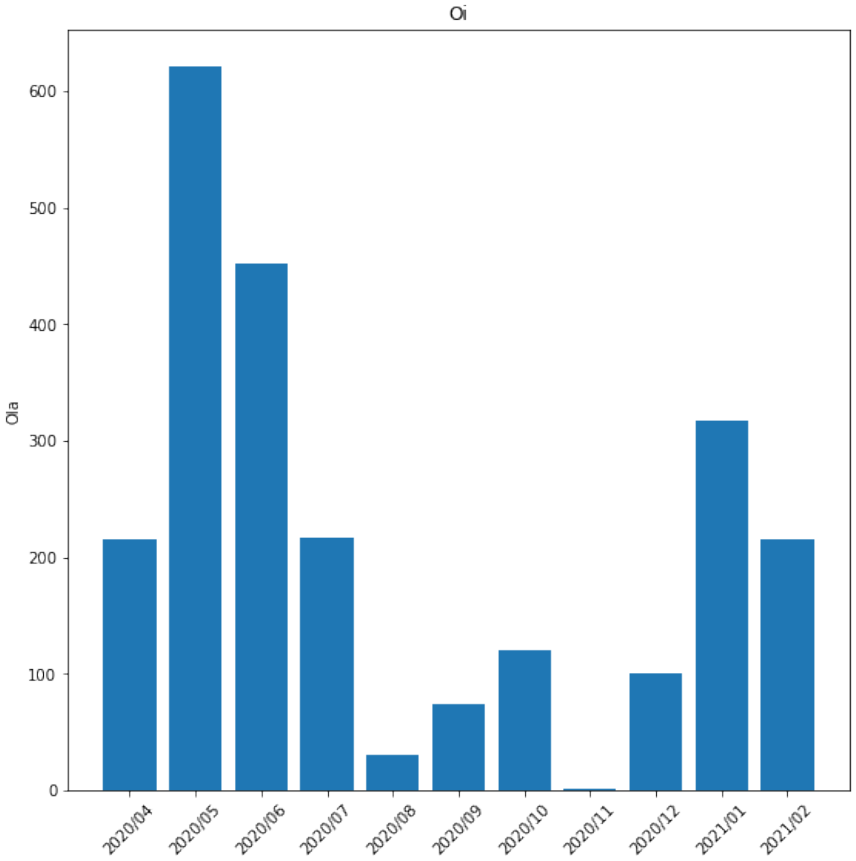
Examples

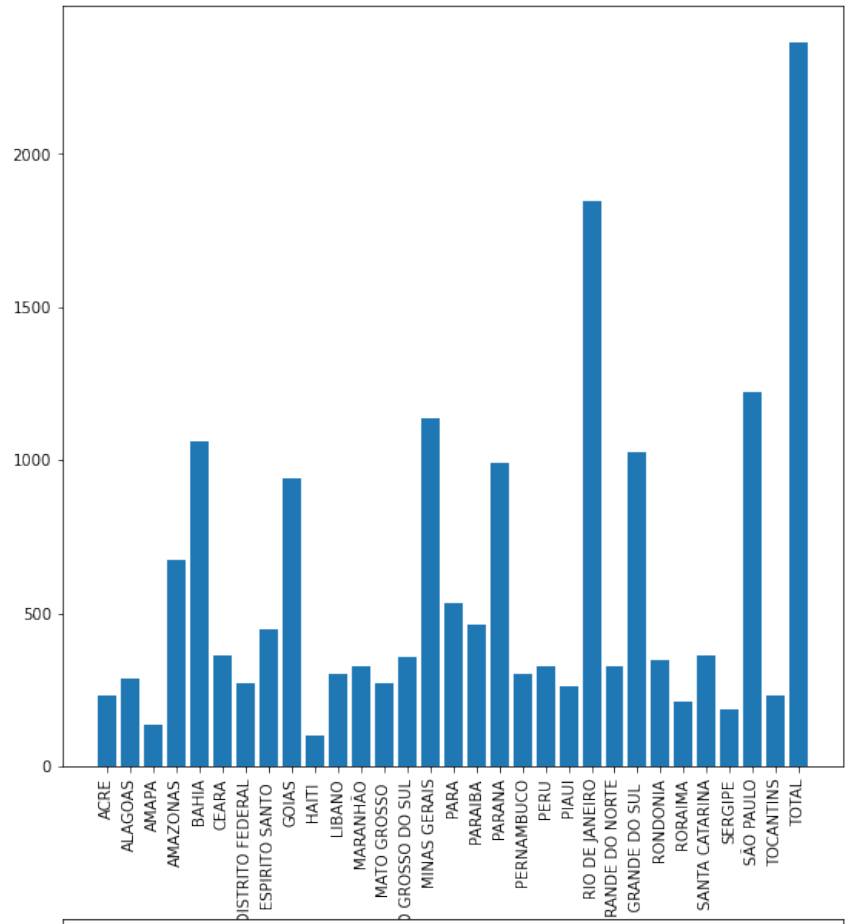
.. plot:: gallery/text_labels_and_annotations/legend.py

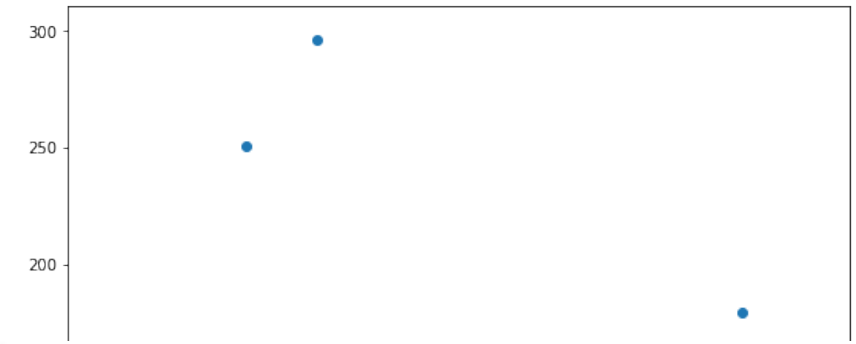
1.5 Subplots

Podemos criar uma figura contendo uma matriz de gráficos dentro dela, para isso utilizamos o **subplots**, informando o número de colunas e de linhas. A função retorna dois valores, o primeiro se refere à própria figura, a segunda é um array de gráficos, onde podemos acessar cada um por indexação.

```
fig, eixo = plt.subplots(nrows = 2, ncols = 2, figsize = (20,20))
eixo[0][0].bar(df.MES,df.TOTAL)
eixo[0][1].bar(df.columns[1:-1],df.sum()[1:-1])
eixo[1][0].scatter(df['MES'],df['GOIAS'], label = 'Goiás')
eixo[1][1].plot(df.MES,df.PARANA, color = 'purple', marker='x', linewidth = 3, markersize=10)

eixo[0][0].set_title("Oi")
eixo[0][0].set_ylabel("Ola")
eixo[0][0].tick_params(axis='x',  labelrotation=45) # os métodos dos eixos de subplots são diferentes das funções do plot
eixo[0][1].tick_params(axis='x', labelrotation=90)
eixo[1][0].tick_params(axis='x', labelrotation=45) # os métodos dos eixos de subplots são diferentes das funções do plot
eixo[1][1].tick_params(axis='x', labelrotation=90)
```







1.6 Gráfico de pizza

1.6.1 Vamos analisar a proporção de compra de respiradores entre os estados da região Sul do Brasil.

Primeiro vamos separar um dataset somente com os dados desses estados:

```
estados_sul = df.loc[:,['PARANA','SANTA CATARINA','RIO GRANDE DO SUL ']]
estados_sul
```

	PARANA	SANTA CATARINA	RIO GRANDE DO SUL
0	20.0	17.0	0.0
1	0.0	0.0	0.0
2	254.0	56.0	297.0
3	291.0	65.0	362.0
4	73.0	5.0	92.0
5	39.0	0.0	108.0
6	47.0	1.0	49.0
7	134.0	2.0	32.0
8	118.0	117.0	73.0
9	3.0	0.0	15.0
10	14.0	100.0	0.0

Agora podemos plotar esses dados num gráfico de pizza, utilizando a função **pie**.

```
estados_sul = df.loc[:,['PARANA','SANTA CATARINA','RIO GRANDE DO SUL ']]
print(estados_sul.sum())
def valor():
    x=estados_sul.sum()["PARANA"]
    y=estados_sul.sum()["SANTA CATARINA"]
    z=estados_sul.sum()["RIO GRANDE DO SUL "]
    return x,y,z
sizes=valor()

p, tx, autotexts = plt.pie(estados_sul.sum(), labels=estados_sul.columns,
        autopct="", shadow=True)
print(autotexts)
```

```
for i, a in enumerate(autotexts):
    a.set_text("{}".format(sizes[i]))

#plt.pie(estados_sul.sum(), labels = estados_sul.columns, autopct=valor)
#plt.title('COMPRA DE RESPIRADORES NO SUL')
#plt.show()
```

PARANA 993.0
SANTA CATARINA 363.0
RIO GRANDE DO SUL 1028.0
dtype: float64
[Text(0.15554597274115683, 0.5794872305443902, ''), Text(-0.5993619326185148, 0.027663581254041125, ''), Text(0.12866273790887076, -0.5860425751375008, '')]

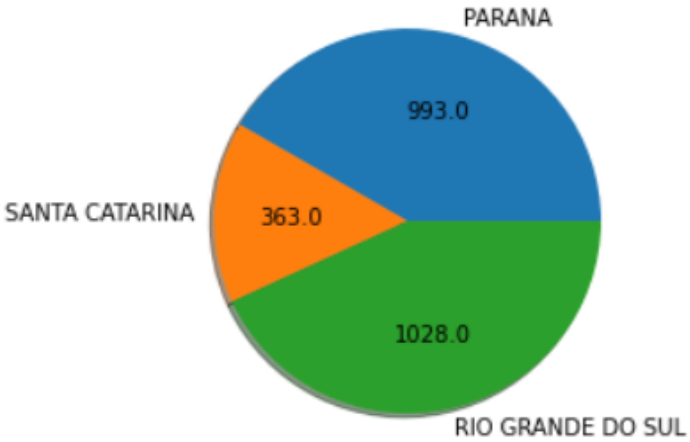


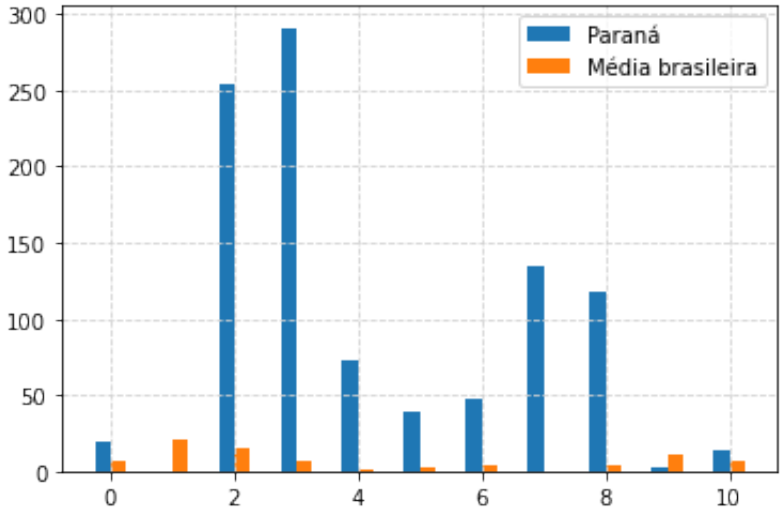
Gráfico de múltiplas barras

Vamos usar um truque para deixar nossas barras deslocadas. Nos valores do eixo X, vamos deslocar cada valor 0.25 para esquerda, para a primeira barra, e 0.25 para a direita, para a segunda barra.
Assim conseguimos comparar facilmente a quantidade de respiradores comprados no Paraná em cada mês com a média brasileira (total de cada mês/ quantidade de estados).

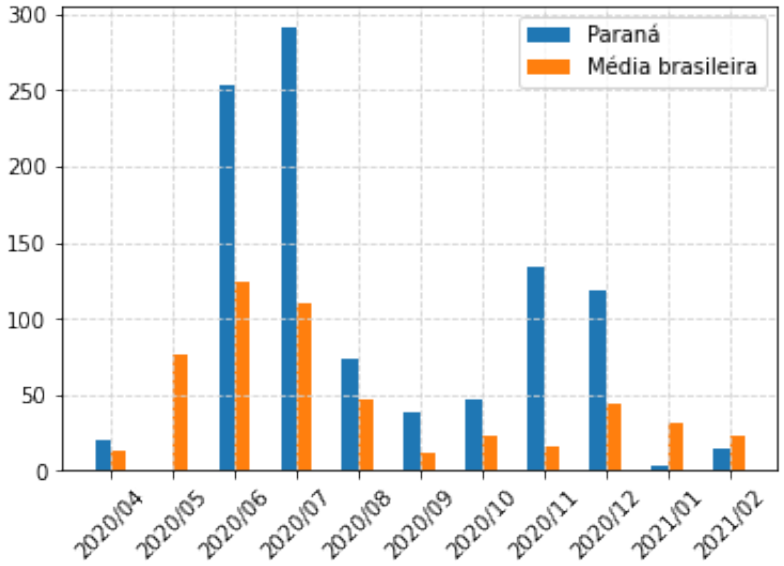
```
import numpy as np
```

```
print(df.shape)
print([a-0.25 for a in range(df.shape[0])])
print([a+0.25 for a in range(df.shape[0])])
print(df.PARANA)

plt.bar([i-0.25 for i in range(df.shape[0])],df.PARANA, width = +0.25,
        label = 'Paraná', align = 'edge')

plt.bar([a+0.25 for a in range(df.shape[0])],df.TOTAL/30,width = -0.25,
       label = 'Média brasileira', align = 'edge')
plt.legend()
plt.grid(color='lightgray',linestyle='dashed')
```

```
(11, 32)
[-0.25, 0.75, 1.75, 2.75, 3.75, 4.75, 5.75, 6.75, 7.75, 8.75, 9.75]
[0.25, 1.25, 2.25, 3.25, 4.25, 5.25, 6.25, 7.25, 8.25, 9.25, 10.25]
0     20.0
1      0.0
2    254.0
3    291.0
4     73.0
5     39.0
6     47.0
7    134.0
8    118.0
9      3.0
10    14.0
Name: PARANA, dtype: float64
```




```
import numpy as np

plt.bar([a-0.25 for a in range(df.shape[0])],df.PARANA, width = 0.25,
        label = 'Paraná', align = 'edge')
plt.bar([a+0.25 for a in range(df.shape[0])],df.TOTAL/30,width = -0.25,
       label = 'Média brasileira', align = 'edge')
plt.xticks(np.arange(11),df.MES, rotation=45)
plt.legend()
plt.grid(color='lightgray',linestyle='dashed')
```



1.7 Histogramas

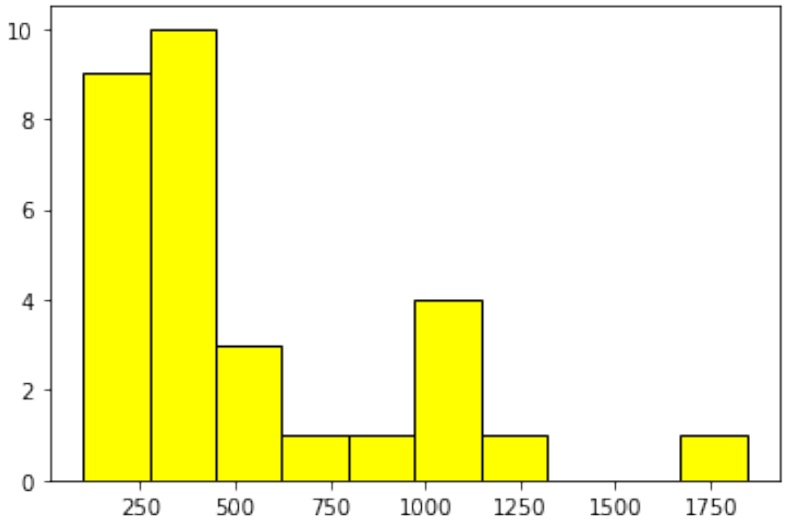
Histogramas são gráficos de barras que apresentam a distribuição de uma variável dentro de um conjunto de dados.
Vamos ver como se distribue o valor total de respiradores comprados em cada estado.

```
total_por_estado = df.iloc[:,1:-1].sum()
total_por_estado
```

```
ACRE          230.0
ALAGOAS       287.0
AMAPA         137.0
AMAZONAS      673.0
BAHIA        1061.0
CEARA         364.0
```

```
DISTRITO FEDERAL       273.0
ESPIRITO SANTO         449.0
GOIAS                  939.0
HAITI                  100.0
LIBANO                 300.0
MARANHÃO               328.0
MATO GROSSO            272.0
MATO GROSSO DO SUL     360.0
MINAS GERAIS          1139.0
PARA                   535.0
PARAIBA                462.0
PARANA                 993.0
PERNAMBUCO             303.0
PERU                   330.0
PIAUI                  260.0
RIO DE JANEIRO        1844.0
RIO GRANDE DO NORTE    330.0
RIO GRANDE DO SUL     1028.0
RONDONIA               349.0
RORAIMA                212.0
SANTA CATARINA         363.0
SERGIPE                186.0
SÃO PAULO             1222.0
TOCANTINS              230.0
dtype: float64
```

```
plt.hist(total_por_estado,ec='black',color='yellow')
plt.show()
# podemos ver que a maior parte dos estados comprou entre +- 100 a 500 respiradores
```



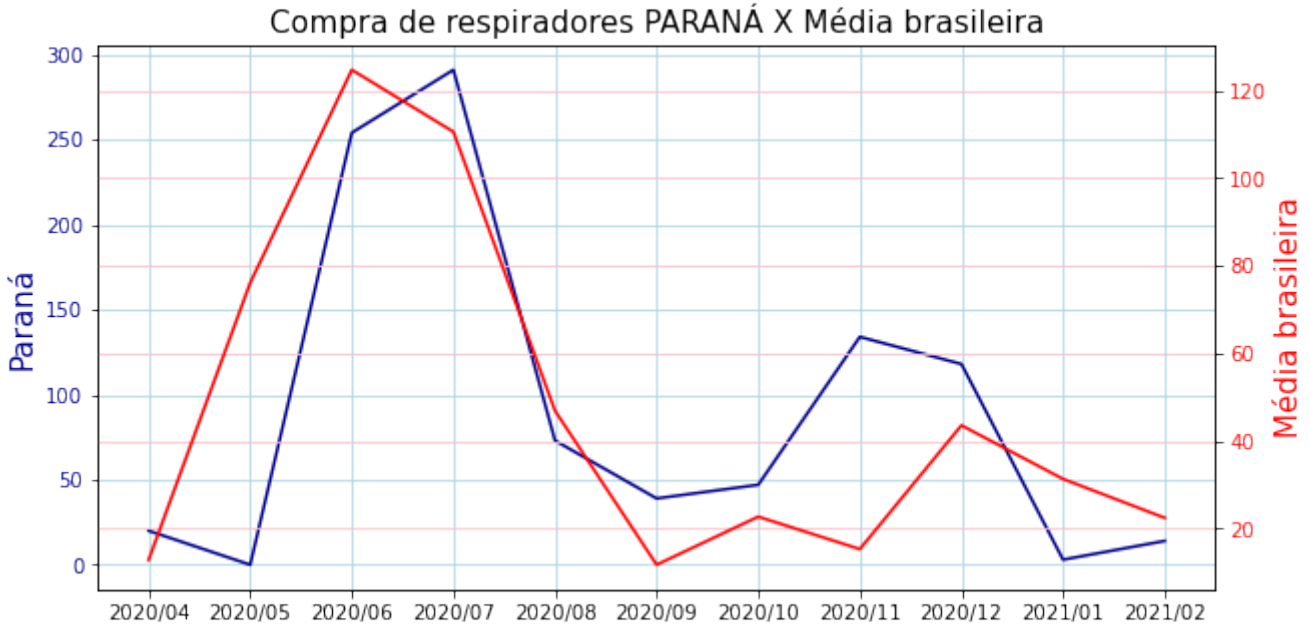
1.8 Duplo eixo Y

Podemos adicionar um eixo Y ao nosso gráfico, assim conseguimos reaproveitar o mesmo eixo X para mostrar duas variáveis diferentes.

```
fig, eixo = plt.subplots(ncols=1,nrows=1,figsize=(10,5))
eixo.plot(df.MES,df.PARANA,label='Paraná', color='darkblue')

eixo2 = eixo.twinx()
eixo2.plot(df.MES,df.TOTAL/30,color='red')

eixo2.tick_params(axis='y', labelcolor='red')
eixo.tick_params(axis='y', labelcolor='darkblue')

eixo.set_ylabel("Paraná", color='darkblue',fontsize=15)
eixo2.set_ylabel("Média brasileira", color='red',fontsize=15)

eixo.grid(color='lightblue')
eixo2.grid(color='pink')
```

```
eixo.set_title('Compra de respiradores PARANÁ X Média brasileira', fontsize= 15)
```

Text(0.5, 1.0, 'Compra de respiradores PARANÁ X Média brasileira')



1.8.1 Podemos salvá-lo como imagem:

```
fig.savefig('grafico_parana.png')
```

Note que eu quero salvar a figura inteira como imagem, por isso utilizamos a variável **fig** e não **eixo**.

1.9 Desafio

Faça um gráfico que mostre a quantidade de respiradores comprados por região do Brasil. (por mês e total)

Norte	Nordeste	Centro-Oeste	Sudeste	Sul
Amazonas	Maranhão	Goiás	Minas Gerais	Santa Catarina
Acre	Piauí	Mato Grosso	Espírito Santo	Paraná
Rondônia	Rio Grande do Norte	Mato Grosso do Sul	Rio de Janeiro	Rio Grande do Sul
Roraima	Ceará	Distrito Federal	São Paulo	
Amapá	Paraíba			
Pará	Bahia			
Tocantins	Pernambuco			
	Alagoas			
	Sergipe			

Resultado

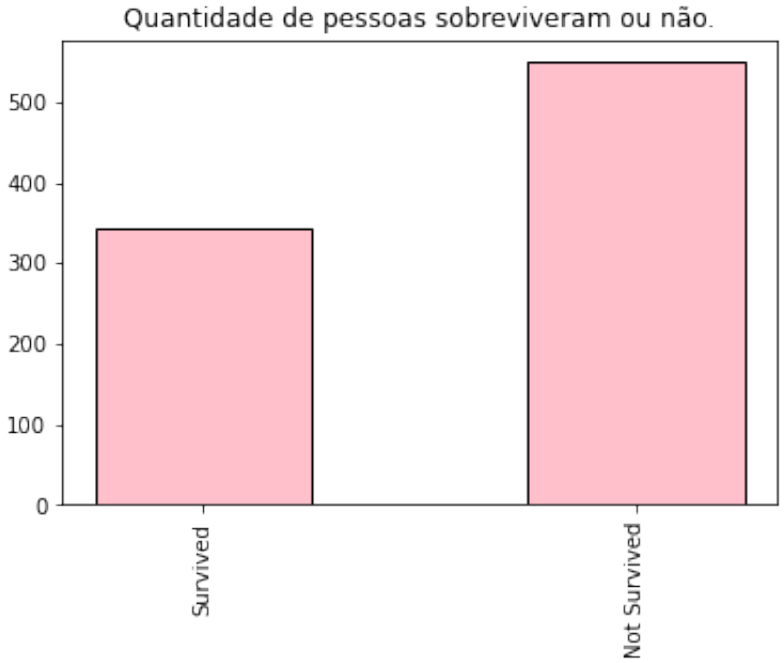
```
norte=df.loc[:, ["AMAZONAS", "AMAPA", "RORAIMA", "ACRE", "PARA", "RONDONIA", "TOCANTINS"]]
sul=df.loc[:, ["PARANA", "RIO GRANDE DO SUL ", "SANTA CATARINA"]]
def soma(linha, regiao):
    x=0
    for i in regiao.columns:
        x+=linha[i]
    return x
```

```
TotalNorte=df.apply(soma, axis=1, regiao=norte)
TotalSul=df.apply(soma, axis=1, regiao=sul)
print(TotalNorte.sum())
print(TotalSul.sum())

#plt.bar("Norte", df["TotalNorte"].sum())
#plt.show()
```

```
2366.0
2384.0
```

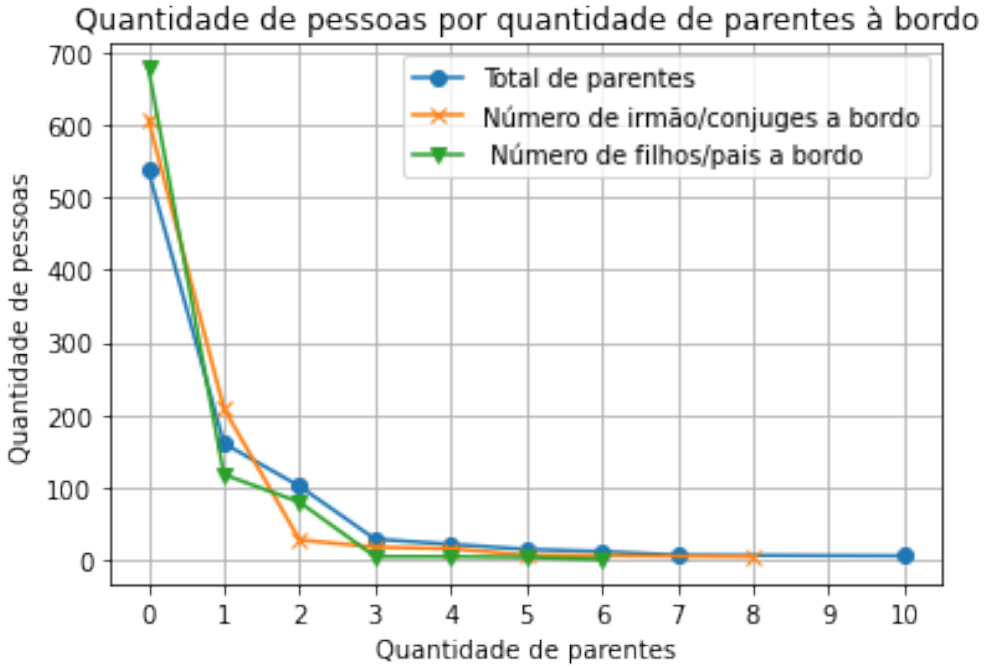
1.10 Aplicando ao Titanic

```
import pandas as pd
titanic = pd.read_csv("data/titanic_1_aula.csv")
x=[titanic.Survived[titanic["Survived"] == 1].count(), titanic.Survived[titanic["Survived"] == 0].count()]
y=["Survived", "Not Survived"]
plt.bar(y,x,0.5,align= "center", edgecolor="black", color="pink" )
plt.title("Quantidade de pessoas sobreviveram ou não.")
plt.xticks(rotation = '90')
plt.show()
```




```
x=sorted(titanic["Relatives"].unique())
x2=sorted(titanic["SibSp"].unique())
x3=sorted(titanic["Parch"].unique())
y=titanic["Relatives"].value_counts()
y2=titanic["SibSp"].value_counts()
y3=titanic["Parch"].value_counts()
plt.plot(x,y, marker="o", label="Total de parentes")
plt.plot(x2,y2, marker="x", label="Número de irmão/conjuges a bordo")
plt.plot(x3,y3, marker="v", label=" Número de filhos/pais a bordo")
plt.legend() #fontsize=10 / prop={"size":10}
plt.xticks([ 0,1,2,3,4,  5,  6,  7,8,9, 10],rotation="horizontal")
plt.title("Quantidade de pessoas por quantidade de parentes à bordo")
plt.xlabel("Quantidade de parentes")
plt.ylabel("Quantidade de pessoas")
```

```
plt.grid()
plt.show()
```




```
import numpy as np
x= sorted(round(titanic.Age[titanic["Sex"]=="male"].fillna(29)).unique())
x2= sorted(round(titanic.Age[titanic["Sex"]=="female"].fillna(29)).unique())
y= titanic[titanic["Age"]==25.0]

array=np.intersect1d(x,x2)


lista=[]
lista2=[]
```

```
titanic2=titanic[titanic["Sex"]=="male"]
index=list(titanic2.index)

for i in range(len(index)):
    y=titanic2.Age[index[i]]
    if y in array:
        lista.append(y)


titanic3=titanic[titanic["Sex"]=="female"]
index=list(titanic3.index)

for i in range(len(index)):
    y=titanic3.Age[index[i]]
    if y in array:
        lista2.append(y)

df=pd.DataFrame({"Age":lista})
df2=pd.DataFrame({"Age":lista2})

plt.scatter(array,df["Age"].value_counts().sort_index(), label = 'Homem')
plt.scatter(array,df2["Age"].value_counts().sort_index(), label = 'Mulher')
plt.legend() #fontsize=10 / prop={"size":10}
plt.xticks(np.arange(0,63,3),rotation=45)
plt.title("Quantidade de pessoas por idade")
plt.xlabel("Idade")
plt.ylabel("Quantidade de pessoas")
plt.show()
```