Data Literacy

Tutor : Ghassan Al-Falouji
Supervisor : Prof. Dr. Roland Mandl

In [1]:
```
## CSS coloring for the dataframe tables

from IPython.core.display import HTML

css = open('../style/style-table.css').read() + open('../style/style-notebook.css').read()
HTML('<style>{}</style>'.format(css))
```

Out[1]:

Data Visualization

Table of Contents

Introduction [[3](#ref3), [4](#ref4)]

Data science is the practice of **deriving insights from data**, enabled by:

- statistical modeling,
- computational methods,
- interactive visual analysis,
- and domain-driven problem solving.

Visualization is an integral part of data science, and essential to enable sophisticated analysis of data.

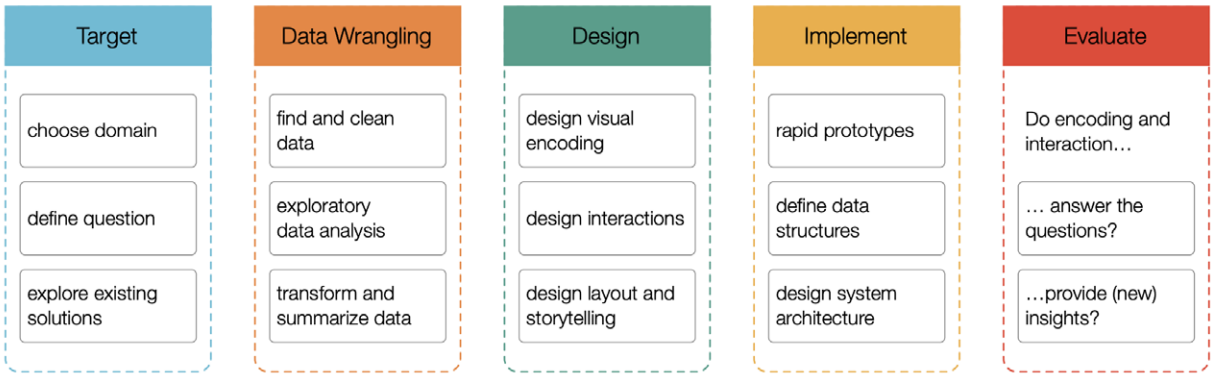
Visualization Goals [[4](#ref4)]

Essentially there are three goals:

- **Data Exploration** : Find the unknown
- **Data Analysis** : Check hypothesis
- **Presentation** : Communicate and advertise

To achieve these goals, the following five-step model is suggested :

source:[4]



Step 1. Target:
one is required to isolate a specific target or question that is to be the subject of evaluation.

Step 2. Data Wrangling:
Data Wrangling represent 90% of the work load in data science. This procedure involves:

- getting the data into a workable format,
- performing exploratory data analysis to understand their data set, which may involve various ways of summarizing or plotting the data.

Step 3. Design:
Design stage, which involves the development of a story that you want to tell with the data. This is closely linked back to the target we defined. What is the message we are trying to communicate? This will also likely depend on who your audience is, as well as the level of objectivity of the analysis.

Step 4. Implement:
The fourth step involves the implementation of the visualization.

Step 5. Evaluate:
The fifth stage is essentially a review stage, you look at your implementation and decide whether it sends the message that you want to communicate, or answers the question you set out to answer.

In this part two visualization libraries will be presented:

1. `matplotlib`
   Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.
2. `seaborn`
   Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

> Rougier et al. share their ten simple rules for drawing better figures, and use matplotlib to provide illustrative examples. As you read this paper, reflect on what you learned in the first module of the course -- principles from Tufte and Cairo -- and consider how you might realize these using matplotlib.
>
> Rougier NP, Droettboom M, Bourne PE (2014) Ten Simple Rules for Better Figures. PLoS Comput Biol 10(9): e1003833. doi:10.1371/journal.pcbi.1003833

`matplotlib` [[1](#ref1)]

Matplotlib is a multi-platform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack. It was conceived by John Hunter in 2002, originally as a patch to IPython for enabling interactive MATLAB-style plotting via gnuplot from the IPython command line. IPython's creator, Fernando Perez, was at the time scrambling to finish his PhD, and let John know he wouldn't have time to review the patch for several months. John took this

as a cue to set out on his own, and the Matplotlib package was born in 2003.

In recent years, however, the interface and style of Matplotlib have begun to age. Newer tools like `ggplot` and `ggvis` in the R language, along with web visualization toolkits based on `D3js` and `HTML5` canvas, often make `Matplotlib` feel clunky and old-fashioned. Still, Matplotlib's strength is well-tested, cross-platform graphics engine.

More modern APIs—for example, `Seaborn`, `ggpy`, `HoloViews`, `Altair`, and even `Pandas` itself can be used as wrappers around `Matplotlib`'s API. Even with wrappers like these, it is still often useful to dive into `Matplotlib`'s syntax to adjust the final plot output.

Basics

Installation and import

- Installation

In [2]:
```
!pip install -U matplotlib
```

```
Requirement already up-to-date: matplotlib in c:\users\g\appdata\local\p
rograms\python\python36\lib\site-packages (3.1.1)
Requirement already satisfied, skipping upgrade: pyparsing!=2.0.4,!=2.1.
2,!=2.1.6,>=2.0.1 in c:\users\g\appdata\local\programs\python\python36\l
ib\site-packages (from matplotlib) (2.4.0)
Requirement already satisfied, skipping upgrade: python-dateutil>=2.1 in
c:\users\g\appdata\local\programs\python\python36\lib\site-packages (fro
m matplotlib) (2.8.0)
Requirement already satisfied, skipping upgrade: cycler>=0.10 in c:\user
s\g\appdata\local\programs\python\python36\lib\site-packages (from matpl
otlib) (0.10.0)
Requirement already satisfied, skipping upgrade: kiwisolver>=1.0.1 in c:
\users\g\appdata\local\programs\python\python36\lib\site-packages (from
matplotlib) (1.1.0)
Requirement already satisfied, skipping upgrade: numpy>=1.11 in c:\users
\g\appdata\local\programs\python\python36\lib\site-packages (from matplo
tlib) (1.17.0)
Requirement already satisfied, skipping upgrade: six>=1.5 in c:\users\g\
appdata\local\programs\python\python36\lib\site-packages (from python-da
teutil>=2.1->matplotlib) (1.12.0)
Requirement already satisfied, skipping upgrade: setuptools in c:\users\
g\appdata\local\programs\python\python36\lib\site-packages (from kiwisol
ver>=1.0.1->matplotlib) (41.0.1)
```

- Importing

In [3]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

The `plt` interface is what we will use most often.

Setting Styles

`plt.style` directive to choose appropriate aesthetic styles for our figures. Here we will set the `classic` style.

In [4]:
```
plt.style.use('classic')
```

Further matplotlib styles can be found HERE.

`matplotlib.pyplot.show()`

The best use of Matplotlib differs depending on how you are using it; roughly, the three applicable contexts are using Matplotlib in a script, in an IPython terminal, or in an IPython notebook.

Plotting from a script

If you are using Matplotlib from within a script, the function `plt.show()` is required to generate/show your plot.

`plt.show()` starts an event loop, looks for all currently active figure objects, and opens one or more interactive windows that display your figure or figures.

So, for example, you may have a file called *myplot.py* containing the following:

```
# ------- file: myplot.py ------
import matplotlib.pyplot as plt
import numpy as np

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

plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))

plt.show()
```

You can then run this script from the command-line prompt, which will result in a window opening with your figure displayed:

```
$ python myplot.py
```

In [5]:
```
! python myplot.py
```

```
Figure(640x480)
```

> One thing to be aware of: the `plt.show()` command should be used *only once* per Python session, and is most often seen at the very end of the script. Multiple `show()` commands can lead to unpredictable backend-dependent behavior, and should mostly be avoided.

Plotting from IPython shell

IPython is built to work well with Matplotlib if you specify Matplotlib mode. To enable this mode, you can use the %matplotlib magic command after starting ipython:

```
In [1]: %matplotlib
Using matplotlib backend: TkAgg

In [2]: import matplotlib.pyplot as plt
```

At this point, any `plt` plot command will cause a figure window to open, and further commands can be run to update the plot. Some changes (such as modifying properties of lines that are already drawn) will not draw automatically: to force an update, use `plt.draw()`. Using `plt.show()` in Matplotlib mode is not required.

Plotting from Jupyter Notebook

Plotting interactively within an IPython notebook can be done with the `%matplotlib` command, and works in a similar way to the IPython shell. In the IPython notebook, you also have the option of embedding graphics directly in the notebook, with two possible options:

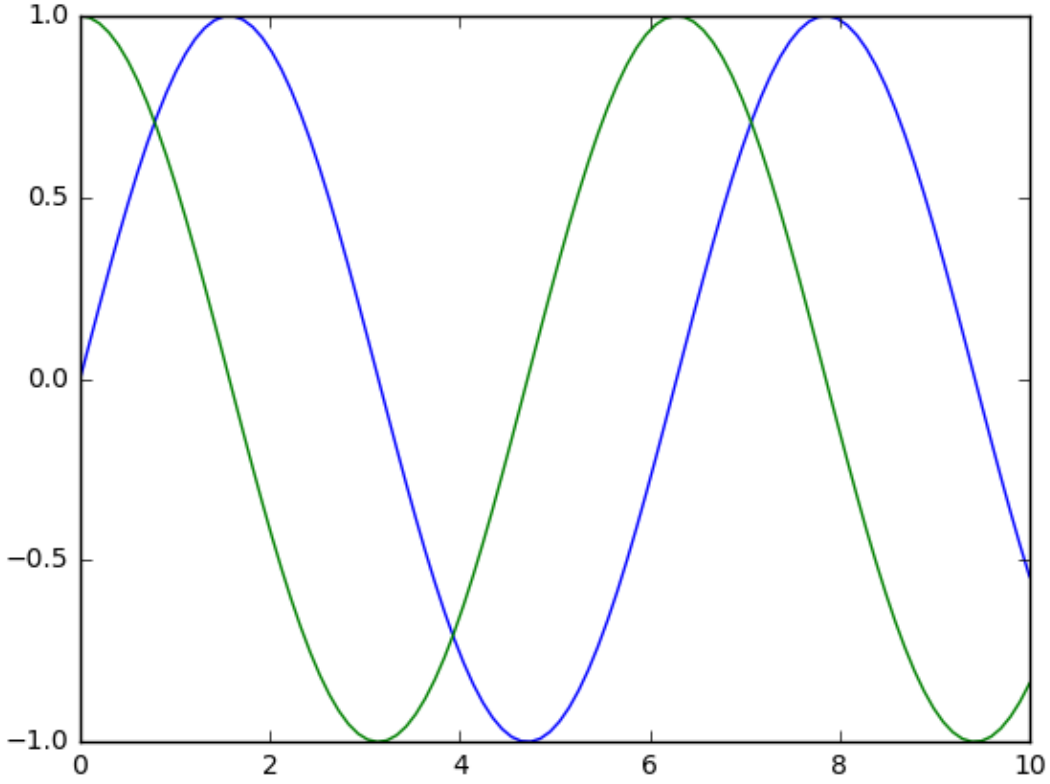
1. `%matplotlib notebook` will lead to *interactive* plots embedded within the notebook
2. `%matplotlib inline` will lead to *static* images of your plot embedded in the notebook

- As example:

In [59]:
```
import matplotlib.pyplot as plt
import numpy as np

# Allow dynamic properties, such as saving and zooming

#==============================# CODE HERE #==============================
==#

# Also, try the second option, where the plot is shown statically plotte
d
# %matplotlib inline
```

In [60]:
```
x = np.linspace(0, 10, 100)
```

```
#============================# CODE HERE #============================
==#
#============================# CODE HERE #============================
==#
```



Out[60]: `[<matplotlib.lines.Line2D at 0x14805c7a940>]`

Matplotlib backend engine

Different backend engines can be used to generate the Matplotlib plots. Backend engines types:

- interactive backends:

```
GTK3Agg, GTK3Cairo, MacOSX, nbAgg, Qt4Agg, Qt4Cairo, Qt5Agg,
Qt5Cairo, TkAgg, TkCairo, WebAgg, WX, WXAgg, WXCairo
```

- non-interactive backends:

```
agg, cairo, pdf, pgf, ps, svg, template
```

- To get the default backend used in your jupyter notebook:

In [8]:
```
import matplotlib as mpl
```

```
#============================# CODE HERE #============================
==#
```

Out[8]: 'nbAgg'

- To change the backend engine:

As example, using the Qt5 engine. This requires installing `PyQt5`. The following bash code can be used to install `PyQt5` .

In [9]:
```
!pip install -U PyQt5
```

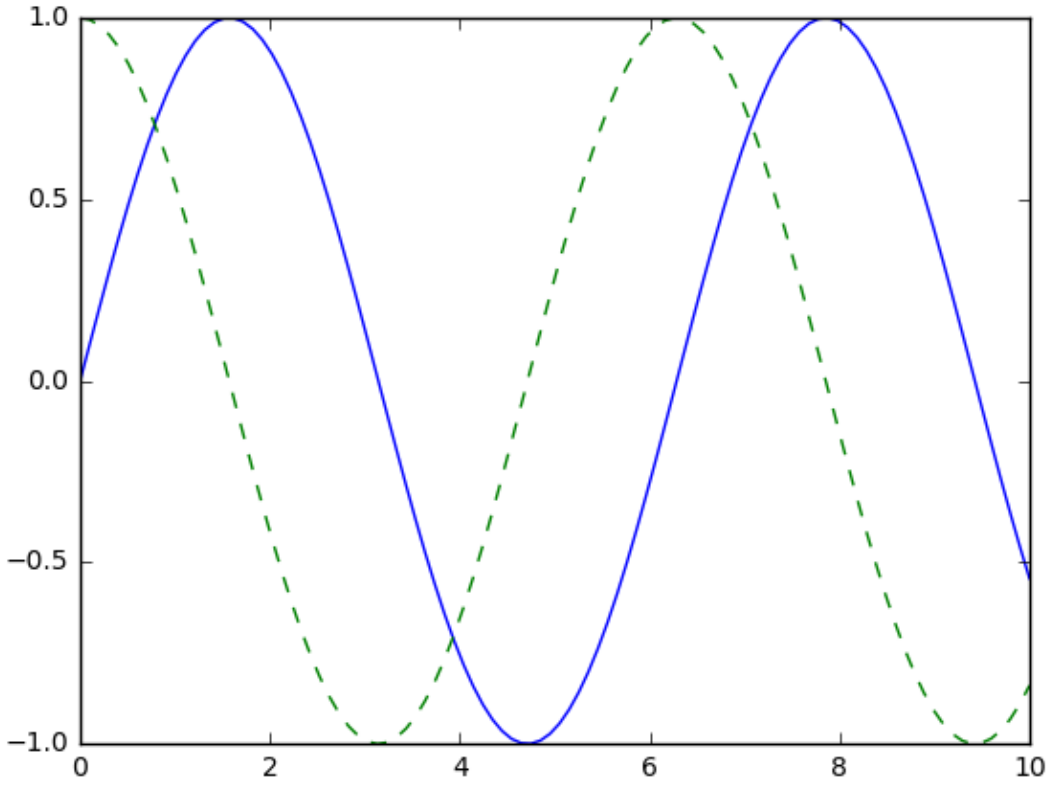
```
Requirement already up-to-date: PyQt5 in c:\users\g\appdata\local\progra
ms\python\python36\lib\site-packages (5.13.0)
Requirement already satisfied, skipping upgrade: PyQt5_sip<13,>=4.19.14
in c:\users\g\appdata\local\programs\python\python36\lib\site-packages (
from PyQt5) (4.19.18)
```

In [10]:
```
# Make sure that QT5 is used
mpl.use('Qt5Agg')
```

Saving a figure

In [11]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt

mpl.use('Qt5Agg');
%matplotlib notebook
#inline
```

In [61]:
```
import numpy as np
x = np.linspace(0, 10, 100)

#============================# CODE HERE #============================
==#
#============================# CODE HERE #============================
==#
#============================# CODE HERE #============================
==#
```



- Save the plot in the current directory under the name `my_figure.png`:

In [13]:
```
#============================# CODE HERE #============================
==#
```

You will find now the figure in the directory were you mentioned:

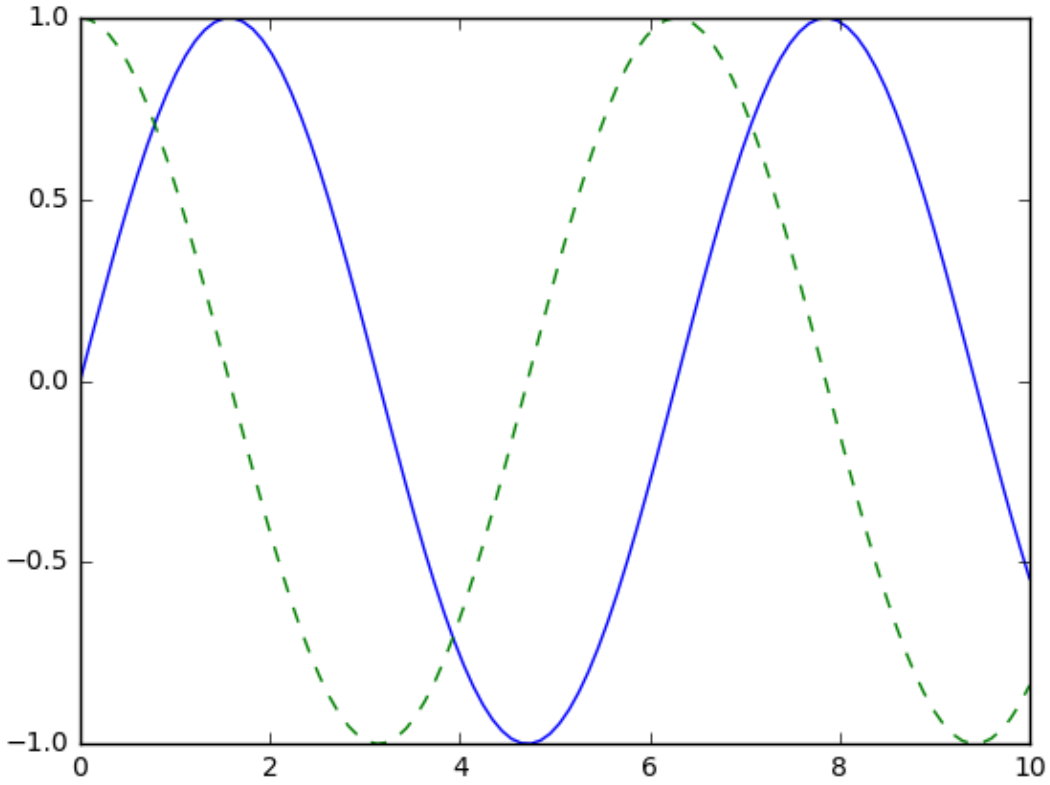
In [14]:
```
!ls -lh images/my_figure.png
```

```
-rw-r--r-- 1 G 197609 30K Sep  1  2019 images/my_figure.png
```

- You can import an image the Jupyter notebook

In [15]:
```
from IPython.display import Image as im

#============================# CODE HERE #============================
==#
```

Out[15]:



Depending on what backends you have installed, many different file formats are available.
The list of supported file types can be found for your system by using the following method of the figure canvas object:

In [16]:
```
#============================# CODE HERE #============================
==#
```

Out[16]:
```
{'ps': 'Postscript',
 'eps': 'Encapsulated Postscript',
 'pdf': 'Portable Document Format',
 'pgf': 'PGF code for LaTeX',
 'png': 'Portable Network Graphics',
 'raw': 'Raw RGBA bitmap',
 'rgba': 'Raw RGBA bitmap',
 'svg': 'Scalable Vector Graphics',
 'svgz': 'Scalable Vector Graphics',
 'jpg': 'Joint Photographic Experts Group',
 'jpeg': 'Joint Photographic Experts Group',
 'tif': 'Tagged Image File Format',
 'tiff': 'Tagged Image File Format'}
```

General Concepts[[6](#ref6)]

Everything in matplotlib is organized in a hierarchy. At the top of the hierarchy is the matplotlib "state-machine environment" which is provided by the `matplotlib.pyplot` module. At this level, simple functions are used to add plot elements (lines, images, text, etc.) to the current

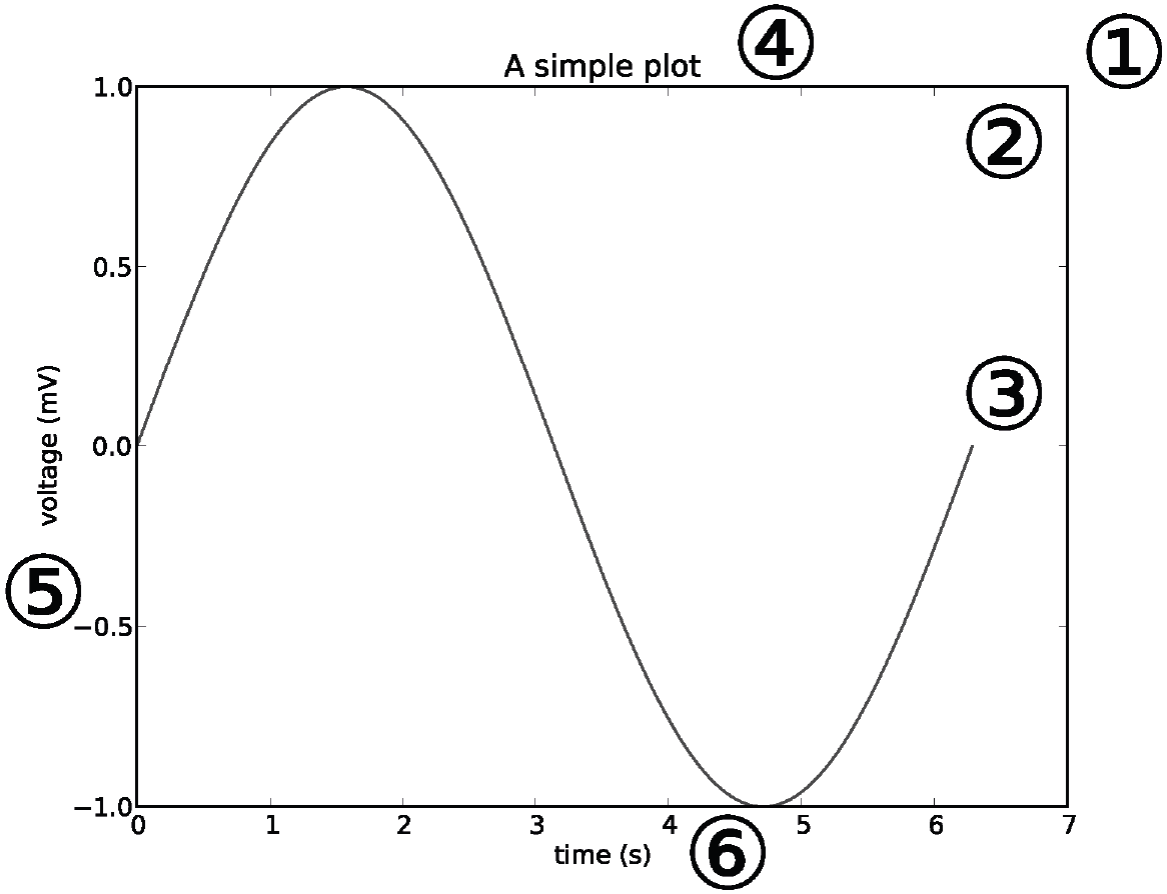
axes in the current figure.

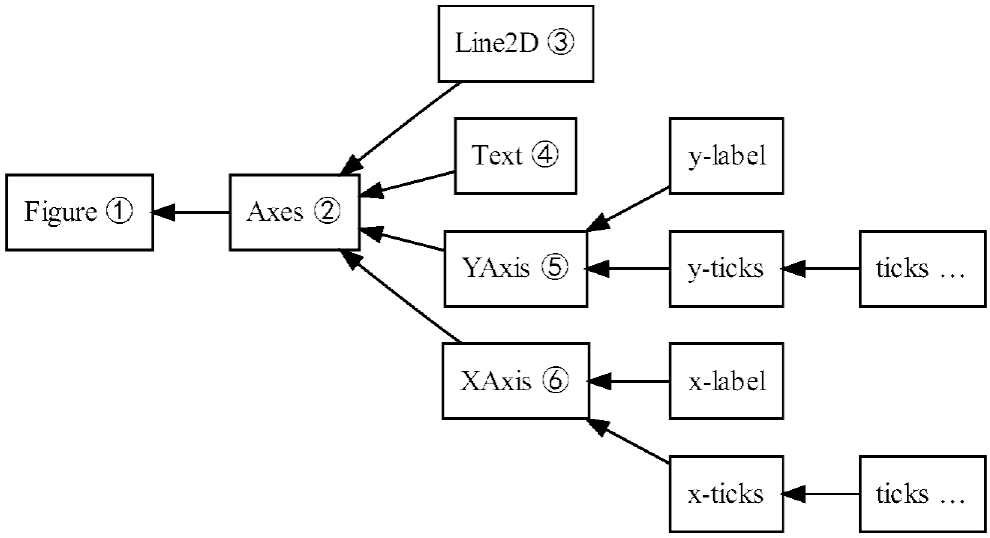
> Note : Pyplot’s state-machine environment behaves similarly to MATLAB and should be most familiar to users with MATLAB experience.

The next level down in the hierarchy is the first level of the object-oriented interface, in which pyplot is used only for a few functions such as **figure** creation. The user uses pyplot to create figures, and through those **figures**, one or more **axes** objects can be created. These **axes** objects are then used for most plotting actions.

The figure below shows an example of matplotlib's hierarchy:

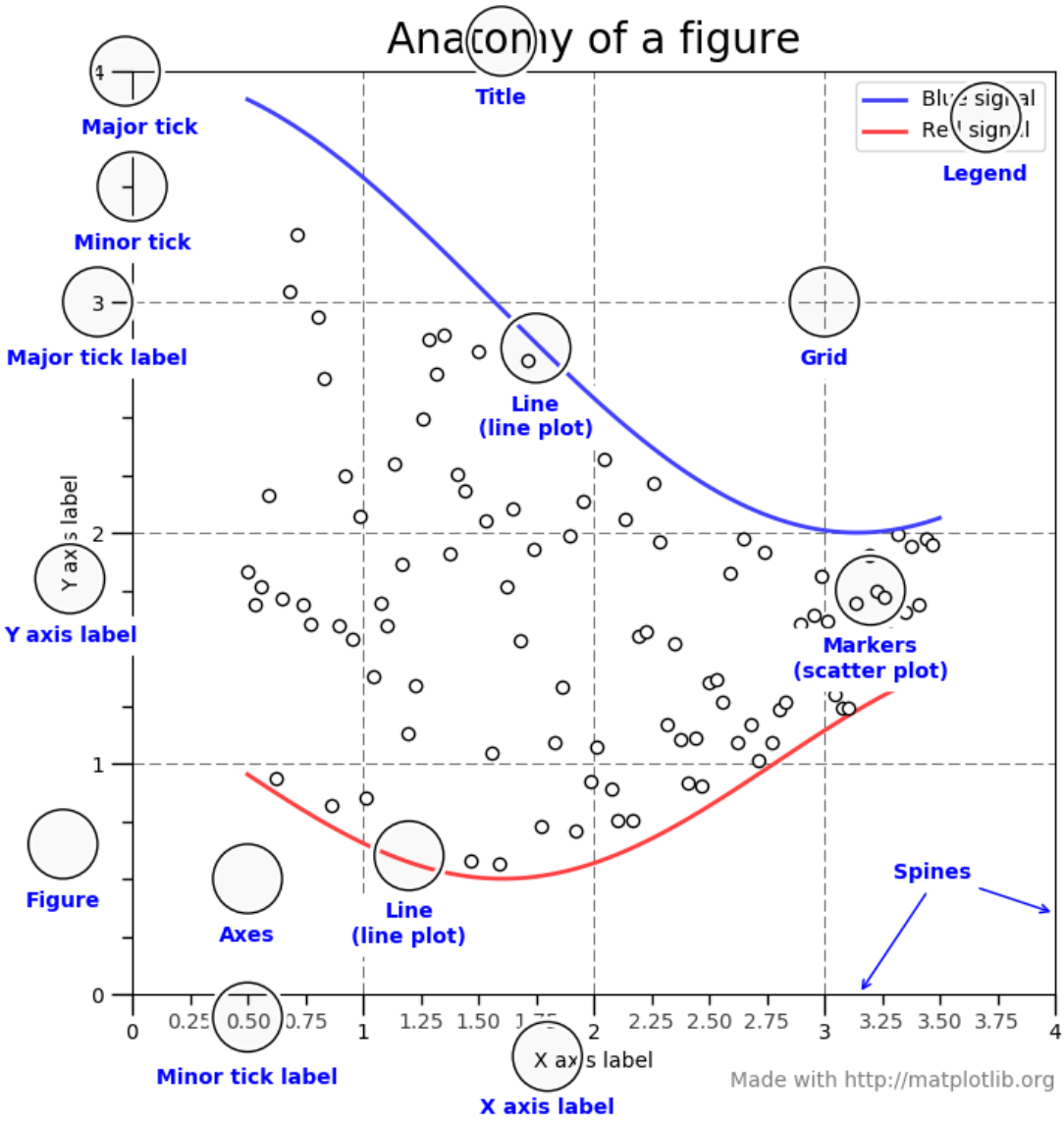
source:*https://bit.ly/2ZJGdYq*





Matplotlib Figure Anatomy

source:[6]



Figure

The whole is a **figure**. The figure keeps track of all the child:

- **Axes** : A figure can have any number of **Axes**
- 'special' artists (such as: **titles**, figure **legend**s, etc),
- and the canvas (Canvas is the object that generated the plotting. It is more-or-less invisible to the user.)

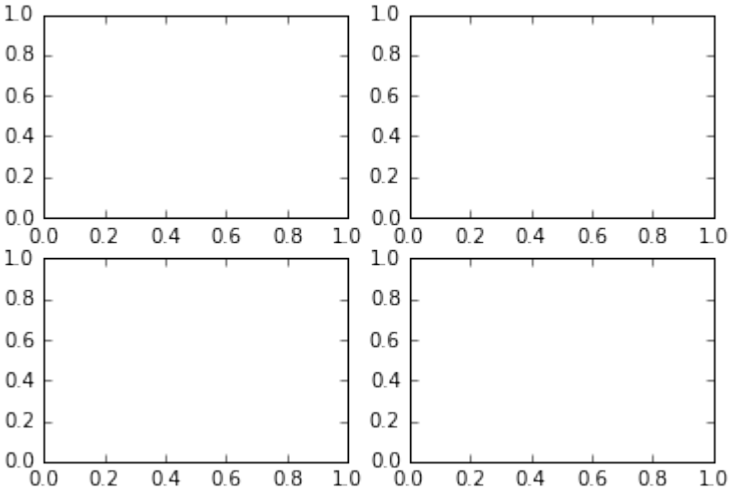
- Example:

In [62]:
```
%matplotlib inline
```

In [63]:
```
# an empty figure with no axes

#============================# CODE HERE #============================
==#

# a figure with a 2x2 grid of Axes
```

```
#==============================# CODE HERE #============================
==#
```

```
<Figure size 432x288 with 0 Axes>
```



Axis

They take care of setting the graph limits and generating the 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.

Artist

Basically everything you can see on the figure is an artist (even the Figure, `Axes`, and `Axis` objects). This includes `Text` objects, `Line2D` objects, `collection` objects, `Patch` objects .. . 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.

Basic operations

- `text()` - add text at an arbitrary location to the Axes; `matplotlib.axes.Axes.text()` in the API.
- `xlabel()` - add a label to the x-axis; `matplotlib.axes.Axes.set_xlabel()` in the API.
- `ylabel()` - add a label to the y-axis; `matplotlib.axes.Axes.set_ylabel()` in the API.
- `title()` - add a title to the Axes; `matplotlib.axes.Axes.set_title()` in the API.
- `figtext()` - add text at an arbitrary location to the Figure; `matplotlib.figure.Figure.text()` in the API.
- `suptitle()` - add a title to the Figure; `matplotlib.figure.Figure.suptitle()` in the API.
- `annotate()` - add an annotation, with optional arrow, to the Axes ;

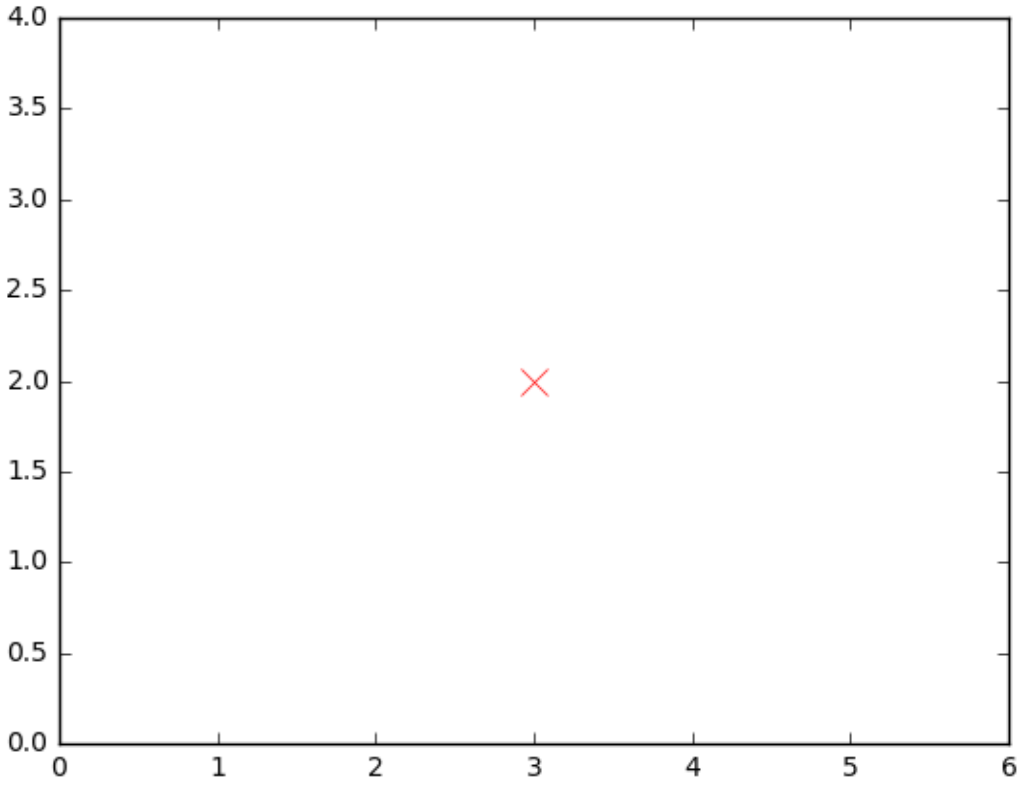
`matplotlib.axes.Axes.annotate()` in the API.

Line plot

Let's plot a point !

In [64]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.use('Qt5Agg')

#==============================# CODE HERE #============================
==#

#inline
```

In [65]:
```
# Create a figure

#==============================# CODE HERE #============================
==#

# plot a dot at the location (3,2) using x marker

#==============================# CODE HERE #============================
==#


# plot a circle arount the x

#==============================# CODE HERE #============================
==#


# create a handle for the current axis

#==============================# CODE HERE #============================
==#


# set axis properties [xmin, xmax, ymin, ymax]

#==============================# CODE HERE #============================
==#
```



In [51]:
```
# get all the child objects of ax axis
ax.get_children()
```

Out[51]:
```
[<matplotlib.lines.Line2D at 0x148040af278>,
 <matplotlib.lines.Line2D at 0x14803d03be0>,
 <matplotlib.spines.Spine at 0x14803d03940>,
 <matplotlib.spines.Spine at 0x14803c5cb00>,
 <matplotlib.spines.Spine at 0x14803c5ca20>,
 <matplotlib.spines.Spine at 0x14803c5c9b0>,
 <matplotlib.axis.XAxis at 0x14803d03780>,
 <matplotlib.axis.YAxis at 0x14803d03f60>,
 Text(0.5, 1, ''),
 Text(0.0, 1, ''),
 Text(1.0, 1, ''),
 <matplotlib.patches.Rectangle at 0x14804078c50>]
```

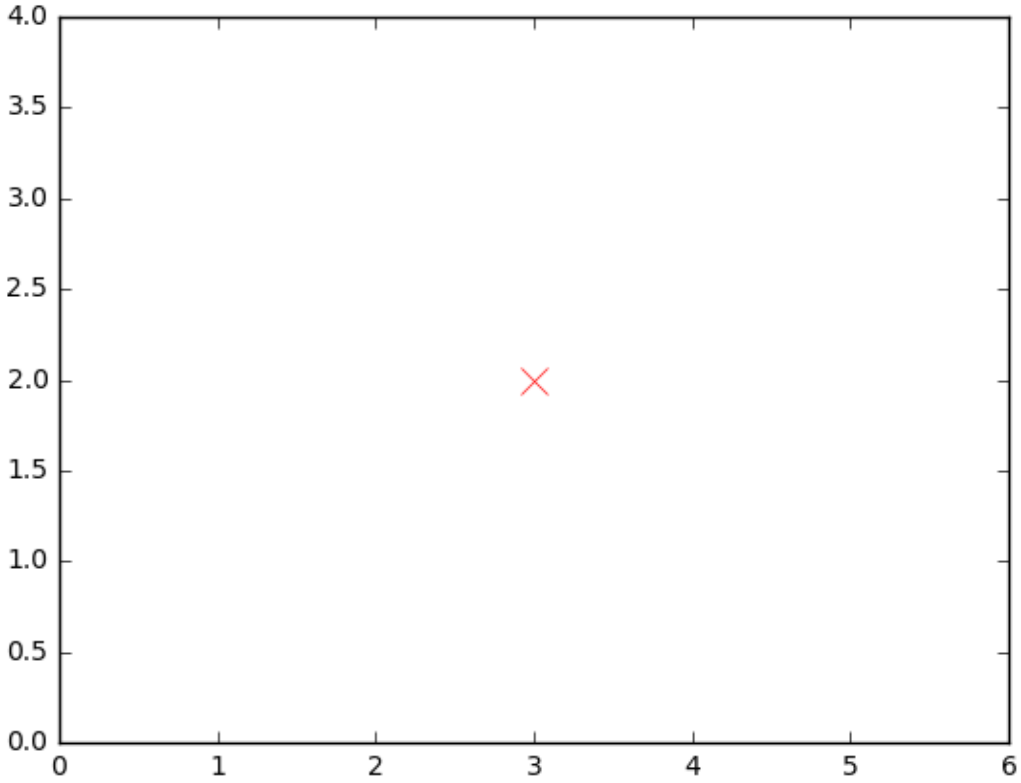
In [52]:
```
print(l1)
print(l2)
```

```
Line2D(dot)
Line2D(circle)
```

- Let's delete the circle and replot

In [53]:
```
# Required to replot
from IPython import display
```

In [66]:
```
#==============================# CODE HERE #==============================
==#

display.display(fig) # You can not re-draw if your plot in inlined !
```



Exercise

Plot two points (with marker "x", and color "red") with coordinates:

- P1 = (1,1)
- P2 = (3,3)

and plot the line connecting between them.

In [67]:
```
%matplotlib inline

### YOUR CODE HERE
#
#
###
```

Exercise

Generate two plots of a one period sine and cosine wave (i.e. [0,2*pi]). In this plot, set the color of the sine line to green, while the cosine is a blue dotted line.

In [164]:
```
plt.style.use('seaborn-whitegrid')
```

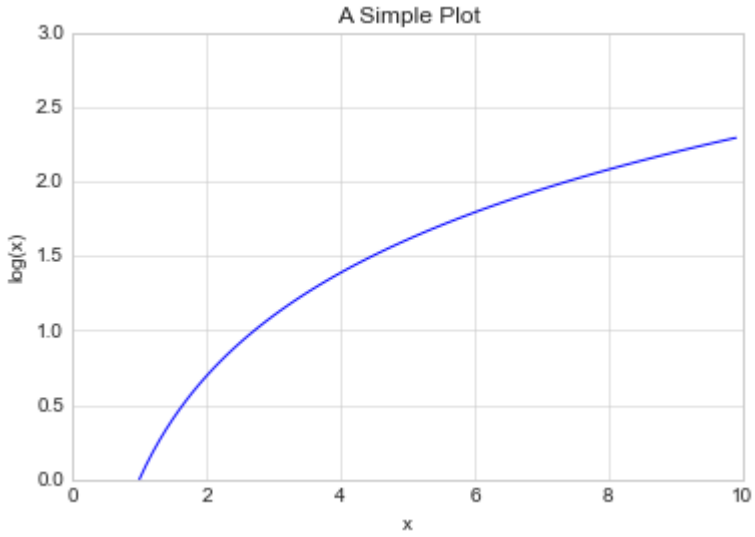
In [ ]:
```
### YOUR CODE HERE
#
#
###
plt.xlabel("x")
plt.ylabel("Amplitude")
plt.ylim([-2,2]);
plt.xlim([0, x[-1]]);
plt.title("Wave");
plt.legend(['sin(x)', 'cos(x)'], loc=3, frameon=False, title='Legend');
```

While most `plt` functions translate directly to `ax` methods (such as `plt.plot()` → `ax.plot()`, `plt.legend()` → `ax.legend()`, etc.), this is not the case for all commands. In particular, functions to set limits, labels, and titles are slightly modified. For transitioning between MATLAB-style functions and object-oriented methods, make the following changes:

- `plt.xlabel()` → `ax.set_xlabel()`
- `plt.ylabel()` → `ax.set_ylabel()`
- `plt.xlim()` → `ax.set_xlim()`
- `plt.ylim()` → `ax.set_ylim()`
- `plt.title()` → `ax.set_title()`

In the object-oriented interface to plotting, rather than calling these functions individually, it is often more convenient to use the `ax.set()` method to set all these properties at once:

In [182]:
```
x=np.arange(1,10,0.1)

#==============================# CODE HERE #==============================
==#


#==============================# CODE HERE #==============================
==#


#==============================# CODE HERE #==============================
==#
```



Scatter plot

You can check scatter plot syntax:

In [105]:
```
plt.scatter?
```

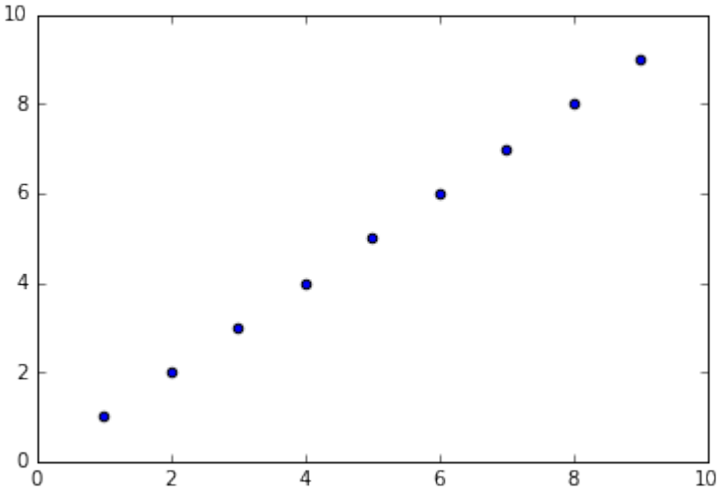
As example:

In [106]:
```
import numpy as np
```

In [111]:
```
x = np.arange(1,10)
y = x

# Plot

#================================# CODE HERE #================================
==#


#================================# CODE HERE #================================
==#


# print the child objects of the axes

#================================# CODE HERE #================================
==#
```

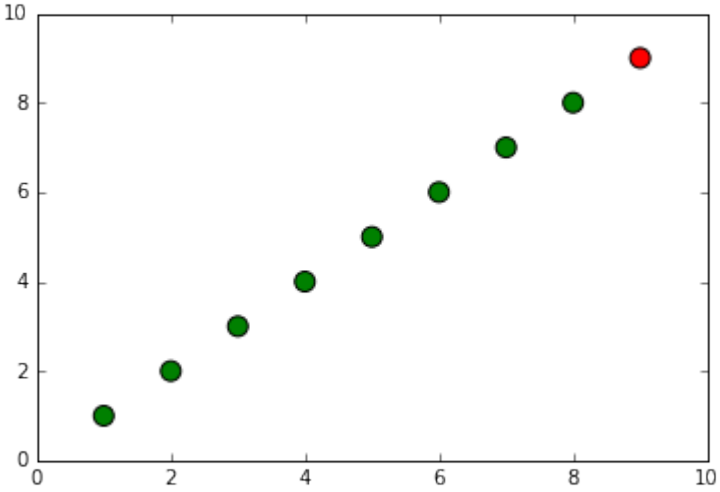
Out[111]:
```
[<matplotlib.collections.PathCollection at 0x1480ab494a8>,
 <matplotlib.spines.Spine at 0x1480ac87278>,
 <matplotlib.spines.Spine at 0x1480ac87358>,
 <matplotlib.spines.Spine at 0x1480ac87438>,
 <matplotlib.spines.Spine at 0x1480ac87518>,
 <matplotlib.axis.XAxis at 0x1480ac87208>,
 <matplotlib.axis.YAxis at 0x1480ac878d0>,
 Text(0.5, 1.0, ''),
 Text(0.0, 1.0, ''),
```

```
Text(1.0, 1.0, ''),
<matplotlib.patches.Rectangle at 0x1480acabd68>]
```



Recreating the previous scatter with list of defined colors:

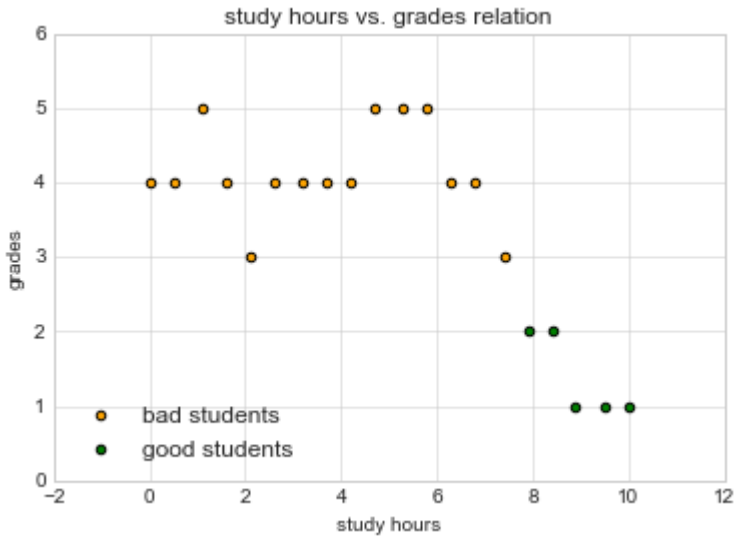
In [113]:
```
#==============================# CODE HERE #============================
==#


#==============================# CODE HERE #============================
==#


#Scatter plot, set the point size to 100 and colors from the list

#==============================# CODE HERE #============================
==#
```



- Example: Let's generate random data representing the grades and the study hours of two groups of students and plot their data:

In [171]:
```
study_hours = np.round(np.linspace(0,10,20),1)
# generate grades
less_study_grades = np.random.randint(3,6,size=15)
more_study_grades = np.random.randint(1,3,size=5)
grades = np.concatenate([less_study_grades, more_study_grades])
```

In [205]:
```
fig=plt.figure();
ax=plt.axes();

#==============================# CODE HERE #==============================
==#


#==============================# CODE HERE #==============================
==#

# Set x and y labels
ax.set_xlabel("study hours"); # also can be done : plt.xlable("study hou
rs")
ax.set_ylabel("grades");  # also can be done : plt.ylable("grades")
# Activate the legened
ax.legend(loc=3) # also can be done : plt.legend

# Add title
plt.title("study hours vs. grades relation");
```



Show all figure properties and methods

Get properties of the figure

In [206]:
```
# ---------- figure properties ----------

#==============================# CODE HERE #==============================
==#
```

Out[206]:
```
{'agg_filter': None,
 'alpha': None,
 'animated': False,
 'axes': [<matplotlib.axes._subplots.AxesSubplot at 0x1480c7089e8>],
 'children': [<matplotlib.patches.Rectangle at 0x1480c7082b0>,
  <matplotlib.axes._subplots.AxesSubplot at 0x1480c7089e8>],
 'clip_box': None,
 'clip_on': True,
 'clip_path': None,
```

```
 'constrained_layout': False,
 'constrained_layout_pads': (0.04167, 0.04167, 0.02, 0.02),
 'contains': None,
 'default_bbox_extra_artists': [<matplotlib.axes._subplots.AxesSubplot a
t 0x1480c7089e8>,
  <matplotlib.collections.PathCollection at 0x1480c648eb8>,
  <matplotlib.collections.PathCollection at 0x1480c648a58>,
  <matplotlib.spines.Spine at 0x1480c6ea320>,
  <matplotlib.spines.Spine at 0x1480c6ea668>,
  <matplotlib.spines.Spine at 0x1480c6eae80>,
  <matplotlib.spines.Spine at 0x1480c6ea8d0>,
  <matplotlib.axis.XAxis at 0x1480c6ea198>,
  <matplotlib.axis.YAxis at 0x1480c6acd30>,
  Text(0.5, 1, 'study hours vs. grades relation'),
  Text(0.0, 1, ''),
  Text(1.0, 1, ''),
  <matplotlib.legend.Legend at 0x1480c708320>,
  <matplotlib.patches.Rectangle at 0x1480c6a50b8>],
 'dpi': 72.0,
 'edgecolor': (1.0, 1.0, 1.0, 0.0),
 'facecolor': (1.0, 1.0, 1.0, 1.0),
 'figheight': 4.0,
 'figure': None,
 'figwidth': 6.0,
 'frameon': True,
 'gid': None,
 'in_layout': True,
 'label': '',
 'path_effects': [],
 'picker': None,
 'rasterized': None,
 'size_inches': array([6., 4.]),
 'sketch_params': None,
 'snap': None,
 'tight_layout': False,
 'transform': <matplotlib.transforms.IdentityTransform at 0x1480c828320>
,
 'transformed_clip_path_and_affine': (None, None),
 'url': None,
 'visible': True,
 'window_extent': <matplotlib.transforms.TransformedBbox at 0x1480c70889
8>,
 'zorder': 0}
```

In [207]:
```
# ---------- figure methods ----------
fig.__dir__() # [method for method in dir(fig)]
```

Out[207]:
```
['_stale',
 'stale_callback',
 'figure',
 '_transform',
 '_transformSet',
 '_visible',
 '_animated',
 '_alpha',
 'clipbox',
 '_clippath',
 '_clipon',
 '_label',
 '_picker',
 '_contains',
```

```
'_rasterized',
'_agg_filter',
'_mouseover',
'eventson',
'_oid',
'_propobservers',
'_remove_method',
'_url',
'_gid',
'_snap',
'_sketch',
'_path_effects',
'_sticky_edges',
'_in_layout',
'callbacks',
'bbox_inches',
'dpi_scale_trans',
'_dpi',
'bbox',
'transFigure',
'patch',
'canvas',
'_suptitle',
'subplotpars',
'_layoutbox',
'_constrained_layout_pads',
'_constrained',
'_tight',
'_tight_parameters',
'_axstack',
'suppressComposite',
'artists',
'lines',
'patches',
'texts',
'images',
'legends',
'_axobservers',
'_cachedRenderer',
'_align_xlabel_grp',
'_align_ylabel_grp',
'_gridspecs',
'number',
'__module__',
'__doc__',
'__str__',
'__repr__',
'__init__',
'_repr_html_',
'show',
'_get_axes',
'axes',
'_get_dpi',
'_set_dpi',
'dpi',
'get_tight_layout',
'set_tight_layout',
'get_constrained_layout',
'set_constrained_layout',
'set_constrained_layout_pads',
'get_constrained_layout_pads',
```

```
'autofmt_xdate',
'get_children',
'contains',
'get_window_extent',
'suptitle',
'set_canvas',
'figimage',
'set_size_inches',
'get_size_inches',
'get_edgecolor',
'get_facecolor',
'get_figwidth',
'get_figheight',
'get_dpi',
'get_frameon',
'set_edgecolor',
'set_facecolor',
'set_dpi',
'set_figwidth',
'set_figheight',
'set_frameon',
'frameon',
'delaxes',
'add_artist',
'_make_key',
'_process_projection_requirements',
'add_axes',
'add_subplot',
'_add_axes_internal',
'subplots',
'_remove_ax',
'clf',
'clear',
'draw',
'draw_artist',
'get_axes',
'legend',
'text',
'_set_artist_props',
'gca',
'sca',
'_gci',
'__getstate__',
'__setstate__',
'add_axobserver',
'savefig',
'colorbar',
'subplots_adjust',
'ginput',
'waitforbuttonpress',
'get_default_bbox_extra_artists',
'get_tightbbox',
'init_layoutbox',
'execute_constrained_layout',
'tight_layout',
'align_xlabels',
'align_ylabels',
'align_labels',
'add_gridspec',
'aname',
'zorder',
```

```
 '_prop_order',
 'remove',
 'have_units',
 'convert_xunits',
 'convert_yunits',
 'stale',
 '_get_clipping_extent_bbox',
 'add_callback',
 'remove_callback',
 'pchanged',
 'is_transform_set',
 'set_transform',
 'get_transform',
 'set_contains',
 'get_contains',
 'pickable',
 'pick',
 'set_picker',
 'get_picker',
 'get_url',
 'set_url',
 'get_gid',
 'set_gid',
 'get_snap',
 'set_snap',
 'get_sketch_params',
 'set_sketch_params',
 'set_path_effects',
 'get_path_effects',
 'get_figure',
 'set_figure',
 'set_clip_box',
 'set_clip_path',
 'get_alpha',
 'get_visible',
 'get_animated',
 'get_in_layout',
 'get_clip_on',
 'get_clip_box',
 'get_clip_path',
 'get_transformed_clip_path_and_affine',
 'set_clip_on',
 '_set_gc_clip',
 'get_rasterized',
 'set_rasterized',
 'get_agg_filter',
 'set_agg_filter',
 'set_alpha',
 'set_visible',
 'set_animated',
 'set_in_layout',
 'update',
 'get_label',
 'set_label',
 'get_zorder',
 'set_zorder',
 'sticky_edges',
 'update_from',
 'properties',
 'set',
 'findobj',
```

```
'get_cursor_data',
'format_cursor_data',
'mouseover',
'__dict__',
'__weakref__',
'__hash__',
'__getattribute__',
'__setattr__',
'__delattr__',
'__lt__',
'__le__',
'__eq__',
'__ne__',
'__gt__',
'__ge__',
'__new__',
'__reduce_ex__',
'__reduce__',
'__subclasshook__',
'__init_subclass__',
'__format__',
'__sizeof__',
'__dir__',
'__class__']
```

Exercise

After importing the iris dataset, use the scatter plot to view the **sepal** and **petal** sets, with the x-axis representing the **length** and y-axis for the **width**. Your plot should contain the following properties:

1. The **sepal** and **petal** color should be different
2. set labels to x- and y-axis using:
   - `axes.set_xlabel("length")` and
   - ``axex.set_ylabel("width")
3. show the legend: `axes.legend()`
4. Set a title for your plot: ``axes.set_title("iris dataset")

In [76]:
```
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
```

In [80]:
```
dataset = sns.load_dataset("iris")
dataset.head()
```

Out[80]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa

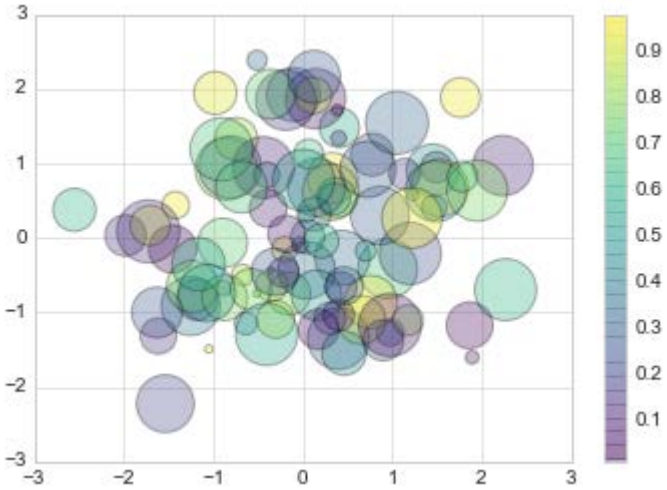
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [96]:
```
sepal=np.zeros([2,len(dataset)])
sepal[0]=dataset.sepal_length
sepal[1]=dataset.sepal_width
#
petal=np.zeros([2,len(dataset)])
petal[0]=dataset.petal_length
petal[1]=dataset.petal_width
```

In [ ]:
```
### YOUR CODE HERE
#
#
###
```

- Exploring some of the scatter plots properties:

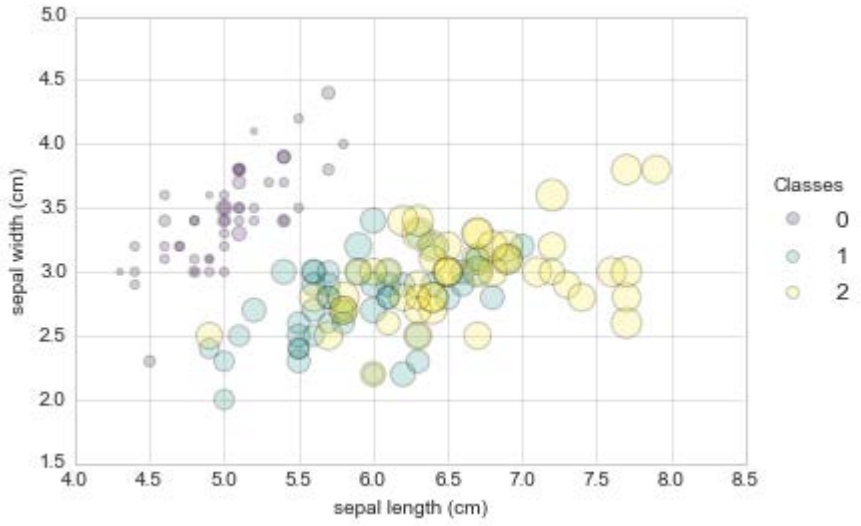
In [183]:
```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)


#==============================# CODE HERE #============================
==#

plt.colorbar();  # show color scale
```



Notice that the color argument is automatically mapped to a color scale (shown here by the `colorbar()` command), and that the size argument is given in pixels. In this way, the color and size of points can be used to convey information in the visualization, in order to visualize multidimensional data.

For example, we might use the Iris data from Scikit-Learn, where each sample is one of three types of flowers that has had the size of its petals and sepals carefully measured:

In [204]:
```
from sklearn.datasets import load_iris
iris = load_iris()
features = iris.data.T


#================================# CODE HERE #================================
==#


#================================# CODE HERE #================================
==#


#================================# CODE HERE #================================
==#


#================================# CODE HERE #================================
==#

# produce a legend with the unique colors from the scatter
# Put a legend to the right of the current axis

#================================# CODE HERE #================================
==#
```



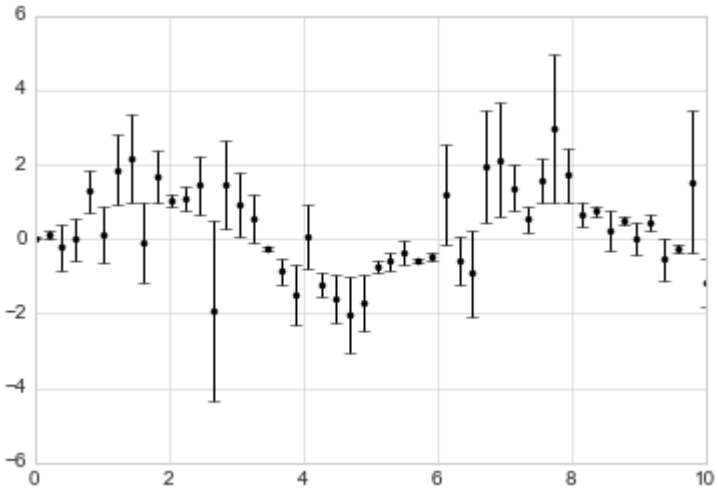
Visualizing Errors [[1](#ref1)]

For any scientific measurement, accurate accounting for errors is nearly as important, if not more important, than accurate reporting of the number itself. For example, imagine using some astrophysical observations to estimate the Hubble Constant, the local measurement of the expansion rate of the Universe. The current literature suggests a value of around 71 $\pm$ 2.5 (km/s)/Mpc (km/s)/Mpc, and what measured is a value of 74 $\pm$ 5 (km/s)/Mpc.

Basic Error-bars

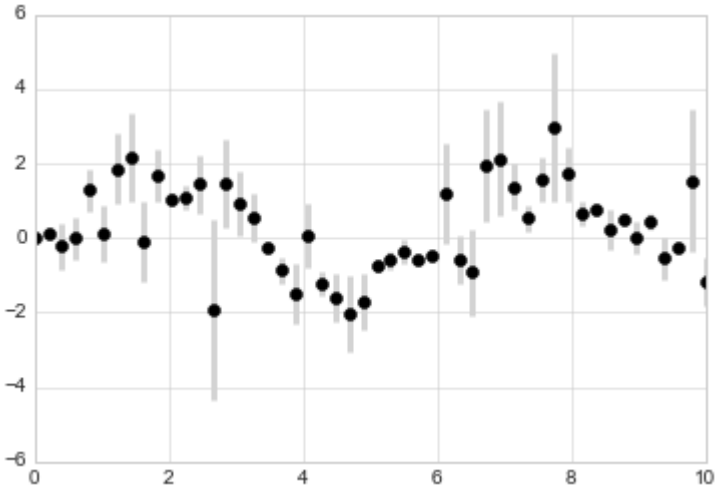
In [208]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
```

In [210]:
```
x = np.linspace(0, 10, 50)
dy = 0.8
noise = dy*np.random.randn(50)
y = np.sin(x)+noise


#==============================# CODE HERE #============================
==#
```



`fmt` is a format code controlling the appearance of lines and points.

In [211]:
```
#==============================# CODE HERE #============================
==#
```



In addition to these options, you can also specify horizontal errorbars (`xerr`), one-sided errorbars, and many other variants. For more information on the options available, refer to the `docstring` of `plt.errorbar` .

Continuous Errors

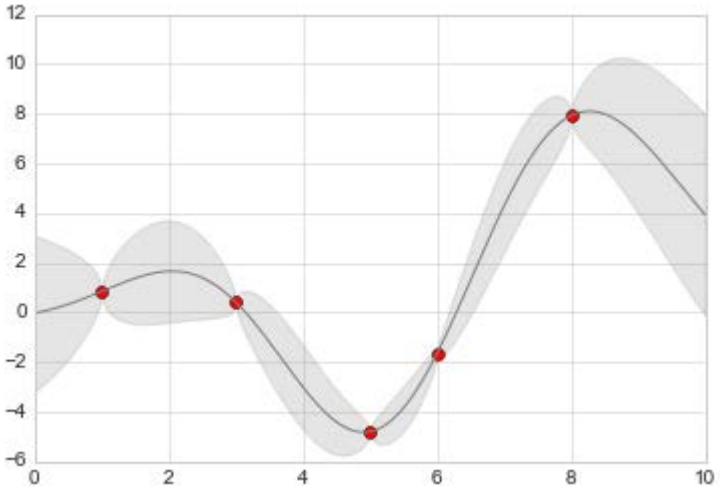
In some situations it is desirable to show errorbars on continuous quantities. Though Matplotlib does not have a built-in convenience routine for this type of application, it's relatively easy to combine primitives like `plt.plot` and `plt.fill_between` for a useful result.

Here we'll perform a simple *Gaussian process regression*, using the Scikit-Learn API. This is a method of fitting a very flexible non-parametric function to data with a continuous measure of the uncertainty.

> Gaussian Process Regression will be discussed in another notebook.

In [234]:
```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, ConstantKernel as C

# define the model and draw some data
model = lambda x: x * np.sin(x)
xdata = np.array([1, 3, 5, 6, 8])
ydata = model(xdata)

# Compute the Gaussian process fit
kernel = C(1.0, (1e-3, 1e3)) * RBF(10, (1e-2, 1e2))
gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)
gp.fit(xdata[:, np.newaxis], ydata)

xfit = np.linspace(0, 10, 1000)
yfit, MSE = gp.predict(xfit[:, np.newaxis], return_std=True)
dyfit = 2 * np.sqrt(MSE)  # 2*sigma ~ 95% confidence region
```

In [235]:
```
# Visualize the result

#==============================# CODE HERE #==============================
==#


#==============================# CODE HERE #==============================
==#


#==============================# CODE HERE #==============================
==#


#==============================# CODE HERE #==============================
==#
```



Contour plots [[1](#ref1)]

Sometimes it is useful to display three-dimensional data in two dimensions using contours or color-coded regions. There are three Matplotlib functions that can be helpful for this task:

1. `plt.contour` for contour plots,
2. `plt.contourf` for filled contour plots, and
3. `plt.imshow` for showing images.

In [236]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
import numpy as np
```

To demonstrate a contour plot using a function $z = f(x, y)$, the following function will be used: \begin{equation*} z=f(x,y)=10\ sin(x) + cos(10 + xy)*cos(x) \end{equation*}

In [237]:
```
def f(x, y):

#==============================# CODE HERE #==============================
==#
```

`plt.contour`

A contour plot can be created with the `plt.contour` function. It takes three arguments:

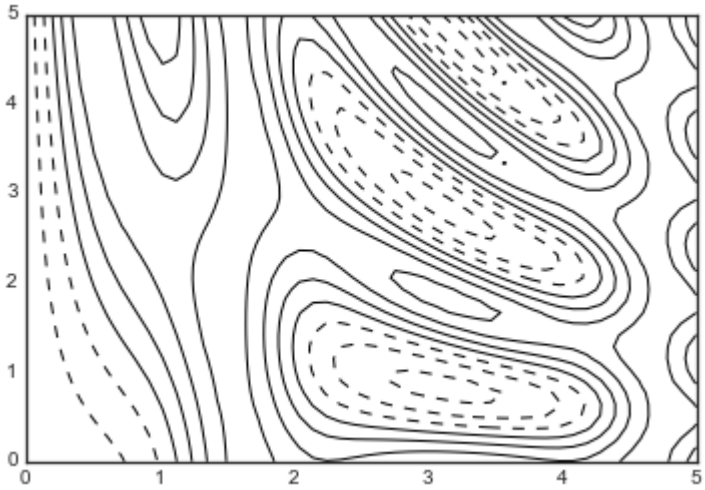
1. a grid of *x* values,
2. a grid of *y* values,
3. and a grid of *z* values.

The *x* and *y* values represent positions on the plot, and the *z* values will be represented by the contour levels.

`np.meshgrid` function will be used to create the *x* , *y* meshgrid:

In [238]:
```
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)


#==============================# CODE HERE #============================
==#


#==============================# CODE HERE #============================
==#
```

In [239]:
```
#==============================# CODE HERE #============================
==#
```



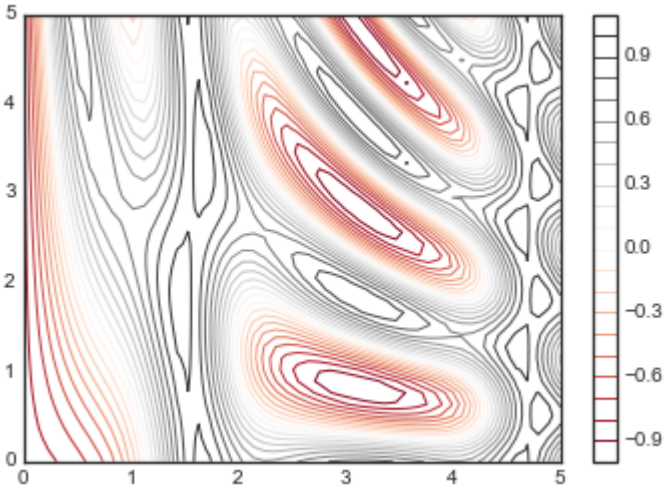
Notice that by default when a single color is used:

- negative values are represented by*dashed lines*, and
- positive values by *solid lines*.

Alternatively, the lines can be color-coded by specifying a colormap with the `cmap` argument.

Here, we'll also specify that we want more lines to be drawn: 20 equally spaced intervals within the data range:

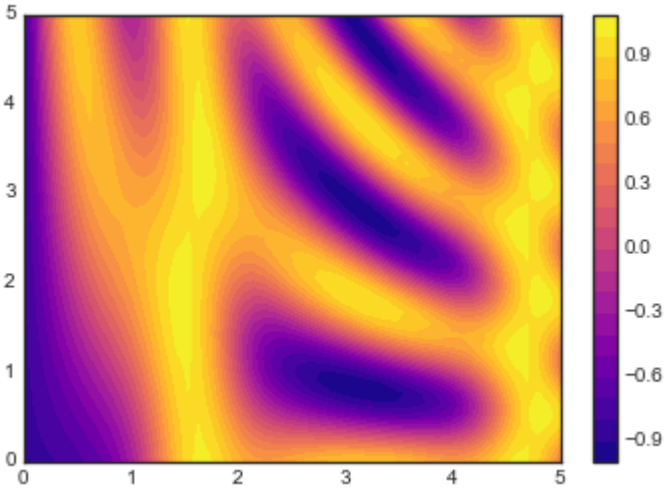
In [241]:
```
#RdGy: Red-Gray colormap

#==============================# CODE HERE #============================
==#

# show color scale

#==============================# CODE HERE #============================
==#
```



`plt.contourf`

The same as `plt.contour` but will filling the space between the lines with colors:

In [244]:
```
#==============================# CODE HERE #==============================#

#==============================# CODE HERE #==============================#
```



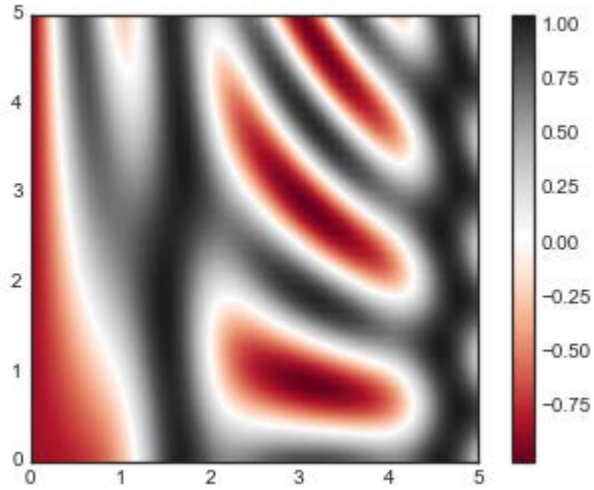
Further colorsmaps provided, can be selected benifiting IPython intellisence by exploring the properties in `plt.cm` module:

```
plt.cm.<TAB>
```

`plt.imshow`

The previously presented contour plots, the color gradient big steps rather than continuous flow. Of course the number of contours can be set to a high number, but the plot will be slower (Matplotlib must render a new polygon for each step in the level). A better way is provided using `plt.imshow()` function:

In [249]:
```
#============================# CODE HERE #============================
==#


#============================# CODE HERE #============================
==#


#============================# CODE HERE #============================
==#
```



`plt.imshow()` has some special properties:

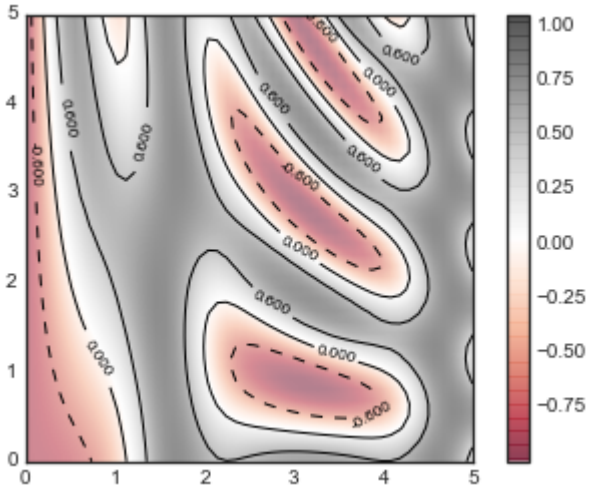
- `plt.imshow()` doesn't accept an *x* and *y* grid, so you must manually specify the *extent* [*xmin*, *xmax*, *ymin*, *ymax*] of the image on the plot.
- `plt.imshow()` by default follows the standard image array definition where the origin is in the upper left, not in the lower left as in most contour plots. This must be changed when showing gridded data.
- `plt.imshow()` will automatically adjust the axis aspect ratio to match the input data; this can be changed by setting, for example, `plt.axis(aspect='image')` to make *x* and *y* units match.

`plt.clabel`

Finally, it can sometimes be useful to combine contour plots and image plots. For example, here we'll use a partially transparent background image (with transparency set via the `alpha` parameter) and overplot contours with labels on the contours themselves (using the `plt.clabel()` function):

In [250]:
```
#============================# CODE HERE #============================
==#


#============================# CODE HERE #============================
==#


#============================# CODE HERE #============================
==#
```

```
#============================# CODE HERE #============================
==#
```



Histograms,binning and density plot[[1](#ref1)]

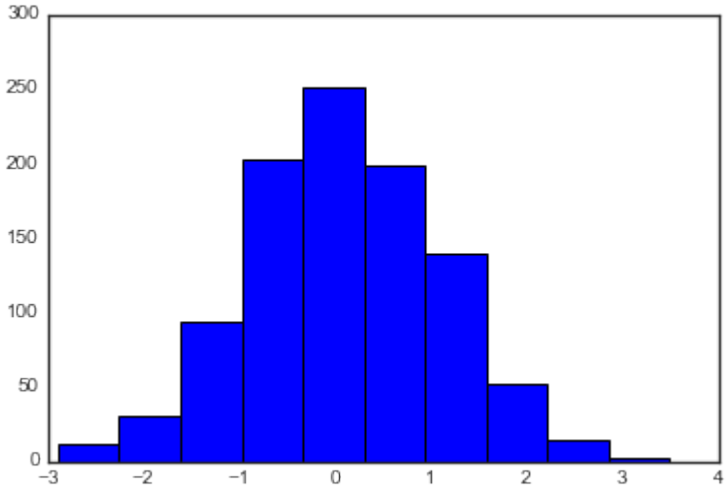
Histograms can be a great tool to understand how your dataset (or each feature in your dataset) is distributed.

1D Histogram

As Example:

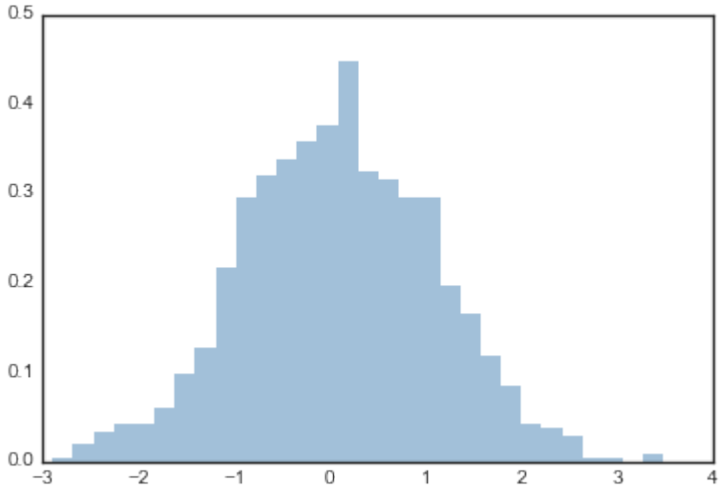
In [252]:
```
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')

import numpy as np
import pandas as pd

%matplotlib inline
```

In [254]:
```
# Create a histogram

#============================# CODE HERE #============================
==#


#============================# CODE HERE #============================
==#
```



As we will see in another section, controlling the bin number in histogram is an essential property to get the read distribution of the dataset:

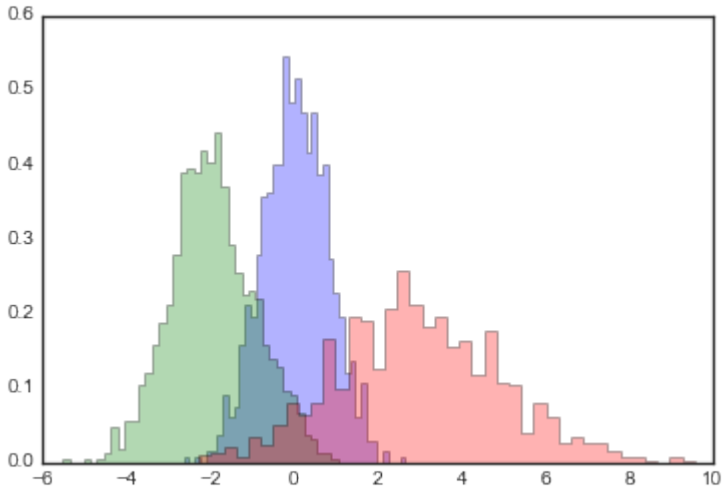
In [261]:
```
#============================# CODE HERE #============================
==#


#============================# CODE HERE #============================
==#
```



As any other plot in Matplotlib, it's possible to stack multiple histograms in one axes:

In [263]:
```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)


#============================# CODE HERE #============================
==#


#============================# CODE HERE #============================
==#


#============================# CODE HERE #============================
==#
```

```
#============================# CODE HERE #============================
==#
```



If you would like to simply compute the histogram (that is, count the number of points in a given bin) and not display it, the `np.histogram()` function is available:

In [264]:
```
#============================# CODE HERE #============================
==#

print(counts)
```

```
[ 43 297 450 192  18]
```

Exercise

From the *Diabetes* dataset, in histogram, plot the *age* and *sex* features.

Variate the number of bins, what do you notice ?

In [267]:
```
from sklearn.datasets import load_diabetes
data=load_diabetes() # the result of type sklearn.utils.Bunch
# Convert to pandas
data=pd.DataFrame(data=np.c_[data['data'], data['target']],
                  columns=data['feature_names']+['target'])

x1=data.age
x2=data.sex

data.head()
```

Out[267]:

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6	ta
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019908	-0.017646	15

1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068330	-0.092204	
2	0.085299	0.050680	0.044451	-0.005671	-0.045599	-0.034194	-0.032356	-0.002592	0.002864	-0.025930	14
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022692	-0.009362	2(
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031991	-0.046641	1:

In [ ]:
```
### YOUR CODE HERE
#
#
###
```

2D Histogram

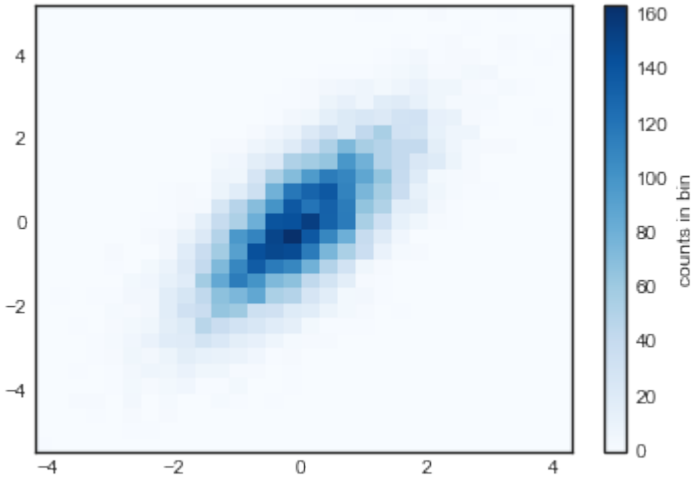
Histograms in one dimension divides the number-line into bins, also it is possible to create histograms in two-dimensions by dividing points among two-dimensional bins.

We'll start by defining some data—an `x` and `y` array drawn from a multivariate Gaussian distribution:

In [271]:
```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 10000).T
```

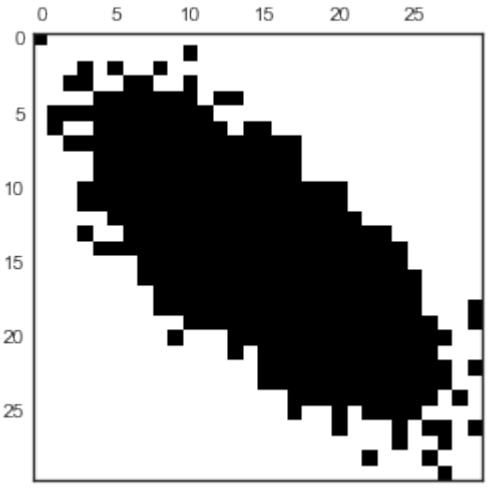
plt.hist2d

In [272]:
```
#============================# CODE HERE #============================
==#


#============================# CODE HERE #============================
==#

cb.set_label('counts in bin')
```



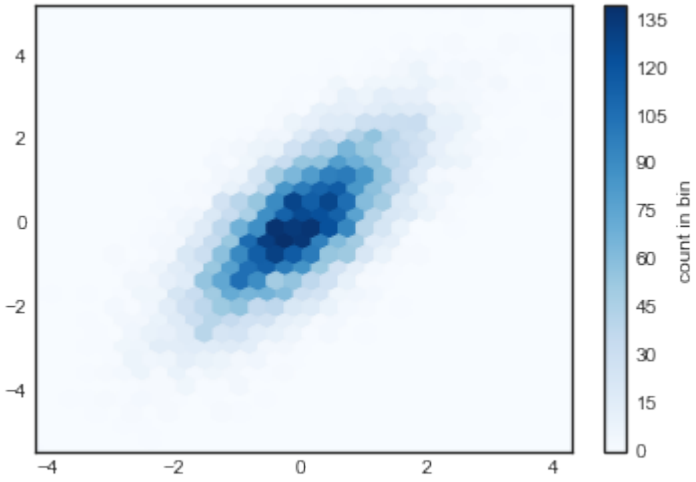
Just as `plt.hist` has a counterpart in `np.histogram`, `plt.hist2d` has a counterpart in `np.histogram2d`, which can be used as follows:

In [276]:
```
counts, xedges, yedges = np.histogram2d(x, y, bins=30)
plt.spy(counts);
# counts is a sparse matrix, the black cells are non zero values
```



`plt.hexbin`

In [277]:
```
#==============================# CODE HERE #==============================
==#

#==============================# CODE HERE #==============================
==#
```



Exercise

From the *Diabetes* dataset in the previous exercise, in two dimensional histogram, plot the relation of `age` (x-axis) with `sex` (y-axis).

In [ ]:
```
### YOUR CODE HERE
#
#
###
```

Subplots[[1](#ref1)]

Some times it's required to plot multiple plots side by side. For this goal Matplotlib provide three possibilities:

- `plt.axes` or `fig.add_axes`
- `plt.subplot`
- `plt.subplots`

In [2]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
plt.style.use('seaborn-white')
%matplotlib inline
```

plt.subplots?

`plt.axes` and `fig.add_axes`

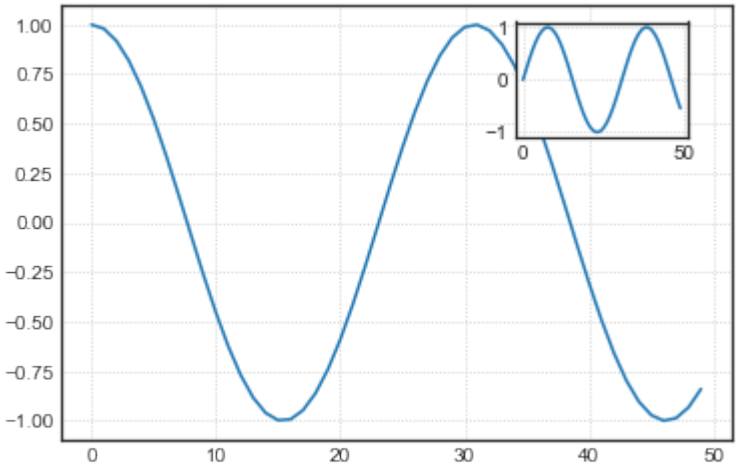
We can benefit the `plt.axes` to create multiple axes inside another.

When creating an axes using `plt.axes` without passing any argument, then one axis is created which fills up the entire figure.

`plt.axes()` accepts a list as input argument, this list contains `[left, buttom, width, height]` properties for the axis desired to be created. Each of these values ranges from 0 at the bottom left of the figure to 1 at the top right of the figure.

For example, we might create an inset axes at the top-right corner of another axes by setting the *x* and *y* position to 0.65 (that is, starting at 65% of the width and 65% of the height of the figure) and the *x* and *y* extents to 0.2 (that is, the size of the axes is 20% of the width and 20% of the height of the figure):

In [23]:
```
#============================# CODE HERE #============================
==#


#============================# CODE HERE #============================
==#

# Plot

#============================# CODE HERE #============================
==#

#============================# CODE HERE #============================
==#

#============================# CODE HERE #============================
==#
```



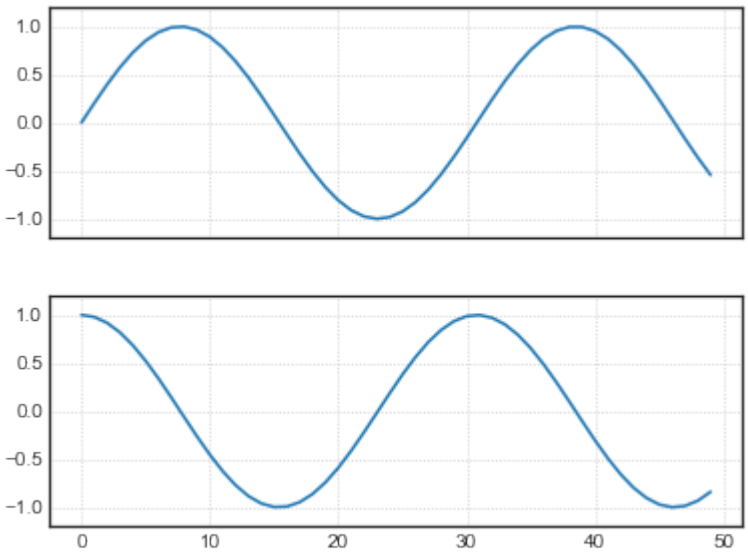
The equivalent of this command within the object-oriented interface is `fig.add_axes()`. Let's

use this to create two vertically stacked axes:

In [8]:
```
fig = plt.figure();
fig.add_axes?
```

```
<Figure size 432x288 with 0 Axes>
```

In [18]:
```
#============================== CODE HERE #============================
==#

# Add axis 1

#============================== CODE HERE #============================
==#


# Add axis 2

#============================== CODE HERE #============================
==#


# Plot
x = np.linspace(0, 10)

#============================== CODE HERE #============================
==#


#============================== CODE HERE #============================
==#
```



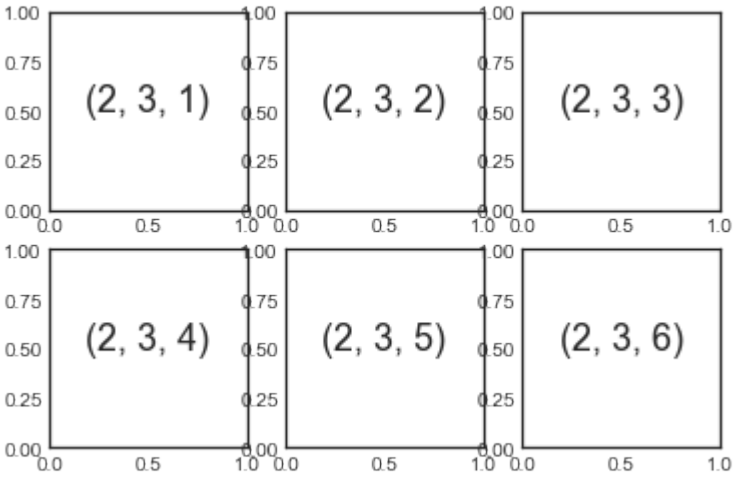
`plt.subplot`

Aligned columns or rows of subplots are a common-enough need that Matplotlib has several convenience routines that make them easy to create. The lowest level of these is `plt.subplot()`, which creates a single subplot within a grid.

As you can see, this command takes three integer arguments:

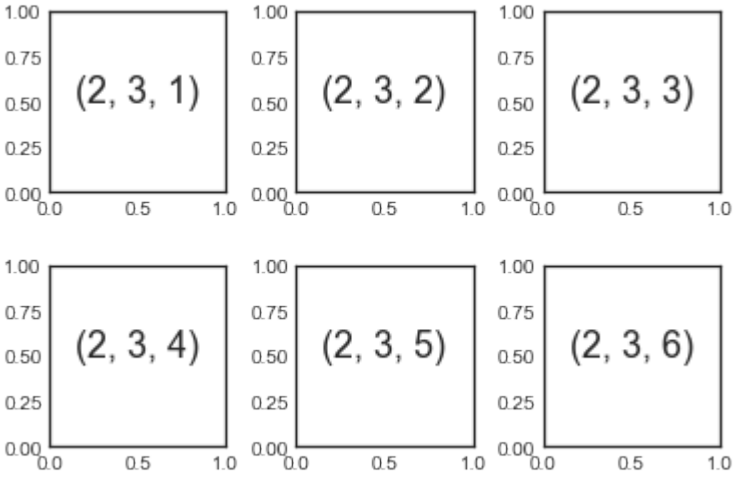
- the number of rows,
- the number of columns,
- and the index of the plot to be created in this scheme, which runs from the upper left to the bottom right.

In [28]:
```
fig = plt.figure()
for i in range(1, 7):
    #==============================# CODE HERE #=======================
======#
    #==============================# CODE HERE #=======================
======#
```



The command `plt.subplots_adjust` can be used to adjust the spacing between these plots. The following code uses the equivalent object-oriented command, `fig.add_subplot()`:

In [29]:
```
fig = plt.figure()

#==============================# CODE HERE #============================
==#

for i in range(1, 7):
    #==============================# CODE HERE #=======================
======#

    #==============================# CODE HERE #=======================
======#
```



`plt.subplots`

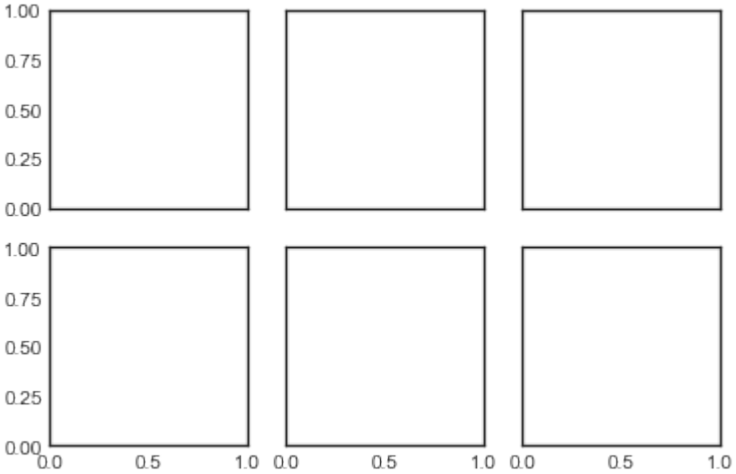
This is the easiest tool to use ! Rather than creating a single `subplot` each time you what to add a new plot, you can from the beginning define how many subplots you need using `plt.subplots`.

`plt.subplots` creates a full grid of subplots in a single line, returning them in a NumPy array.

The arguments are the number of rows and number of columns, along with optional keywords `sharex` and `sharey`, which allow you to specify the relationships between different axes.

As example: let's create a 2×3 grid of subplots, where all axes in the same row share their y-axis scale, and all axes in the same column share their x-axis scale:

In [30]:
```
#==============================# CODE HERE #============================
==#
```

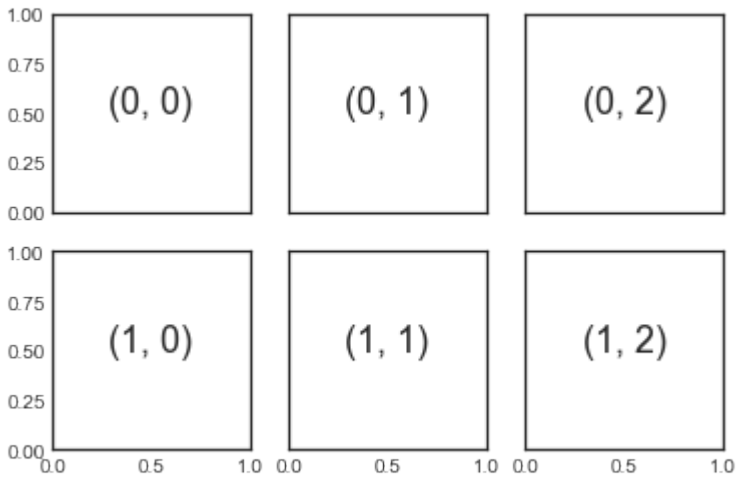


The index of each subplot, is represented below:

In [32]:
```
# axes are in a two-dimensional array, indexed by [row, col]
for i in range(2):
    for j in range(3):
```

```
        axs[i, j].text(0.5, 0.5, str((i, j)),
                       fontsize=18, ha='center')
fig
```

Out[32]:



Exercise

From the iris dataset imported below:

1. In (2×2) subplot, plot the: a. sepal_length b. sepal_width c. petal_length d. petal_width
2. In every subplot mentioned above, every *specie* from the `species` should be plotted in a separate color.
3. Add a legend to every subplot

In [37]:
```
import matplotlib.pyplot as plt
import seaborn as sns # to import a dataset
import pandas as pd
import numpy as np

plt.style.use("seaborn-white");
%matplotlib inline
```

In [38]:
```
dataset = sns.load_dataset("iris")
dataset.head()
```

Out[38]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [40]:
```
sepal_length = np.array(dataset.sepal_length)
sepal_width = np.array(dataset.sepal_width)
petal_length = np.array(dataset.sepal_length)
petal_width = np.array(dataset.sepal_width)
species = np.array(dataset.species)
```

In [ ]:
```
### YOUR CODE HERE
#
#
###
```

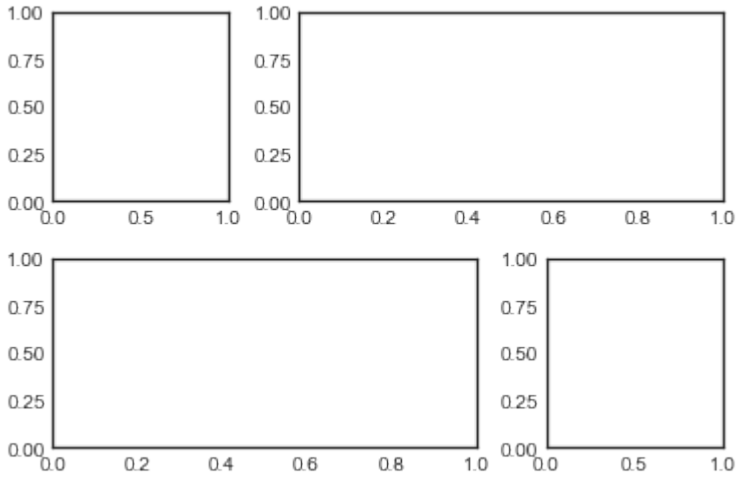
More complicated subplots

To go beyond a regular grid to subplots that span multiple rows and columns, `plt.GridSpec()` is the best tool. The `plt.GridSpec()` object does not create a plot by itself; it is simply a convenient interface that is recognized by the `plt.subplot()` command. For example, a gridspec for a grid of two rows and three columns with some specified width and height space looks like this:

In [34]:
```
grid = plt.GridSpec(2,3, wspace=0.4, hspace=0.3)#We want space for 2*3 subplots
```

From this we can specify subplot locations and extents using the familiar Python slicing syntax:

In [35]:
```
ax1=plt.subplot(grid[0, 0])
ax2=plt.subplot(grid[0, 1:]) # Take the area of subplots: (0,1) and (0, 2)
ax3=plt.subplot(grid[1, :2]) # Take the area of subplots: (1,0) and (1, 1)
ax4=plt.subplot(grid[1, 2]);
```



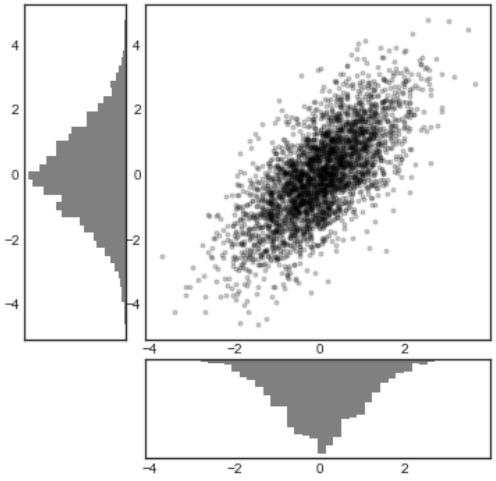
Exercise

In [79]:
```
# Create some normally distributed data
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 3000).T
```

From the previous generated random variables (x,y):

1. Scatter plot
2. Histogram for each

The structure of the subplots shall take the same form as shown in the figure below:



In [ ]:
```
### YOUR CODE HERE
#
#
###
```

References

[1] **BOOK** Python Data Science Handbook, J.VanderPlas

[2] **BOOK** Python for Data Analysis, L.Malid, S.Arora

[3] **BLOG** Visualization in Data Science(VDS at IEEE VIS 2019)

[4] **BLOG** The power of visualization

[5] **BLOG** Matplotlib : Style sheets reference

[6] **BLOG** Matplotlib : General Concepts

[7] **GITHUB** Anatomy of Matplotlib

[8] **BLOG** Scikit learn dataset