L06: Data Visualisation

Dr. Todd Jones

t.r.jones@reading.ac.uk

Department of Computer Science

Visualisation Libraries

- Implement **data discovery** and analysis by creating **graphical** representations of our data.
- **Matplotlib**: https://matplotlib.org/
  - Interfaces with or underlies many Python plotting utilities
  - Enables export to all of the common vector and raster graphics formats
    - PDF, SVG, JPG, PNG, BMP, GIF, PS, etc.
  - Use to fine-tune plot details
- **Pandas**: https://pandas.pydata.org/
  - Has own methods to create very simple graphs
  - Tries to automate some details (e.g., labelling)
- **Seaborn**: https://seaborn.pydata.org/
  - Builds on top of Matplotlib
  - Attempts to improve readability and aesthetics
- **Plotly**: https://plotly.com/python/ **(Further, independent resource)**
  - Interactive plots
  - Builds on top of the Plotly.js JavaScript library

Matplotlib

- `matplotlib` is a major plotting library for Python. It is included in Anaconda 3 and works nicely in Jupyter Notebooks.
  - **Low-level** plotting
  - Several other Python plotting libraries use `matplotlib` as a base.
  - Originally developed to mimic plotting in the **Matlab** language.
  - Plots can be highly customised, but the process is often rather **verbose** (lots of code).

- Within a Jupyter notebook its main functionality is typically imported as `plt`:

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

- To display images in-line (typically enabled by default), use a bit of IPython "magic":

```
%maplotlib inline
```

- To display images with an interactive panel, use:

```
%maplotlib notebook
```

- The **simplest method** for drawing `matplotlib` plots is to use various functions to generate plots, then modifying the result to adjust details.

In [ ]:
```
# Load pyplot/numpy
import matplotlib.pyplot as plt
import numpy as np

# Version check
from matplotlib import __version__ as mplversion
mplversion
```

Line plot: `plt.plot()`

- Line plots connect a collection of ordered coordinate points.

In [ ]:
```
# Very simple line plot

##  NOTE: The default x-axis values are a count of array elements.

data = np.arange(100)
plt.plot(data)
```

Figures and subplots: `plt.figure()` and `plt.subplots()`

- Plots in `matplotlib` reside in a `Figure` **object**.
- On its own, `plt.figure()` will be blank.
- One or more `subplots` are needed to fill the space.
- `subplots` are `Axes` **objects** (recently these were more specifically `AxesSubplot` objects).
- In most spaces in this notebook, the `Figure` and `Axes` objects are formed in the background automatically and are no longer accessible after cell execution.

> **Tip:** In Jupyter, all plot commands for a single plot need to reside within a **single cell** because plots are reset after execution.

...unless you are storing and modifying a plot object as a variable for later use...

In [ ]:
```
##  Let's also play with the notebook's display properties.
'''
Allows multiple statements/outputs to be displayed at once
    Options:
```

```
        'all', 'last', 'last_expr', 'none', 'last_expr_or_assign'

    Default:
        'last_expr'
'''

get_ipython().ast_node_interactivity = 'all';

##  NOTE: If you terminate a cell with ';', then only the last statement
#         will display output.
```

In [ ]:
```
# Make a 2x2 subplot figure with only 3 subplots.

fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)  # Axes object, 1-based (nrow, ncols, position)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)

# Create data
data = np.random.randn(100).cumsum()

# Plot data as a green dotted line (shorthand style option)
plt.plot(data,
         'g:')  # Automatically placed in most recently created subplot

# Add to other subplots (any order is ok)
(vals, bins, _) = ax1.hist(np.random.randn(100), bins=20,
                           color='k', alpha=0.3)

ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.randn(30))
ax2.set_xlabel('time')

###  NOTE: plot commands return objects!
```

In [ ]:
```
# Inspect object
ax1
```

In [ ]:
```
# Quicker way to work with subplots:

nrows = 2
ncols = 3
fig, axes = plt.subplots(nrows, ncols, sharex=True, sharey=True)

# Configure a list of colours to match the shape of `axes`
colors = [['r', 'b', 'y'],
          ['g', 'k', 'c']]

# Explicitly loop over both dimensions
for i in range(nrows):
    for j in range(ncols):
        axes[i, j].hist(np.random.randn(500),
                        bins=50,
                        color=colors[i][j],
                        alpha=0.5)

# Remove space between subplots
plt.subplots_adjust(wspace=0, hspace=0)
```

In [ ]:
```
# Inspect axes
axes
```

Plot feature options

- There are a multitude of configurable options.
- Please, explore the many arguments available.
- For example, you can change the sizing of the output figures for **all plots** in the notebook by modifying the `rcParams`:

```
import matplotlib
    matplotlib.rcParams['figure.figsize'] = (4.5, 3)  ## (width, height) in
inches
```

- There are **TONS** of default styling parameters:
  https://matplotlib.org/stable/tutorials/introductory/customizing.html

---

You are already familiar with some of its functionality from the discussion of `pandas` plots!

- The **simplest method** for drawing `matplotlib` plots is to use its various functions to generate plots, then modifying the result to adjust details.

In [ ]:
```
## Before we proceed, let's disable the notebook multiple display
## by reverting back to the default settings

get_ipython().ast_node_interactivity = 'last_expr'
```

In [ ]:
```
# Plot features examples

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

plt.plot(data,
         color = 'c',
         linestyle = ':',
         marker = 'o',
         markerfacecolor = 'm',
         label = r'line label, $\theta = \frac{5 - \frac{1}{x}}{4}$')

plt.ylabel('y-axis')
plt.title('title here')
plt.suptitle('Higher title for all subplots')

ax = plt.gca()  # get current axis
ax.set_xticks([10,20,30,40])
ax.set_xticklabels(["one", "two", "three", "four"],
                   rotation=40)
ax.set_xlabel('x-axis')
plt.legend(loc='upper left')
plt.savefig('example_plot.png', dpi=300)  ## Format inferred from extension
```

In [ ]:
```
## Repeating the plt.plot() before the figure is closed
#  will add lines to the current `Axes` instance.

plt.plot(data)
plt.plot(data + 5)
plt.plot(data - 5, marker='x');
```

Scatter plots: `plt.scatter()`

- To visualise how two variables are **related**.

- Helps us to quickly identify associations between two variables.
- Colour and size can also be specified, allowing us to represent **3 or 4 variables** at once.

In [ ]:
```
import pandas as pd

# Loading Data Set
diamonds = pd.read_csv("data/diamonds.csv")
```

In [ ]:
```
# Creating the plot
plt.scatter(diamonds['carat'], diamonds['price'])
plt.xlabel('Carat')
plt.ylabel('Price');
```

In [ ]:
```
# Scatter plot with varying sizes and colours.

# Subset data
subset = diamonds.iloc[800:900].copy()

# Configure sizes using min/max scaling (and cubing)
size = 2000 * ((subset.table - subset.table.min()) /
               (subset.table.max() - subset.table.min()))**3

# Configure color values for the color categories
ucol = subset.color.unique()
colmap = dict(zip(ucol, np.linspace(1, 254, len(ucol))))
subset['plot_color'] = subset.color.map(colmap)

# Plot
plt.scatter(subset['carat'],
            subset['price'],
            s=size,
            c=subset.plot_color,
            alpha=0.5)
plt.xlabel('Carat')
plt.ylabel('Price');
```

In [ ]:
```
# Inspect
subset['plot_color']
```

In [ ]:
```
# Examine a small (5%) random sample of the data
diamonds_tmp = diamonds.sample(frac=0.05)

plt.scatter(diamonds_tmp['carat'],
            diamonds_tmp['price'],
            c=diamonds_tmp['table'])

cbar = plt.colorbar()  ## Add a colorbar
plt.xlabel("Carat")
plt.ylabel("Price")
cbar.ax.set_ylabel('Table');  ## Label the colorbar
```

Bar plots: `plt.bar()`

- Good for displaying **categorical** data, one bar per category.

In [ ]:
```
# Bar Plot: specify x vs. y data

# Create a count variable
```

```
cc = diamonds['cut'].value_counts()

plt.bar(cc.index, cc)
plt.xlabel('Cut')
plt.ylabel('Count');
```

Named colours

- Colour specification in `matplotlib` can be done in a variety of ways:
  - Shorthand: `'r','g','y'`
  - Names: `'blue', 'tab:blue', 'peachpuff'`
  - RGB(A)
  - CMYK
  - HSV (Hue, Saturation, Value)
  - HTML codes: `'#FF0000'`

Explore *named colors*:

```
import matplotlib.colors as mcolors
```
https://matplotlib.org/stable/gallery/color/named_colors.html

In [ ]:
```
import matplotlib.colors as mcolors
mcolors.TABLEAU_COLORS
```

Let's try another bar plot example, this time for the mean price by cut.

- We will implement the plot in a different way, manually specifying where to place the bars and how to label and position them.

In [ ]:
```
# Group by cut
diamonds_bycut = diamonds.groupby('cut')

# Calculate average price and reorder the result
bc = diamonds_bycut[['price']].mean().loc[cc.index]

plt.bar(np.arange(len(bc)), bc['price'],
        color=[x for x in mcolors.TABLEAU_COLORS],
        width=0.5)
ax = plt.gca()
ax.set_xticks(np.arange(len(bc)))  ## Tick numerical position in x [0-4]
ax.set_xticklabels(bc.index, rotation=40)
ax.set_xlabel('Cut')
ax.set_ylabel('Average price');
```

A different version of the previous figure...

In [ ]:
```
# Group by cut
diamonds_bycut = diamonds.groupby('cut')

# Calculate average price and reorder the result
bc = diamonds_bycut[['price']].mean().loc[cc.index]

plt.bar(np.arange(len(bc)), bc['price'],
        color=[x for x in mcolors.TABLEAU_COLORS],
```

```
        width=0.5)
ax = plt.gca()
ax.set_xticks(np.arange(len(bc)))  ## Tick numerical position in x [0-4]
ax.set_xticklabels(bc.index, rotation=40)
ax.set_xlabel('Cut')
ax.set_ylabel('Average price')
ax.set_ylim((3250, max(bc.price)*1.05));
```

In [ ]:
```
# Inspect the values
bc.loc[cc.index]
```

Pie charts: `plt.pie()`

- **Pie charts** (and the related donut plot) are common alternative to bar plots.
- However, many people discourage their use.
  - Gaining insights based on the comparison of angles and areas is challenging for many.
  - With a large number of categories, it can be difficult to differentiate between sectors and compare similar ones.
- That said, here is how you can make one using `plt.pie()`:

In [ ]:
```
# Pie Chart

# Create an array for a single "exploded" wedge
ex = np.zeros(len(cc))
ex[-1] = 0.4

# Pie returns a long bit of plotting information.  Ignore its result.
_ = plt.pie(cc.values, labels = cc.index,
            explode = ex,            ## explode
            autopct = '%1.1f%%',     ## display percentage
            shadow = True)           ## enable shadow effect
```

Histograms: `plt.hist()`

- **Histograms** divide the values into multiple **bins**, each corresponding to a different range of values.
- Based on `numpy.histogram()` to bin and count.
- By default, `plt.hist()` will set the range from the minimum to the maximum of the data.
  - Configurable via `range` parameter (tuple)

Bins

- The `bins` parameter can set the number (int, default of 10) or edges (sequence) of the bins.
  - Also allows some specific methods as strings (`'auto', 'sturges', ...`)
  - https://numpy.org/doc/stable/reference/generated/numpy.histogram_bin_edges.html#numpy.histog
- Plotting too few bins might cause you to miss some details.
- Helpfully returns the values and the bin edges.

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.hist.html

In [ ]:
```
# Most basic example.  All defaults.

##  NOTE: Difference in size of value and bins arrays.
#         No labels.

values, bins, _ = plt.hist(diamonds['price'])
print(f'values = {values}, size = {len(values)}')
print(f'bins = {bins}, size = {len(bins)}')
```

In [ ]:
```
## Enable 'auto' binning

values, bins, _ = plt.hist(diamonds['price'], bins = 'auto')
print(f'values = {values}, size = {len(values)}')
print(f'bins = {bins}, size = {len(bins)}')
```

Box and whisker plots: `plt.boxplots`

- **Box and whisker** plots allow us to see the **distribution** of the data and compare distributions in different categories.
  - **Box:** denote the 1st and 3rd **quartile**, with a line for the **median**
  - **Whiskers:** show the largest value occurring within 1.5 x the **interquartile range** (1st to 3rd quartiles)
  - **Outliers:** shown as circles for any data points **outside** the whiskers.

In [ ]:
```
# A time series of the daily mean temperature of the Fisher river over 4 year.
# Data by K. W. Hipel and A. I. McLeod (1994), sourced from Hyndman,
# R.J. Time Series Data Library, https://github.com/FinYang/tsdl.

river = pd.read_csv('data/river.csv')
river

## Make a boxplot with labelled elements

# Use river temperature dataset
vname = 'Mean daily temperature'
data = list(river[vname])

# Modify to ensure outliers are present
data.extend([55,60,-55,-60])

# Configure plot
plt.figure(figsize=(2.5,6))
stats = plt.boxplot(data, labels=['river'])
plt.title('Labelled Boxplot')
plt.ylabel(vname)
plt.xlim(0.9,1.3)
offset = 1.1
plt.annotate('Outliers', xy=(offset,55))
plt.annotate('Outliers', xy=(offset,-60))
for item in stats['whiskers']:
    plt.annotate('Whisker', xy=(offset, item.get_ydata()[1]), va='center')
plt.annotate('1st Quartile', xy=(offset, stats['boxes'][0].get_ydata()[-1]), va='center')
plt.annotate('3rd Quartile', xy=(offset, stats['boxes'][0].get_ydata()[-2]), va='center')
plt.annotate('Median', xy=(offset, stats['medians'][0].get_ydata()[1]), va='center');
```

In [ ]:
```
# Looking at the distribution of depth values for each diamond cut:

# Empty list for depth data
cuts = []

# Custom order of cuts
cutlabel = ['Fair', 'Good', 'Very Good', 'Premium', 'Ideal']

# Add depth data to the list
for c in cutlabel:
    cuts.append(diamonds.loc[diamonds.cut == c, 'depth'])

plt.boxplot(cuts, labels=cutlabel)
plt.xlabel('Cut')
plt.ylabel('Depth')
```

In [ ]:
```
## Closer look at the data for Fair and Premium as histograms

# Prepare subplot figure (rows, columns, figsize=(width, height))
fig, ax = plt.subplots(1, 2, figsize = (12, 4))

# Plot histograms
_ = ax[0].hist(cuts[0], bins='auto', label=cutlabel[0])
_ = ax[0].hist(cuts[3], bins='auto', label=cutlabel[3], alpha=0.7)
ax[0].set_title('linear')
ax[0].set_xlabel('depth')
ax[0].set_ylabel('count')
ax[0].legend()

# Plot histograms on log scale
_ = ax[1].hist(cuts[0], bins='auto')
_ = ax[1].hist(cuts[3], bins='auto', alpha=0.7)
ax[1].set_title('logarithmic')
ax[1].set_xlabel('depth')
ax[1].set_ylabel('count')
plt.yscale('log')

fig.tight_layout()
```

Density Estimates

- Histograms show a *coarse* estimate of the **density** of a variable.
- The **density** is the *underlying* probability function of a variable.
  - That is, estimate the probability density function (PDF) if a random variable.
- Various approaches can be used to **estimate** a smoothed density for a continuous variable.
- **Kernel density estimate** (KDE)
  - Places a kernel function centered at each data point to generate a **smooth** density estimate.
  - Often, K is a **Gaussian** density with a variance (or bandwidth) selected automatically from the data.

$$\hat{g}(x) = \sum_{i=1}^{N} K(x - x_i)$$

- Let's look at the diamond price information:

In [ ]:
```
## Use pandas plot.hist() to quickly recall the data

diamonds['price'].plot.hist(bins=100)
```

In [ ]:
```
# Import a helper package
import scipy.stats

# Create kernel function
# https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.gaussian_kde.html
kernel = scipy.stats.gaussian_kde(diamonds['price'])

# Generate x-values to use in kernel function (much larger range than needed)
xs = np.linspace(-1000,max(diamonds['price']),1000)

# Apply function and plot
plt.plot(xs,kernel(xs), label='KDE')

_ = plt.hist(diamonds['price'], bins=100, density=True, label='hist')

plt.legend();
```

NOTE:

- The density estimate is non-zero below a price of zero.
- The density estimate is non-zero outside the range of the data.
- Above method works best for a unimodal distribution. Multi-modal distributions tend to be overly smoothed.

Simple Plots with Pandas

- `pandas` has some built-in functionality for quickly generating **simple plots** from `DataFrame`s and `Series` objects.
- This is built on top of `matplotlib`.

Line plot: `.plot()`

- 1-dimensional data
- $y = f(x)$
- x vs. y
- Groups of 1-dimensional data

https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.DataFrame.plot.html

In [ ]:
```
# Recall: simple cross tabulation
cross = pd.crosstab(diamonds['cut'], diamonds['clarity'])
cross
```

In [ ]:
```
# plot the number of entries for Fair-cut diamonds of all clarity

cross.loc['Fair'].plot()
```

In [ ]:
```
# Plot the count of IF-clarity diamonds versus the count of VVS1-clarity diamonds for
# corresponding cuts

cross.plot.line(x='IF', y='VVS1')

##  WARNING: this is a silly plot!
```

In [ ]:
```
# Without specifying individual colums
cross.plot()
plt.ylabel('label')
```

Histograms: `plot.hist()`

- **Showing the frequency of values in a given column in the `DataFrame`**

https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.DataFrame.hist.html

In [ ]:
```
# Quickly see histograms for all of the columns

##  NOTE: None of the categorical dimensions are shown.

diamonds.hist();
```

In [ ]:
```
# Call on matplotlib to fine-tune the final display subfigure spacing.

#### NOTE:  We no longer benefit from a trailing ';' because it is
#           not the last command in the cell.

import matplotlib.pyplot as plt

diamonds.hist()

plt.tight_layout()
```

In [ ]:
```
# Delve deeper to quickly visualise histograms for depth,
# grouped by the cut category.

##  NOTE:  Custom configuration -
#            - Use consistent, pre-defined bins that span data range
#            - `column` selects data of interest
#            - `by` groups the data by a category
#            - `bins` specifies bin edge values
#            - `sharex` and `sharey` force the same range in each panel

# Use numpy help
import numpy as np

# Set up 100 linearly spaced bin edges for the data range
#   - to be used in all cases
bins = np.linspace(diamonds['depth'].min(), diamonds['depth'].max(), 100)

# Display customised histograms
diamonds.hist(column='depth', by='cut', bins=bins, sharex=True, sharey=True)
plt.tight_layout()
```

Scatter plots: `plot.scatter()`

- Visualise how two variables are related to one another.
  - Here, we will specify which columns of the `DataFrame` we would like to compare on the `x` and `y` axes.

https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.DataFrame.plot.scatter.html

In [ ]:
```
# How are carat and price related - in light of the x-dimension length?

##  NOTE:
#       - `c` specifies a kind of colour value
#       - `cmap` is a pyplot argument for a colormap
#       - `sharex` - we have seen before, but in this case,
#                    it fixes a glitch in the xlabel display

diamonds.plot.scatter(x='carat',
                      y='price',
                      c='x',
                      cmap='gist_ncar_r',
                      sharex=False)
plt.title('Scatterplot'); # mpl customisation
```

Scatter plots: `.plot.bar()` and `.plot.barh()`

- Bar charts are usually used to compare **numerical data** for a relatively small number of categories.
  - This works well with **grouped** data.

https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.DataFrame.plot.bar.html

In [ ]:
```
# with horizontal bars, outlined in red:

diamonds_bycolor = diamonds.groupby('color')
diamonds_bycolor[['price']].mean().plot.barh(edgecolor='r');
```

In [ ]:
```
# Further group the data by color and cut

# Aggregate by mean price
# unstack the resulting multiIndex data
multi = diamonds.groupby(['color', 'cut']).agg({'price': ['mean']}).unstack()


# Use `droplevel()` twice to remove 'price' and 'mean' labels
multi.columns = multi.columns.droplevel().droplevel()

multi.plot.bar()

# A different way to label the axis
plt.gca().set_ylabel('Mean price')
```

Parallel Plots: `pd.plotting.parallel_coordinates()`

- One way of visualising data with more than **two dimensions** is to use a **parallel plot**.
- **Great to differentiate clustered data.**

- Each variable is shown as an axis, side by side.
- Data points are drawn as lines across the axes.
- `parallel_coordinates` requires a `DataFrame` and a column of categorical data to use as category names.
- **Best when:** all data features are on approximately the same scale.

https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.plotting.parallel_coordinates.html

In [ ]:
```
#  Import functionality
from pandas.plotting import parallel_coordinates

# Read iris plant data
iris = pd.read_csv("data/iris.csv")

iris.head()

# Select to break down data by Species
parallel_coordinates(iris,'Species');

##  NOTE: This works best when the variables are of a similar scale.
#         Allows us to look for differences in similar catergories of objects.
```

Seaborn

- `seaborn` is a library built on `matplotlib` that takes away some of the extra work required to produce nicely labelled plots in Python.
- `seaborn` can act on `pandas` `DataFrames` and a specification of which columns to use for each aspect of the plot.
  - It also works with `numpy` arrays.
- It is particularly good for **multivariate** data visualisation.
- Typically imported as `sns`:

```
import seaborn as sns
```

- Employs "*facet grids*" to incorporate categorical variables with additional grouping dimensions easily.
  - Handles **aggregation** and **summarisation** automatically.

In [ ]:
```
import seaborn as sns
```

Line plots: `sns.relplot()`

- Creates **relational plots** to show the relationship between 2 datasets.
- Can specify `kind='line'` or `kind='scatter'`
- Easily specify colors and sizes and line styles to differentiate data further.

In [ ]:
```
## Load example flight data from seaborn

flights = sns.load_dataset("flights")
flights.head()
```

In [ ]:
```
sns.relplot(data=flights, x="year", y="passengers", hue="month",
            kind="line")
```

In [ ]:
```
# Enable the default theme
sns.set_theme()
sns.relplot(data=flights, x="year", y="passengers", hue="month", kind="line")
```

Bar plot: `sns.countplot()` and `sns.catplot()`

- `countplot` shows the counts of observations in each categorical bin using bars.
  - A count plot can be thought of as a **histogram** across a categorical, instead of quantitative, variable.
- `catplot` shows the relationship between a **numerical and one or more categorical variables** using one of several visual representations

In [ ]:
```
sns.countplot(x="cut", data=diamonds)

## Look how it labels it for us!

## See how there's evidence of matplotlib (Axes object).
```

In [ ]:
```
# Grouping variables
sns.countplot(x="cut", hue = 'color', data=diamonds)

## So much simpler than before!
```

In [ ]:
```
# Plot against a numerical value in specified order
sns.catplot(x='cut',
            y='price',
            hue='color',
            data=diamonds,
            order=cutlabel,
            kind="bar")

##  NOTE: it has automatically calculated the mean and added error bars
```

In [ ]:
```
# Enhanced box plot with a larger number of quantiles
sns.catplot(x='cut', y='price', data=diamonds, order=cutlabel, kind="boxen")
```

In [ ]:
```
# Violin plots include both boxplot and KDE information.
# Can be further subdivided to compare a binary category.

sns.catplot(x='cut', y='price', data=diamonds, order=cutlabel, kind="violin")
```

In [ ]:
```
# Simpler implementation
sns.catplot(x='cut', y='price', data=diamonds, order=cutlabel, kind="point")
```

Joint Plots: `sns.jointplot()`

- Scatterplot with marginal distributions (and linear regression lines)

In [ ]:
```
# Iris
sns.jointplot(x='Sepal.Length', y='Petal.Length', data=iris, kind='scatter',hue='Species')
```

In [ ]:
```
# Add kde
sns.jointplot(x='Sepal.Length',
              y='Petal.Length',
              data=iris,
              hue='Species',
              kind='kde')
```

Pair plots: `sns.pairplot()`

- To visualise relationships between multiple variables we can use pair plots:

In [ ]:
```
# Iris Dataset
sns.pairplot(iris)
```

In [ ]:
```
# Adding a hue level gives KDE along the diagonal,
sns.pairplot(iris, hue="Species", kind='kde')
```

Seaborn: See also...

- `sns.histplot()`
- `sns.boxplot()`
- `sns.displot()`
- `sns.kdeplot()`
- `sns.scatterplot()`

In [ ]:
```
# Simple 4-variable scatter plot
plt.figure(figsize=(8,8))
sns.scatterplot(x='carat', y='price', hue='clarity',
                data=diamonds.sample(frac=0.1), alpha=0.5, size='table')
```

Time Series

- When working with time series data, it is useful to have a data type for storing **dates** and **times**.
  - Data may have **fixed** or **irregular** record time frequencies.
    - **Timestamps**: specific instants in time
    - **Intervals**: indicated by a start and end timestamp
    - Fixed **Periods**: a specific month or year, *a special instance of an interval*
    - **Experiment or elapsed time**: a series of timestamps denoting a measure of time relative to a reference time

In [ ]:
```
# datetime stores both the date and time, down to the microsecond.

from datetime import datetime

now = datetime.now()
```

```
print(type(now))

print(now)

print(now.year, now.month, now.day, now.minute, now.second, now.microsecond, now.weekday(), now

print(f'isocalendar: {now.isocalendar()}')

print(f'UTC time: {now.utcnow()}')  # (Year, week number, day of week number)

print(f'Custom format: {now.strftime("%y-%m-%d_%H:%M:%S")}')
```

In [ ]:
```
# We can find the difference between two datetime objects

now2 = datetime.now()

delta = now2 - now

print(type(delta))
print(delta)
```

In [ ]:
```
# We can compare times that we define, too:

delta = datetime.now() - datetime(1970, 1 , 1)

print(delta)
print(f'Converted to seconds: {delta.total_seconds()}')
```

In [ ]:
```
# Inspect river data (read from file above)
river
```

In [ ]:
```
# Quick plot
river.info()
river.plot()   # Makes a simple line plot (only 1 column).
```

In [ ]:
```
# Change the `index` column

river = pd.read_csv('data/river.csv', index_col=0)
river
```

In [ ]:
```
# Retry plot
river.info()
river.plot()   # Makes a simple line plot (only 1 column).
```

In [ ]:
```
# Use `parse_dates=True` to try parsing the index values

##  NOTE:  index type will now become DatetimeIndex

river = pd.read_csv('data/river.csv', index_col=0, parse_dates=True)
river.info()
```

In [ ]:
```
# Plot labels are much improved!
river.plot();

##  But we could still benefit from some units!
```

In [ ]:
```
# Adding a label
river.plot()
plt.ylabel('[$\degree C$]')
```

`.resample()` - Resample time-series data (new frequency)

In [ ]:
```
Wmax = river.resample('W').max()
Wmax.columns=['Weekly Maximum Temperature']
ax = Wmax.plot()

Wmin = river.resample('W').min()
Wmin.columns=['Weekly Mimimum Temperature']
Wmin.plot(ax=ax)

plt.ylabel('[$\degree C$]')
plt.title('Weekly Temperature Range');
```

`.rolling()` - rolling window calculations (smoothing)

In [ ]:
```
window = 30
river.rolling(window).mean().plot()
plt.ylabel('[$\degree C$]')
plt.title('30-Day Rolling Average');
```

In [ ]:
```
# Connection to f-strings

day = datetime(1936, 8, 1)
print(f"{day} was a {day:%A}")
```

The `datetime` object's `strftime()` method accepts format codes that can also be used in f-strings.

https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes

---

Plotly

- Python graphing library for **interactive** plotting.
    - *Figures dynamically resize when you change the size of your browser window.*
- Builds on top of the Plotly.js JavaScript library.
- Typically imported as `px`:

```
import plotly.express as px
```

- Figures are represented in Python as `dicts` or as instances of the `plotly.graph_objects.Figure` class:

```
import plotly.graph_objects as go
```

In [ ]:
```
# plotly imports
import plotly.express as px
import plotly.graph_objects as go
```

In [ ]:
```
# Example
label = ["A", "B", "C", "D", "E", "F"]
source = [0, 3, 1, 1, 0, 4, 5]  ## correspond to label index
target = [2, 3, 2, 5, 4, 1, 2]  ## correspond to label index
value = [10, 4, 6, 7, 2, 8, 1]  ## correspond to source/target pair
```

```
fig = go.Figure(data=[
    go.Sankey(

        # Define nodes
        node=dict(label=label, ),

        # Add links
        link=dict(
            source=source,
            target=target,
            value=value,
        ))
])

fig.update_layout(title_text="Second Sankey Graph", font_size=15)
fig.show()
```

Additional Resources

- **Pandas plotting**
  - Chart visualisation: https://pandas.pydata.org/docs/user_guide/visualization.html
- **Matplotlib**
  - Pyplot tutorial: https://matplotlib.org/stable/tutorials/introductory/pyplot.html
  - Gallery of examples: https://matplotlib.org/stable/gallery/index.html
  - More tutorials: https://matplotlib.org/stable/tutorials/index.html
  - **Colormaps**: https://matplotlib.org/stable/gallery/color/colormap_reference.html
  - Shorthand styles for **format strings**: https://matplotlib.org/3.7.3/api/_as_gen/matplotlib.pyplot.plot.html#:~:text=Notes-,Format%20String%20format%20string
- **Seaborn**
  - Example gallery: http://seaborn.pydata.org/examples/index.html
  - Tutorial: https://seaborn.pydata.org/tutorial.html
  - Many examples given on documentation page for each kind of plot.
- **Plotly**, **(Further, independent resource)**
  - https://plotly.com/python/basic-charts/
  - https://plotly.com/python/statistical-charts/
- **Time Processing:**
  - `datetime` : https://docs.python.org/3.11/library/datetime.html
    - for manipulating dates and times
  - `time` : https://docs.python.org/3.11/library/time.html
    - various time-related functions
  - `calendar` : https://docs.python.org/3.11/library/calendar.html
    - General calendar-related functions

Further Reading

- McKinney, W. (2022). *Python for Data Analysis : Data Wrangling with Pandas, NumPy, and IPython*
  - **Chapter 9:** Plotting and Visualisation

    - **Chapter 11:** Time Series
- Mukhiya S.K. and Ahmed, U. (2020). *Hands-On Exploratory Data Analysis with Python*
    - **Chapter 2:** Visual Aids for EDA
    - **Chapter 8:** Time Series Analysis
- VanderPlas, J. (2017). *Python Data Science Handbook*
    - **Part 4:** Visualisation with Matplotlib

In This Week's Practical

3 Datasets:

- `faithful.csv` : geyser eruption timings
    - Explore histograms and joint plots.
- `marine-historical-2021-en.csv` : Hong Kong Marine Water Quality indicators from several stations in 2021
    - **Slightly complex** data cleaning
    - Explore many visualisation types alongside new `pandas` manipulations and figure customisation tools.
- `longley.csv` : U.S. macroeconomic variables 1947-1962
    - Explore correlation matrices and their visualisation with the `seaborn` heatmap.

Next Week

Machine Learning with `sklearn`