15-110 CMU-Q: A reference card for Python - Part III

April 30, 2019*

Contents

Matplotlib and Pyplot

Module `matplotlib` provides an extensive set of tools for the graphical display of data. We have focused on the use of sub-module `pyplot`, which is quite handy to use and reflects the approach used in the popular *Matlab* software for data handling and display. `pyplot` can be imported using the following statement:

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

In matplotlib, once a new figure is created, multiple graphical elements can be added and/or specified (to have a behavior different from the default). This means that a graphical figure can include, for instance, multiple datasets displayed according to different formats (e.g., a scatter plot, a histogram, a plot, a box plot, etc.).

In the following a list of useful methods from pyplot and matplotlib are described.

- `plt.figure()` creates a new figure; it is not strictly necessary (but highly recommended) to invoke it, since a new figure will be created implicitly when invoking any plotting method.

- `plt.figure(figsize(xsize, ysize))` allows to specify the dimensions of the figure along x and y dimensions.

- ```
  plt.figure()
  fig, subplots = plt.subplots(nrows, ncols,
                               figsize=(xsize, ysize))
  ```

  allows to place multiple plots (*subplots*) in the same figure; the `subplots(nrows, ncols, figsize)` method defines a grid (a matrix) of `nrows` $\times$ `ncols` locations where the different plots will be placed. Overall, the figure will have the size defined by `figsize`.

  Graphical elements can be added to subplot (i, j) by using the notation: `subplots[1, 0].hist(my_data)` that for instance adds a histogram plot to the subplot in row 1 and column 0.

  If only one row of subplots is used, then the notation `subplots[i]` should be used.

- `plt.title(title)` uses the string `title` as a title for the plot.

- `plt.xlabel(label)`, `plt.ylabel(label)` use the string `label` as the label for the selected x, y axis of the plot.

- `plt.xlim(a, b)`, `plt.ylim(a, b)` set the limits for the values displayed on the x and y axis.

- `plt.xticks(ticks_pos, <my_labels>)`, `plt.yticks(ticks_pos, <my_labels>)` control the appearance of the ticks on the x and y axis, where `ticks_pos` is a sequence defining where the tick marks should be placed, the optional `my_labels` sets explicit labels to be placed at the given `ticks_pos`.

- `plt.plot(y_seq)` plots the data in the sequence `y_seq`; since x coordinates are unspecified, the point data in `y_seq` are plotted as in the `plot()` invoked as `plt.plot(x_seq, y_seq)`, where `x_seq = range(0,len(y_seq))`: data points are plotted at equally spaced integer x coordinates starting from 0. The default behavior is to plot the data as a 2D *solid line*, such that `plot()` should be seen as a *line plot*.

- `plt.plot(x_seq, y_seq)` plots the given *pairs* $(x_{seq}, y_{seq})_i$, $i = 0, \ldots, n$, where n is the length of the sequences. Again, the default behavior is to plot the data as a 2D *solid line*.

- `plot()` can be customized by setting many positional and keyword-passed parameters, a few examples are given below, different combinations of the parameters can be used to get different effects:

  - `plt.plot(x, y, marker='.', markersize=4)`
  - `plt.plot(x, y, linestyle='None', marker='v', color='b')`
  - `plt.plot(x, y, linestyle='--', linewidth=2, color='r')`
  - `plt.plot(x, y, linestyle='-', linewidth=1, marker='s', markersize=8, color=(0.1, 0.4, 0.3))`
  - `plt.plot(x, y, 'ro', linestyle=':')`

*Contact G. A. Di Caro for pointing out mistakes, missing information, and for suggestions (gdicaro@cmu.edu).
  ```

- `plt.scatter(x_seq, y_seq)` makes a *scatter plot* of the pairs $(x_{seq}, y_{seq})_i$, $i = 0, \ldots, n$: no lines are drawn between points. A scatter plot *needs* two input sequences to create the point pairs.

- `plt.scatter(x, y, marker='.', s=12, color='r')` makes a scatter plot using the selected marker, defining its size with the keyword parameter `s`, and setting the color. `color` and `marker` arguments follow the same options of the `plot()` method.

- `plt.hist(values, n_intervals)` plots a *histogram* for the data in `values` by considering their distribution in `n_intervals` bins/intervals of the same size. The color of the histogram can be specified by using the keyword argument `color`, a normalized histogram is obtained by setting the option `density=True`.

- `plt.bar(x_pos, heights)` makes a *bar plot* where `x_pos` are the x coordinates of the bars, and `heights` are the heights of the bars. The optional argument `width` can be used to control the width of the bars (default is 0.8). Filled color can be set by the optional argument `color`. Tick labels on the x axis can be set with the argument `tick_label`, for instance `tick_label = ['1998', '2010', '2015']`, or `tick_label= [100, 200, 1000]`.

- `plt.pie(values)` makes a *pie chart* of data `values` where the fractional area of each wedge i is given by `values[i]/sum(values)`. The `labels` optional parameter allows to pass a list of strings for the labels of each wedge. The optional parameter `colors` allows to specify a sequence of colors for the wedges (matplotlib will cycle the sequence if it is less than the number of wedges).

- `plt.legend(handles=list_of_label_references)` allows to place a *legend* for each element in the plot:

  ```
  plt.figure()
  y, = plt.plot(x, y, label='y')
  xx, = plt.plot(x, x**2, label='$x^2$')
  plt.legend(handles=[y, xx])
  plt.show()
  ```

  Note the commas after `y` and `xx`, as well as the use of `$ $` to enclose and expression using the LaTeX syntax to generate a nice math-like formatting of the output.

- `plt.show()` completes the figure and shows it on the screen. It is not strictly necessary, but it's better to include it.

Notes

Notes

Numpy module

The `numpy` module is the main module providing mathematical and numerical tools. We haven't really explored the potentialities of the module apart from using the basic method `arange(from, to, step)` that generalizes the built-in function range to float numbers. Below a few methods and examples from the library are listed:

- `import numpy as np` is the typical way to import the module.

- `seq = np.arange(from, to, step)` where `from, to, step` are floats, for the rest it follows the same rules as `range()`.

- If `x` and `y` are created as numpy *arrays*, then arithmetic operators can be applied to them (as long as the arrays have the same length):

  ```
  x = np.arange(0, 1, 0.1)
  y = np.arange(2, 3, 0.1)
  z = x + y
  zz = x * y
  ```

- `numpy` provides most of the mathematical functions provided by `math`, with the same names, such `np.sqrt(x), np.sin(x)`, etc.

- `np.average(z), np.median(z)`

Notes

Random module

Random number generation can be used for Monte Carlo *simulation*, as well as in a number of other useful tasks. The `random` module provides ways to generate *pseudo-random* numbers according to many different modalities. We only have focused on simple *uniform* random generation in a given interval of real or integer numbers, or over a provided sequence of symbols. Below the considered methods are listed, together with the import statement:

- `import random`

- `random.seed(seed=None)` initializes the random number generator based on the the value of `seed`; if the integer argument `seed` is not given, the seed is initialized in an automatic way (Python takes care of it).

- `r_in_seq = random.choice(sequence)` returns an element from `sequence` selected in a random uniform way.

- `r_in_range = random.randint(from, to)` returns an integer number from the integer interval between `from` and `two` selected in a random uniform way.

- `r_in_range = random.uniform(from, to)` returns a real number from the real interval between `from` and `two` selected in a random uniform way.

Notes

Notes

Complexity

Please refer to the lecture slides for the discussions on the time complexity of an algorithm. Below some basic definitions and arguments to check are pointed out.

- Counting the number of *basic operations* performed by an algorithm is a way to make an assessment about the *running time* of an algorithm which is machine- and implementation-independent.

  Basic operations include:

  - Assignments
  - Arithmetic operations
  - Relational operations
  - Indexing, memory access operations

- *Time complexity*, depends, in general, on *size* and *value* of the input.

- *Worst-case analysis* for the running time is a way to eliminate the dependence on the value of the input, and provides an *upper bound* on the expected running time.

- In time complexity analysis we are interested in the relative comparison between algorithms for very large sizes of the inputs n, that is, for $n \to \infty$. Therefore, we can drop additive constants and multiplicative factors when representing the asymptotic behavior of the running time of an algorithm.

- Based on the previous assumptions, the asymptotic behavior of the running time of an algorithm is described using the *big O* notation, that gives an *upper bound on the asymptotic growth* of a function $f(n)$ representing the running time of an algorithm as a *function of its input size n.*

- An algorithm with time complexity $O(1)$ has *constant* running time (independent from the inputs).

- An algorithm with time complexity $O(n)$ has running time that grows *linearly* with the size of the input.
- An algorithm with time complexity $O(\log n)$ has running time that grows *logarithmically* with the size of the input.
- An algorithm with time complexity $O(n^2)$ has running time that grows *quadratically* with the size of the input.
- An algorithm with time complexity $O(n^k)$ has running time that grows *polynomially* with the size of the input.
- An algorithm with time complexity $O(k^n)$ has running time that grows *exponentially* with the size of the input.
- An algorithm with time complexity $O(n!)$ has running time that grows *factorially* with the size of the input.

Notes