Scientific Programming

Practical exercises on Python - I

Julien Bloino Marco Mendolicchio

May 18, 2020

Exercise 1:

Write a small program, which returns all numbers between 1000 and 2000 (included) divisible by 6 but not multiple of 9.

Exercise 2:

Write a program asking the user for a positive integer between 0 and 10000. The program should meet the following requirements:

- check if this is a valid integer and within the range.
- ask the user to give a proper value if incorrect.
- give the possibility for the user to quit.
- print the number given, the number of digits in the number and their sum.

Extra: How could the number be checked using regular expressions?

Exercise 3:

Write a program asking the user to choose a color between 'red', 'blue', 'green', 'purple', 'yellow' and checking that the value is correct. The choice should be case-insensitive.

Exercise 4:

Write a function which takes 2 real numbers as arguments and returns True if the first value is smaller in absolute value than the second one. An optional argument should give the possibility to inverse the test (returns True if the first argument is larger). The function should raise an exception if the 2 arguments are not float. The function should also support other keyword-based arguments and raise an error about the lack of implementation if such keywords are given.

Write a small test program in the same source file giving the possibility for a user to test the function. The program should expect at least 2 values and optionally 3 to be passed to the function and return in output the values in exponential form with 8 significant digits and the result of the function.

Exercise 5:

This exercise will parse a JCAMP-DX file given by a user in input and plot the results.

Reminder: The Joint Committee on Atomic and Molecular Physical Data exchange (JCAMP-DX, `http://www.jcamp-dx.org`) format is a file format specification proposed by the International Union of Pure and Applied Chemistry (IUPAC) committee to facilitate the exchange of spectroscopic data, primarily related to Nuclear Magnetic Resonance (NMR). This format has been adopted by various organizations, for instance the American National Institute of Standards and Technology (NIST, `http://nist.gov`), which uses it as the export format for the spectra stored in their database (`http://webbook.`

nist.gov/chemistry/). This format is not readable directly by most plotting software, and must be converted to a more common format. Such a format is provided by the comma-separated (alternatively, character-separated) values (CSV) file format, where data are stored as columns separated by a specific type of characters (plotters may differ on this part). Some additional information on this format can be found at http://en.wikipedia.org/wiki/Comma-separated_values.

Exercise 5.1:

Write a program taking in argument a filename and checking its existence. The program should be able to list alternative files with a jdx extension in the directory where the file was supposed to be (a parsing of the filename given in input for possible directories may be necessary).

Exercise 5.2:

Write a function, which parses a jdx file, and returns a dictionary of all present keys, assuming that they have the form "##KEY=VALUE". The function should also support an optional argument to search for a specific keyword. In this case, only the key and value corresponding to the keyword should be returned. An error should be raised if the keyword cannot be found.

Exercise 5.3:

Write a program to obtain δx in two ways (one by reading directly the value associated to the keyword DELTAX and one by calculating it from the range of energy).

Print the two values and print the percentage error between the saved value and the calculated value (the latter used as reference).

Exercise 5.4:

Extract the x and y axes values from the file and print all the values with at lest eight digits of precision. Note that some x values need to be computed. Alternatively, if a user has given a second file in argument, the program should check if the file exists, and asks if the file should be overwritten. If the file does not exist or can be overwritten the program should write the data in the file. Otherwise, it prints it as usual.
Note: the frequencies and intensities are stored in a multi-column format, with the following layout:

$$\begin{array}{lllll} x_i & y_i & y_{i+1} & y_{i+2} & \cdots & y_{i+m} \\ x_{i+m+1} & y_{i+m+1} & y_{i+m+2} & \cdots & y_{i+2m} \end{array}$$

Note: m may vary between jdx files.

Exercise 6:

The "top" command allow users to visualize and monitor all Linux processes. Based on the available documentation, the "top" program provides a dynamic real-time view of a running system. It can display system summary information, as well as a list of processes or threads currently being managed by the Linux kernel.

Exercise 6.1:

Write a Python program to run a single iteration of the top command and save the result in a file specified by the user. If the file already exists, the program should print a warning message and ask if it is ok to overwrite it. Read the file previously generated and identify all processes presenting a CPU usage (%CPU) larger than a threshold defined by the user (default: 5.0). Regular expressions can be used to facilitate the parsing of the data. The "sys" or "argparse" modules can be used to parse commandline arguments.
Note: the command "top -b -n 1" executes a single iteration (or frame) of top and sends it to a program or a file.

Exercise 6.2:

Write a Python program to run a single iteration of the top command and identify all processes presenting a CPU usage (%CPU) larger than a threshold defined by the user as percentage (default: 5.0). This time do not use any external file to parse the top command output.
Regular expressions can be used to facilitate the parsing of the data.

Exercise 6.3:

Write a class "Task", characterized by the following attributes,

- process ID (PID),
- user (USER),
- CPU usage (%CPU),
- time (TIME+), internally stored in the seconds,
- command (COMMAND),

properly encapsulated.

Note: The standard format of time in "top" is: "*hours:minutes.seconds*"

Exercise 6.4:

Write a program to run a single iteration of the top command and save all processes in a list of objects of the class "Task".

6.4a identify and print all processes for which the CPU usage is above a threshold defined by the user (default: 5.0);

6.4b sort by PID, time or CPU usage and print all processes. The sorting order (ascending or descending) is supposed to be provided by the user (default: ascending);

6.4c sort all processes in bins on the basis of the CPU usage. The user should be able to give the number of intervals or the width of each bin (the boundaries are given by the maximum and minimum values of the CPU usage).

6.4d print the processes characterized by a time above a threshold specified by the user. The threshold can be given in seconds or as "hours:minutes".

Exercise 7

Exercise 7.1:

Write a function that takes 2 arguments a and b, each one of them being a float or a list of floats, and computes their sum as follows:

- if two scalars are given, computes and returns their sum;
- if a list a and a scalar b are given, adds a to each element of b and returns a list of sums;
- if two lists are given, checks that their length are equal, computes the sum element-by-element and returns a list of sums.

The function must be independent on the order of the arguments.

Exercise 7.2:

Starting from **exercise 7.1**, and add the following user-dependent features:

- choice between the absolute and actual values of a and b;
- choice between the sum and average operation of a and b.