L01-PythonBasics

December 9, 2024

Supporting textbook chapters for week 1: 2, 3 and 4.3

This is an example of "lecture notes". As you will quickly find out, this course is by nature a lab course. Therefore, my "lecture notes" will not often follow the linear progression of regular lecture notes. This is particularly true for this first lecture, in which I merely want to give you pointers to do the first lab.

1 General Problem-Solving Approach

1. **Math model:** often, but not always, continuous.
2. **Translate and discretize:** set up discrete arrays of independent variables (e.g., x, t), dependent variables (e.g. $v(t)$, $a(t)$), and define operators on these variables (dv/dt, ma...) according to the model.
3. **Initialize** parameters and variables appropriately.
4. **Evaluate:** run algorithms (using the operators) on these variables.
5. **Debug:** bang your head against the wall for awhile to figure out why it didn't work properly, fix it, repeat as many times as necessary.
6. **Analyze:** some extra processing of the raw results (figures, etc.)
7. **Sanity check:** ask yourself whether your analysis outputs make sense. If not, go back to step 5.

2 Coding Tips

Modularity is the concept of breaking up your code into pieces that are as independent as possible from each other. That way, if anything goes wrong, you can test each piece independently.

- Python scripts: define external functions for repetitive tasks. Place them in a separate file (called e.g. MyFunctions.py) in the same folder as your main code file. In your main file, import MyFunctions, and call and use the external functions.
- Notebooks: define functions in dedicated cells. Run them before you run the cells containing your main code.

Give your parameters and variables meaningful names - reserve i, j, k, etc. for loop counters - careful about capitalization

Avoid hard-coding parameter values. It should be easy to find and set these values somewhere near the top of your code.

Test your code as you go, not just when it is finished.

Debugging: - carefully read any error messages produced - add print statements to see intermediate values of variables - before you try looping with several iterations, try just one iteration - you may want to use a built-in debugger in an IDE - just because the code runs without errors, doesn't mean it's bug-free. Remember, there's almost always another bug sitting somewhere in your code.

Sanity check: - are all output arrays the length you expected? do all the plots have the number of points (or histograms the number of entries) you expected? - are all output values the correct order of magnitude? - are all curves in your plots the expected shapes? - if possible, check the result (maybe for a simple case) analytically or by-hand or using a different computational method

3 Pseudocode and Comments

The concept of "Pseudocode" is loosely defined, see for example https://en.wikipedia.org/wiki/Pseudocode. In this lecture, I will use a **very** loose definition for it, i.e., mostly plain English with bullet points. You are free to use your own version. Just make sure that it is understandable by someone who speaks English and is vaguely familiar with the problem you're trying to solve.

- Pseudocode is the planned version of your code. It describes your algorithm(s).
- Writing pseudocode helps ensure that your planned logic for the algorithm is sound. Therefore you should write pseudocode before starting any actual code.
- Pseudocode should be concise, logical, step-by-step.
- Coding = turning your pseudocode → specific programming language. You should be able to take your pseudocode and convert it into any typical programming language.
- You may distribute your pseudocode throughout your real code, in the form of **comments**, as you go through the process of coding. Also **keep a copy of the pseudocode intact** so you can refer back to it, and ensure each block of code starts with a brief overview of what the block will do.

Examples for sequential stuff: * Input: `READ`, `OBTAIN`, `GET` * Initialize: `SET`, `DEFINE` * Compute: `COMPUTE`, `CALCULATE`, `DETERMINE` * Add one: `INCREMENT`, `BUMP` * Output: `PRINT`, `DISPLAY`, `PLOT`, `WRITE`

Examples for conditions and loops: * `WHILE`, `IF-THEN-ELSE`, `REPEAT-UNTIL`, `CASE`, `FOR`

Should also include calling functions: * `CALL`

4 Learning (or Reviewing) the Basics

Ensure that you take the time this week to learn (or review) the material in Chapters 2 and 3 as this will be expected knowledge for all the future labs. Particularly useful review material includes: - Assigning variables: Sections 2.1, 2.2.1 - Mathematical operations: Section 2.2.4 - Loops: Sections 2.3 and 2.5 - Lists and Arrays: Section 2.4 - User defined functions: Section 2.6 - Making basic graphs: Section 3.1

[ ]: