1 Coding as a Creative Process

In the previous chapters, we reviewed the various elements of a programming language (variables, conditionals, loops, etc .). However, more often than not, the challenge is not the mastery of those elements but rather the skill to combine them in a useful manner.

Coding requires you to solve a problem for which you see no obvious solution. And the only way out is to hit the "OK, let's be creative button". For some of you, this will work. For others, not so much, since few of us are trained or gifted creative thinkers. That being said, there are strategies you can employ that will help you to find solutions. You will need those strategies in pretty much all of the following coding projects.

This chapter is inspired by "Think like a Programmer" by V. A. Spraul. Unlike most programming books, it focuses on problem-solving rather than computing. Unfortunately, all its exercises are in `C++`, but the introductory and conclusions chapters are highly recommended reading.

The book's basic premise is that most students do not struggle with a computing language per se but rather with the process of using a program to solve a problem. The following chapter is a heavily condensed version of Chapter 1 in Spraul's book.

1.1 General Problem-Solving Strategies

Problem-solving skills vary tremendously between people, and it is not something you can learn like any other method because it is an inherently creative process, not a methodical one. However, you will get better with practice, and a few guidelines can help with the process. The following list will prove helpful whenever you struggle with a problem where there is no obvious solution:

Make a plan: Even if you don't know what is going on, you can always write down a plan. Think of mapping, you may not have the faintest clue of what you are dealing with, but you know that by methodical mapping (taking stations, measuring strike and dip, mapping perpendicular to strike, etc.), you will zero in on the solution. The same is true for programming. Use the following steps as a guideline to develop a plan. Start by making a list of steps you need, even though you may not know how to solve each step. You will likely have to refine your ideas as you go, but this is still infinitely more useful than just random exploration.

1. **Restate the Problem:** An excellent way to achieve this is to restate the problem in your own words. Next, explain how you see the task to one of your peers (or your instructor) for comments. This is often the first and most crucial step in formulating a plan. Remember to describe the goal, not the way.

2. **Divide the Problem:** Most complex problems can be understood as a series of smaller problems. In coding terms, we would think of loops, if-statements, etc... This is the part where you start writing high-level pseudo-code. Pseudo-code is like short form, but it is not actual code:

```
while loop to get user input
  y = input

for loop to test x>y
t = transform_magic(x,y)
print result
```

You can do this even though you may not yet know how to code the individual blocks.

3. **Start with what you know:** This is like writing an exam. Always start with the pieces you already know how to solve. In doing so, you may find essential clues on how to solve the pieces you don't know.

4. **Reduce the Problem:** Say you have to write code that converts any number into its binary notation. To simplify the problem, you can state that your function will only deal with positive integers. Or you can further simplify the problem to the point that your code will only deal with numbers between 0 and 10 etc. Once you have an intermediate solution, you will have gained much clarity on how to solve the entire problem or what specific questions to ask your peers or instructor.

5. **Look for Analogies:** Often, you may be able to reduce your problem so that you can recognize that you already solved a similar problem elsewhere. Consider the case of strings, which are just a specific type of list. The exact commands may not match, but now you probably remember how to query the methods associated with a list object or

that you need to ask the specific question "how do I replace the third element in a string".

6. **Experiment:** Often, it is useful to test the workings of a command or code sequence in a separate cell. The key techniques here are the idea to isolate the code snippet from your main code, and to test whether it does what you think it does. E.g., consider the following `a[3]: str = 4`. This may work well in your code, but it may not do what you expect. So a little experiment can go a long way... Similarly, test and explore new commands before adding them to your code.

7. **Don't get frustrated:** When you are frustrated, you won't think clearly, and everything will take twice as long. Take a walk, rethink the problem (i.e., use the above steps), and come back to the problem when you are ready. Then, you can try to reduce the problem to its most fundamental form, e.g., take a piece of the code out of your program and run it in isolation; use print statements to check whether your variables have indeed the values you think they have. Once you have a minimal example that shows the problem, ask for help. Or if all else fail, have a look at your plan and see if you can work on something else. If need be, take a creative break and work on something else for a while. To quote from Spraul's book:

   When you allow yourself to get frustrated - and I use the word "allow" deliberately - you are, in effect, giving yourself an excuse to continue to fail. Suppose you're working on a difficult problem and you feel your frustration rise. Hours later, you look back at an afternoon of gritted teeth and pencils snapped in anger and tell yourself that you would have made real progress if you had been able to calm down. In truth, you may have decided that giving in to your anger was easier than facing the difficult problem.