Summary: Mathematica is best tool for teaching good programming better, faster, easier

Why Mathematica wins as a first programming language

For use in Augment™ training:
To learn and build successful programming habits.
To avoid acquiring, by accident, failure-prone programming habits.

Table of Contents

All Engineers will learn multiple programming languages

We believe it is best to start with Mathematica

Auto mechanics are required to use diagnostic computers because all modern cars have computers in them. Similarly engineers will not be able to engineer anything without using several programming languages. Unfortunately most engineers have only limited training in programming. Thus, wading into a career, it is not so easy to know how well they have consolidated their daily "lessons learned" into deployable programming skills which augment their engineering capabilities. The Big Question is, 'How do you know what you know?'

1. Programming languages are inherently difficult to learn and retain.
Technologically constructed environments tend to be highly uniform. The UI gets integrated over a long period of time and widespread adoption (like cars -- you can drive almost any car without needing to be retrained). Integration in this case means nearly uniform terms, controls, and operations. **Uniformity greatly simplifies learning and recall because memories of**

patterns can be 'reused'. Little such uniformity exists with programming languages. There are significant differences between languages which subtly but significantly limit skill transfer. As an analogy--if you downhill ski, you can transfer your skill to cross-country skiing pretty easily. Transfer between programming languages is not obvious, nor direct, nor intuitive.

2. New knowledge and skills must be converted to automatic (subconscious) behaviors.
This is called habituation or automaticity. Habituation is limited by two factors: (1) long term calendar time (in months) and (2) congruence of the environment. The calendar factor cannot be shortened so much as held in place by intermediate scaffolding until long-term memory takes hold and consolidates and automates the multiplicity of perceptions, skills, actions, effects with feedback loop, etc. More experience is everything. The environment, however, can be engineered so it makes more sense. For example the user interface in an automobiles has 'converged' and each item fits well with the other. The turn indicator is predictably on the left of the steering column. And we see immediately the problem of counter-intuitive non-uniformity being very dangerous. Programming environments lack uniformity, convergence, congruence, predictability...and thus habits are much harder to transfer and you often need to retrain skills from the beginning. Much like learning to drive all over again in London.

3. When learning to drive, we just want to focus on the essential tasks
When learning to drive a car, the essential task is keeping the car moving, on the right side of the line, while avoiding obstacles. Learning to diagnose mechanical difficulties, understand periodic maintenance, or navigate via paper maps are all largely arbitrary. Each depends on many contextual conditions, such as the type of the engine in the car and whether you are driving in a city or in the country. If the essential learning task is learning to drive the car then all the rest ought to be set aside until you can drive safely.

4. Same with programming
The essential learning task ought to be the primary focus and the rest set aside. When we say learning programming is hard, we mean you bear heavy cognitive load. Attention, focus, memory, etc. must be coordinated without lapses. Otherwise the cognitive load is split between essential and nonessential learning, mistakes are made and the process of learning takes much longer and the learner retains less capability. I.e. If you have forgotten the lesson by tomorrow, you have not learned much. In the case of programming the cognitive load is very high -- roughly a function of the square of the new tasks being learned. When driving a car, the cognitive task is almost automatic from the beginning -- keep the car moving on the right of the line, at a slow speed, and watch for obstructions of any sort. But programming is far more difficult because the learning task requires thinking using abstractions within abstractions, Often the difficulty is trying to keep abstractions appropriately separate in the mind's eye, so to speak. Little things are not so little when you are trying to learn a great many little things -- each of which, done with slight miscalculation, might cause the program to fail.

5. There is not a good name for what we loosely call 'programming'.

Programming has come to be used as a label for a host of activities -- The word is used for a child telling a turtle program to go FORWARD, LEFT, BACK, RIGHT, etc. The same word is used for an adult telling a space shuttle to go to the moon. As an analogy, telling a skilled gymnast that all they do is a form of running and jumping -- and reasoning anybody can do it because anybody can run and jump would not prove out well for anyone trying to perform a backward somersault on a beam. This type of reasoning and behavior is called dilettantism -- pretending you can do something by doing a part of it. Adding 2+2 does not make you a mathematician and would not even impress that child with the turtle program. It is all too easy to think you have become an expert programmer when you in fact are barely competent.

6. The Big Question -- what *is* programming -- is not at all obvious, as we shall see.

Pounding a nail or sawing a board is not carpentry -- building a house to code is. Using a computer-based symbolic language to instruct the computer to perform complex actions is most certainly programming -- but getting past the programming equivalent of 2+2 (often called “Hello World”) -- is difficult, as most who try quickly find out.

7. Programming is about expressions.

For this document -- and to understand the general idea of programming in a more deeply insightful way -- programming might be considered to be the chaining of expressions, each of which contains a single complete concept. Programming is about organizing our thoughts systematically inside a tool that will do useful work, i.e. a computer, reliably, economically, in the real world. Whichever language you use, that is the objective.

8. For beginner programmer, essential task is forming and manipulating expressions

In the beginning stages in Mathematica, we just want to focus on the essential task which is to learn to write one or two lines of code which form a complete expression to be evaluated. This is Novice programming.

> Other aspects of programming, such as those associated with the operating system or the User Interface are also programming -- but they are of a much different sort. The tasks are more arbitrary and deal with the contextual needs. We want to ignore these in the beginning. Just as in the car analogy, first focus on safe driving -- worry about navigating downtown with a paper map later.

9. Isolating the different contexts of concepts used in programming:

1. Math and programming are in many ways very similar. Math has operations, notation, syntax, and logic. So does programming. And like driving a stick shift car you have to operate the gas, brake, clutch, and shift all at the same time -- and exactly -- or you stall the engine. Same with programming. Your code simply will not work.
2. Programming is logical and exact. We know logically 2+2 = 4 -- but sometimes the computer thinks 2 is the same as 1.999999999999999 and thus 2+2 ≠ 4.
3. You can ask your program to divide 1 by zero and your code will fail. If you shift your car into neutral accidentally when struggling up a hill with a heavy load you will race your engine past the red line and it will explode.

10. Three principles to observe while learning programming:

1. **Learn only essential skills.**
2. **One new skill at a time.** Use a progressive example, where you know the answer at each additional experiment with another or more complex operation, syntax, notation, etc.. E.g. Draw a point, a line, a square, a circle, a disk, an arc, a circle, a sphere. Learning programming by doing graphic programming examples first perhaps is the easiest. Preferably try to have the output in graphic form so it can be inspected visually at a glance.

3. **Build on solid understanding only:** It is easy to outwit yourself. Keep the total number of different, non-automatic operations as small as possible --even if you think you understand them. Perceived complexity goes up with the square of the total number of things you are doing. There is a difference between watching 1 independent variable, 2, 3, …, n -- to the tune of 1,4,9, …, n^2 -- in effort to watch them. Discretionary or available cognitive resources are quite volatile once the program has crashed. Most everyone probably could learn to juggle three oranges but have you ever tried to do it? So if your program has five different operations, maybe drop two of them and then add the new skill/operation. Keep your examples in an archive to refer to at a later time if you forget something.

11. NOW we can say why learning to program first in Mathematica is a good idea

Most of these above conditions are very easy to sequence and coordinate. And if you do something wrong, Mathematica will tell you fairly quickly something is wrong. If you go slow you can learn the operating principles (at least the default form) one at a time and easily. Operations, syntax, notation, and logic 'fit' together fairly intuitively.

But there is more: Mathematica operations resemble mathematical operations you already are familiar with ---probably-- if you have a technical major. Cognitive load is eased because you already know the concepts having learned them in a not too different language.

So Mathematica is like Legos. It is hard to make mistakes of construction since the rules are obvious visually --you can visually inspect your ideas before assembly them.

But there is more: Given these advantages we can talk more about what it is when you are learning programming. You are learning how to put together one action with another action. Mathematically, you are creating an expression which processes inputs into outputs. E.g. Create an integer function, create a list of integers with the function, plot the list of integers using a graphic function.

Most programs have user interface aspects and computational aspects. Learning to program user interfaces is more about human performance engineering following rule systems for UIs. However most programs which do interesting things use numbers or numeric quantities:

dimensions, lists, time, sequences, metric units, measurements, etc. Learning to program is far more associated with logical ie numeric type concepts. And yes, there other forms of programming (i.e. object programming) -- However the concepts we are suggesting here would be very similarly done. Programming is more about learning how to assemble expressions to do useful work and such tasks are essentially the same in procedural (Basic), functional (Mathematica), or object (Smalltalk) languages. So we start with learning to program in functional language (Mathematica) and explain as we go.

Operations (i.e. Expressions) are created which process numbers or variables into useful schemes.

Example Programs:
What would a car cost if assembled from various catalog items?
How might I write a program which allow "what-if" cars for different budgets?

Perhaps more practical examples for most students:
Is it better to repair or replace my junker car, given my present cash savings?
Write a program which computes purchase and operating costs, including probability of repair costs for various makes/models of cars?

These programs would be created by assembling one expression, then adding it to another, and so on.

12. Moon gravity juggling:
All of the above scheme of explanation might be summarized in another way using an analogy of learning to juggle, say three oranges. Most people can juggle one orange between two hands --but that is hardly juggling and what you learn with one orange does not make it easier to juggle two oranges. And by three oranges, almost all people fail nearly every single time they try. No consolidation of skills is achieved and eventually most people quit.

Well, some clever person suggested a real time virtual learning program. Perhaps they ran a thought experiment and got the idea that learning to juggle on the moon would be much easier as the oranges would stay up longer and you could 'learn' how to move your hands in a coordinated way -- because you wouldn't be dropping every every time you tried. Then imagine a virtual learning juggling application where, by turning a knob, the now programmably controlled gravity gradually increases. You could gradually learn to juggle faster and faster until you were juggling at 1g as on Earth. This idea was implemented with special motion-sensing gloves (and no physical oranges) and a special program which showed virtual oranges moving in parabolas on a screen at ½g, as you juggled the virtual oranges with your hands. Then the virtual gravity would be turned up until you could juggle first two, then three oranges. Interestingly most of the learning has to do with eye-hand coordination and not so much the physical feel of the oranges. So you can practice and succeed on Earth with first two then three oranges having learned the spatial awareness aspects first. Cognitive load now only requires

learning about the feel of the moving oranges in the hands as they come and go with increasing velocity. Thus the virtual orange juggling program works --because it uses the principle of reducing cognitive load. Now imagine an imaginary programming environment where the different cognitive tasks are separated and reduced in effect...and then turned up as you write more complex codes. That is how we are recommending learning programming. Mathematica is well-suited to do just this approach.

Other languages (3gls such as Java, C++, C#) do not work well at all in this sense. This is because even simple programs in these 3gls are voluminous. Hence, they are difficult to physically view, because they take at least several pages and accomplish very little. You are looking at sparse terrain. It is hard to maintain focus as there is little to imagine but much to do in understanding the code dependencies. So the simple explanation is that they don't juggle well. It takes a very long time to automatically write and understand simultaneously several pages of code, how the computer will execute the code, and how to sort out unexpected behavior of incorrect code. What is so intriguing about Mathematica is that you can solve very complex problems with just a few lines of code. Testing is often not necessary as you can see everything and inspect correctness at a glance.

Long term learning, often called consolidation or 'automaticity', requires changes in brain cells. It's a now well-documented physical change called myelination. Creation of new mental routines (which we call habits) takes about six months. Major consolidations of learning will take place roughly every six months. Learning to program on an automatic basis will take 3-6 cycles, depending on prior knowledge -- or about 2-3 years. Mathematica offloads some of that memory burden in the structure of the language since STEM people already have myelinated many cycles of higher math. Because of this prior deep learning, you can do real work even before new 6 and 12 month neurological (myelination) cycles have been achieved.