PS: Debugging (Tell it to the duck!)

 The second problem solving activity we will consider in this exploration is Code Debugging.

We have a bit of a chicken and egg (not to be confused with ducks!) problem here, as we have not yet completed the Programming Modules in this course. However, it should be helpful to simply consider the following bullet points about debugging, written at a high level of abstraction. In the upcoming programming explorations, these high level ideas will be re-introduced.

Debugging is the process of removing errors from programming code. The debugging process is common and frustrating enough that software tools continue to evolve to aid in the process.

Debugging tools have come a long way:

- Development environments and editors now include tools such as code completion, real time syntax checking, and more. (Contrast this to a binary data dump of 0's and 1's on a printout with multiple shades of green rows...)
- Programming languages have evolved to support block-based languages, in which the syntax has been abstracted away into blocks. (Contrast this to text based programming, where a teacher could spend hours chasing a missing closing paragraph symbol ...)
- Planning tools (from sticky notes to electronic displays) now facilitate individual and collaborative algorithmic development. (Contrast this to plastic pocket-fitting flow chart symbol cut outs...)

The programming unit will look at debugging in more detail, but let's begin a high level list here:

1. There are two kinds of errors in misbehaving code
   A. **Syntax errors:** e.g. you left out the semi-colon, after all
   B. **Semantic errors:** e.g. the code runs, but isn't to specification or intent, such as use of an un-initialized variable
2. Finding errors is not always straight forward. If you forget a closing parentheses, for example, most translators and editors will scan down the code and declare a match on the first closing parentheses encountered--even if it is part of a different set of parentheses. This is why feedback identifying a specific line of code in error should be interpreted as "the errant line of code is **around this location**", and also why fixing one error can often make many errors disappear.
3. The best way to find an error is to "trap it". If a programmer hasn't developed fluencies yet in tools that support code tracing through break points, etc., a significant amount of de-bugging can be accomplished through the insertion of simple print statements. "I'm inside the grade averaging function" can assure that a program didn't terminate before a particular function was called. Once debugging has been completed, the print statements are removed before "product" release.
4. Years ago, some amazing teacher stole an industry practice that helped students debug and polish code. Interpreted in the school setting, the practice utilizes rubber ducks. [It is a tradition in

computer science courses to provide ducks to students. (The College Board is asked every year if students can have their ducks with them during final exams; as of yet, the answer remains "no".)] Here's how it works: a student is trying to figure out what is wrong with her code. Instead of calling the teacher over, she explains code operation to the duck. A significant percentage of the time, the act of verbalizing thoughts aloud can give focus to an error. And then we as teachers get to hear that wonderful statement: "Never mind, I figured it out". You can read about the tradition of duck debugging **here, (https://en.wikipedia.org/wiki/Rubber_duck_debugging)** and your students might enjoy watching one of the many YouTube videos about the practice, such as this one that explains the process locally, and then positions the practice within the framework of general problem solving.

youTube The Rubber Duck Story (https://youtu.be/huOPVqztPdc)

(https://youtu.be/huOPVqztPdc)

The YouTube author considers the duck tradition first in computer science, and concludes by pointing out the process has interdisciplinary value, which seems like a perfect place to end our exploration of problem solving, and look at problem solving differentiated by grade band.

K-8 teachers should complete differentiated learning materials by clicking on the Next button, below.

9-12 teachers are encouraged to browse the K-8 materials, but may choose to **skip to the 9-12 materials (https://iu.instructure.com/courses/1903143/pages/ps-differentiated-learning-9-12)** .