ETA Week 9 – Summary Review

This week we had a visit from Scott Miles who provided an introduction to Ruby programming.

Scott started the session by discussing, in broad terms what programming is. He then spoke about why knowing how to program, or knowing things about programming or code, might help you to test. Being able to understand what a developer is talking about, and being able to describe problems in more detail can help communication. Knowing things such as maximum values within specific architectures can help you discover important problems. Scott also mentioned a quite close relationship between testing and programming – automation. Automated testing could also be referred to as programmed testing (or programmed checking).

The introduction focused on enabling you to create a program that output a comment, “This programming is running”, to the screen. This was an introduction the “puts” command. Next we added a layer of complexity by adding a variable. The variable was called name and the program was structured to output Hello and the name variable to your computer screen. Then to make it just a little more interesting we added a gets to allow entry of a name rather than just storing a name in the program. It looked like below:

puts “What is your name?”
name = gets
puts “Hello” + name

Lesson from this exercise – text case is important.

At this point Scott suggested we increase the complexity just a bit and changed our focus to math. Starting with simple math we saw how simple it can be to get Ruby to solve equations. Don’t forget to use the “puts” instruction to output the calculation answer to your screen. During this process, we discovered that to get answers that involve decimals, you need to declare that decimals are allowed (“1.0/3.0” rather than “1/3”). We also saw that Ruby has output limits and applies the standard mathematical processing rule “BODMAS”. Other things we explored:

- modulus operation
- algebra

The next thing we did was to build a program that counts from a starting point to a finishing point. This required using a “for” loop. It also had you identifying what you wanted the program to do, bit by bit. You might not have even realised that you were incrementally building a program, getting bits to work in a limited way then increasing the functionality and complexity.

After a break Scott introduced the idea of data types. Data types used in the first session:

- integers (whole numbers)
- floats (decimals)
- strings (text)

We discovered that Ruby treats strings and integers differently. Joining two strings together is called concatenation.
a.downcase = a way of getting Ruby to change user input to lower case letters. This can be used to manipulate input in a variety of ways. It will not work on integers. Some other ways of manipulating strings include “a.reverse”, “a.upcase” and “a.uniq”.

You can also manipulate integers. For example, “puts 5>4” will return a value of true and “puts 5<4” will return false. You can also use expressions such as “>=” or “<=”.
When using “=” you are assigning a value, when you use “==” you are asking Ruby to compare values and decide if they are equal (true) or not equal (false). You could change this around and

use “!=” (not equals). This was followed by an example of how to convert a string to an integer (using a.to_i), or an integer to a string (using a.to_s).

Then a challenge was issued, build a calculator from the lessons we have learned to date. This required taking two integers from the user and adding them together. Well done people, you got there.

It was then time for something just a little more complex. A program that takes a test score, entered by a user and tells them what grade that score is equivalent to. You’re now asking the program to take a value, compare values to those we have defined and find an appropriate letter grade. Once the grade is found, return it to the screen for the user. Again this program was built in pieces and each piece tested before moving to the next bit (debugging as you go).
This exercise required the use of “if” statements. These can get tricky. To help with the complexity, and avoid meeting more than one comparison condition, we added “elsif”. Scott showed us how to structure a set of conditions where a number falls into a single unique grade partition, it cannot be matched to multiple grade partitions.