# Educational Programming Languages: The Motivation to Learn with Sonic Pi

Arabella Jane Sinclair[1]

Computer Laboratory 1, Cambridge University
ajs291@cl.cam.ac.uk

**Abstract.** This work explores the differences in novice users experiences when learning a simplified variant of the Ruby programming language for the first time. Sonic Pi and Kids Ruby both aim to teach users via a media-oriented set of programming exercises and environment, on the premise that this accessible domain will motivate novice learners. Users retention of the language and interaction style with each environment were measured and the results indicate that Sonic Pi facilitates a greater level of user experimentation and concept retention.

## 1 Introduction

This study explores the differences in learning motivation and effectiveness between the programming environments of Sonic Pi and Kids Ruby. Both are a simplified version of Ruby, a language which was designed in 1994 with the intention of being easy and enjoyable to use, yet still having powerful performance[11].

Kids Ruby uses turtle graphics as part of their lesson plans[6], following in the footsteps of LOGO, one of the first widely adopted educational languages in schools[9]. Like Logo, one of the aims of Sonic Pi is that learners have permission to explore using their own individual styles whilst creating something concrete[1]. However in its case this is done through allowing the learner to create their own sounds, emphasising the importance of creativity in the learning process and allowing learners immediate concrete feedback during the implementation of their ideas. The element of immediate feedback as applies to live-coding has been discussed as one attractive feature of Sonic Pi in terms of maintaining student engagement[2], however this study focuses more on the effects that learning programming through making music have on students.

There has been a lot of work on developing programming languages which lower the barrier to gaining programming proficiency, so there are various theories about the best types of language to do so, a taxonomy of which has been made for some of the better known educational languages[8]. The main point of agreement and conclusion in the hunt for a more accessible entry point into programming that can be drawn from that study is that a social barrier is in finding a real reason to program, and that the ideal educational language would (domain specific or not) have an application with a broad appeal for the more under-represented groups in computer science.

Even within the current novice programmer demographic, there is no single programmer archetype. The links between different programmer types and the environment that they are working in is an interesting one. It has been shown that in some environments, that levels of *tinkering*, the trial and error of new interactions with a computer interface, shown by the subject can have an effect on their performance on completing tasks[4]. Elsewhere it has been observed in a larger-scale study of novice computer science students that there are three distinct types of programmer, namely *stoppers, movers* and *extreme movers*[7]. Although in these studies a slightly different vocabulary is used to describe programmer behaviour, the underlying thought remains the same; that too much "brainless" interaction with the code produces negative effects

on performance, yet that a certain level is required in order for the subject to learn.

I believe that with Sonic Pi, the simplified syntax and broad appeal of the musical domain are very well-suited for the novice programmer. The interface of Sonic Pi has been specifically designed to make it simple and uncluttered, allowing the user to interact with it using only simple commands, directly linked to a response, removing the need for the normal wrapper commands needed within a typical coding interface. This is so as to allow novices to focus on key concepts for which they can see results initially rather than have to wade through many commands to create something simple. I would like to therefore explore whether the positive effects of this motivation help it achieve its more pedagogic goals, doing so via the teaching and examination of the core principles learned in combination with subject feedback and the tracking of interaction patterns whilst completing tasks.

Both languages are taught through a media-oriented environment, both encouraging learners to learn through exploring their creativity in order to complete the lesson material. It has been widely discussed that there should be more emphasis on the benefits of creativity in learning in schools [3] and it would be interesting to find out whether this is the case for these languages and whether the retention of the concepts learnt was more pronounced in those students who learned them in a more creative manner.

## 2 Research Questions

The aim of this research is to explore the effects the two learning environments have on the novice programmer in terms of motivation and assimilation and retention of information. The main research question is whether there is a significant improvement in the learning of programming concepts when Sonic Pi, rather than Kids Ruby is the teaching environment. This will be explored through the following questions:

- Is there a significant improvement in the learning of programming concepts when Sonic Pi is the teaching environment?
  *- The hypothesis being that since the concept of music being "encoded" (notated) is already so* normal *that the jump to programming it will be less than for example images. This should be explored in terms of measuring the confidence and interaction style with the enviroment of participants whilst performing programming tasks.*
- Is there a link between the amount of tinkering or tweaking of code a novice does whilst learning and their retention of concepts? And is there a higher retention of concepts when learning with Sonic Pi?
  *In terms of the percentage of concepts retained approximately 3 weeks after initial tuition. Each environment will have its own set of keywords, in addition to those common to both; in the delayed recall of the concepts these will be measured and compared through keywords retained.*
- In terms of the motivation whilst learning and engagement with the task, are there interesting traits we can observe in learners of Sonic Pi?
  *This was be measured in terms of the correlation between the users own self efficacy reporting before and after the session, and in their level of interaction (in terms of prolificacy and tinkering) with the code*

## 3 Experimental Method

This study took place in the form of a controlled experiment, where novice programmers were taught and made to perform programming exercises with either Sonic Pi or Kids Ruby. The

analysis made is a between-subject comparison of the two environments, as educational programming tools. The participants were selected to represent an equal distribution of half male, half female and of that half science and half arts first year undergraduate students. 12 unpaid volunteers were used, a number a little under that of the initial study of Logo by Seymour Papert[9]. Ideally a subsequent study would analyse the performance of school children in the same manner as here, since this is the target student of both packages, and could follow from both this study and the testing and material developed during the development of Sonic Pi. Although both the programming environments involved in this study are aimed at school children, the concepts and teaching are still applicable to these older equally inexperienced users, as they are very recent school leavers and are new to programming.

**Tasks**

The participants were initially asked to fill out a questionnaire, which aims to measure how confident they felt about learning a programming language. This questionnaire was a modified replication of the one proposed by Compeau and Higgins 1991 to measure self-efficacy [5]. The questionnaire was presented alongside an explanation of its origin, and the instruction to treat the questions of everyday life like the technology discussed was modern. Only the scale of the original questionnaire was changed, to make it from 1 to 5 rather than 1 to 10 to tie in with the scale on the other questionnaire participants received.

The participants then followed a tutorial sheet (see Appendix A and Appendix B) for a timed period of half an hour before being asked to then carry out a loosely constrained set of tasks in ten minutes, meant to assess their understanding of what they had learnt. These sessions were individually supervised, with the participant aware they could ask for help with any of the tasks. The tutorials consisted of a selection of relevant material from both languages' lesson guides, aimed to teach the concepts of conditionals, random numbers, and iteration. The tutorials were written in the same format, covering the same sets of pedagogic tasks; only the keywords and the descriptions of shape vs. sound differed. The tasks which followed were written and delivered in the same style as the tutorial sheet, and the participants interactions with the software were recorded through capturing the screen using CamStudio[1]. The video of their interaction with the software was then analysed and details such as the number of lines of code typed and alterations made were recorded.

After the participants' interactions with the software were complete, they filled out a second questionnaire about how interesting they found the lessons, how well they felt they had understood the concepts covered in the session and how well they might remember them. They were also asked if they had any creative hobbies, how much programming or mathematical experience they had, and whether they would view what they had just done as creative rather than following a set of instructions.

The participants were asked to provide contact details in case of any questions which might come up about the study, and were contacted approximately three weeks later and were asked to freely try to recall and list any of the commands they had learnt from their programming session. They had no expectation of this second test, which was done in the manner of the Rundus, Loftus and Atkinson[10] in their paper exploring the differing effects of immediate and free recall. The number of keywords recalled were then compared with the total number each participant had been exposed to. Of the 12 students used in the initial study, for this follow-up free recall test, only 10 participated.

[1] http://camstudio.org/

## 4 Results and Analysis

On the examination of the screen recordings from the experiment, the number of lines of code written in total by each participant were counted, as well as how much they were prone to *tinker* with their code. The criteria used to qualify an action as tinkering used here are:

- Tweaking a line already written
  *These tweaks or modifications, if on the same line would be separated by the participant running the program, or changing another line before coming back to the tweaked line to perform a secondary modification.*
- The deletion of a line of code
- The addition of a single line of code within a block
- The movement of a chunk of code (either the copying and pasting, or deletion and re-insertion)

Since the amount of tinkering will be dependent on the amount of code written and the two have a correlation coefficient of greater than 0.5 when compared, the value used to measure it is that of the percentage of lines tinkered with. There is a positive correlation between the quantity of tinkering and the number of lines typed, which was expected, from what intuitively you could expect, and the self-efficacy results discussed in the previously mentioned paper on tinkering [4]. To measure the participants' retention of the concepts learnt, the percentage of keywords recalled was measured. Each correctly remembered keyword was tallied and the inability to remember the specific name, but the convincing description of the function and how it was used would count as half a point.
The results of the experiment are displayed in the table below:

**Table 1.** Participant Results

| Participant | Lines Written | Code Changes | Environment | % Tinkered | % Retained |
|---|---|---|---|---|---|
| 1 | 73 | 19 | S | 26.03 | 56.25 |
| 2 | 33 | 9 | R | 27.27 | 27.27 |
| 3 | 43 | 8 | S | 18.60 | 81.25 |
| 4 | 44 | 7 | S | 15.90 | 85.50 |
| 5 | 53 | 14 | S | 26.41 | - |
| 6 | 47 | 14 | S | 29.78 | 75.00 |
| 7 | 35 | 9 | R | 25.71 | 63.63 |
| 8 | 53 | 32 | S | 60.37 | 87.50 |
| 9 | 32 | 18 | R | 56.25 | - |
| 10 | 38 | 6 | R | 15.79 | 54.54 |
| 11 | 33 | 3 | R | 9.09 | 63.63 |
| 12 | 63 | 15 | R | 23.80 | 72.72 |

**Table 2.** Where the Environment is either SonicPi or KidsRuby, %Tinkered is the percentage of lines tweaked out of the number written and %Retained is the percentage of commands or codewords retained after a period of approximately 3 weeks after the tutorial

From the data collected from the experiment, it is shown that those who learned with Sonic Pi not only typed more, but remember a significantly higher percentage of the commands used, with a p-value of 0.039 in comparison to KidsRuby.

When a comparison of lines of code vs. concepts retained is compared, there is no significant correlation, which cannot confirm the original second hypothesis mentioned in section 2.

Within the media-oriented languages considered here, there was no significant difference in motivation between learners between the two, however within the comments section of the

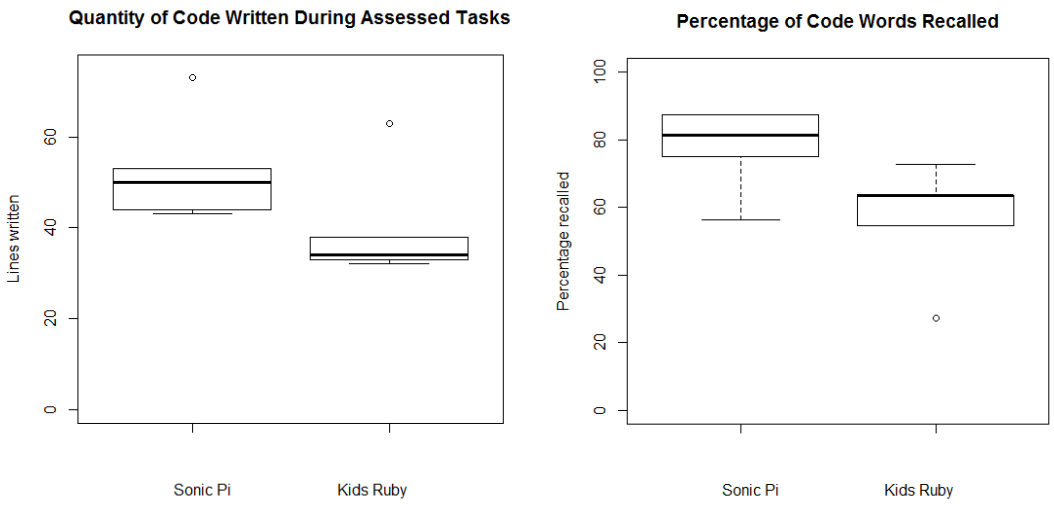


**Fig. 1.** The boxplots show the differences in distribution of lines of code written and percentage of codewords recalled between the two environments; both differences between the performances are significant with p-values of 0.043 for the quantity of code written and 0.039 for the percentage of code words recalled.

questionnaires, a qualitative conclusion which can be drawn is that the motivation levels for these sorts of creative, media-oriented methods of learning programming are much higher than other methods the participants could think of, however since this was not one of the hypotheses, there is no data to support this conjecture, and it would be an interesting parameter to explore in any follow-up experiments.

## 5 Discussion

The results indicate that using the environment of Sonic Pi is preferable to using Kids Ruby due to both a significantly greater ($p < 0.05$) number of codewords retained and lines written by students learning with Sonic Pi. The benefit of more codewords being retained long term is an obvious one. That of quantity of code typed less so, however, intuitively the more a learner is encouraged to interact constructively with a new environment the more they are expected to become familiar with its working.

There was no significant correlation (using Spearman's p correlation coefficient) between tinkering/lines of code written and concept retention; this was one of the hypotheses of the experiment, and it is surprising that there is no link. However, there is a greater level of code written, code tinkered with, and concepts retained in users of Sonic Pi.

There is a significant correlation between the quantity of code written (not just how many lines the resulting program is) and the level of tinkering (number of times code is tweaked over the course of its creation) with Spearman's p correlation coefficient greater than 0.5. This suggests that the fact Sonic Pi promotes more prolific code writing will also encourage greater levels of tinkering (it is also shown in the data that users of Sonic Pi have a significantly higher percentage of tinkering to code written) and this, if considered as in Beckwith et al.[4] as a valuable activity in user interaction with code when used constructively and not to the extreme[2], means the interactions recorded within the Sonic Pi environment can be said to be more beneficial to learning than that of Kids Ruby.

[2] Learners exhibiting this problem are referred to as *extreme movers* in this paper.

The data collected from the questionnaires was disappointing, as the responses were so similar as to be uninteresting (The average level of motivation being 4 on a scale of 1 to 5 with standard deviation of 0.43, with that of confidence being 3.92, s.d. 0.66), however on the whole the responses were that the participants were relatively confident and that the fact that they were learning programming in a creative manner increased their motivation. The other form of feedback which was not measured in this study was the questions participants asked about wanting more programming techniques to allow them to achieve what they wanted to create, both visually and aurally. This would have been interesting to have thought of and tried to measure or record, as during the tutorials I observed that many of the more creatively inclined students wanted to and were guessing how to program these more complicated structures.

## 6 Concluding Remarks

In conclusion, the results indicate that Sonic Pi is the slightly more favourable environment for a novice programmer to learn in as it both encourages more prolific code writing, and tinkering. These in turn promote a better retention of the concepts learnt; and this is shown by the Sonic Pi learners' far better scores in the delayed free recall test. Future experiments might want to measure the difference between Sonic Pi and a non-media-oriented environment, and have a different metric for measuring motivation, as these would be interesting aspects to consider, which were not addressed in this study. It would also be interesting to perform the experiment with a larger group of participants, to explore more subtle correlations, which was not possible within this study.

## References


1. Sam Aaron. Sonic pi http://www.cl.cam.ac.uk/projects/raspberrypi/ sonicpi/, 2013.
2. Samuel Aaron and Alan F. Blackwell. From sonic pi to overtone: Creative musical experiences with domain-specific and functional languages. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, FARM '13, pages 35–46, New York, NY, USA, 2013. ACM.
3. Romina Cachia Anusca Ferrari and Yves Punie. Innovation and Creativity in Education and Training in the EU Member States: Fostering Creative Learning and Supporting Innovative Teaching. Literature review on Innovation and Creativity in E&T in the EU Member States (ICEAC), 2009.
4. Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. Tinkering and gender in end-user programmers' debugging. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 231–240. ACM, 2006.
5. Deborah Compeau and Christopher A. Higgins. Computer self-efficacy: Development of a measure and initial test. *MIS Quarterly*, 19(2):189–211, 1995.
6. The Hybrid Group. Kidsruby http://www.kidsruby.com/, 2011-2014.
7. Matthew C. Jadud. A first look at novice compilation behaviour using bluej. *Computer Science Education*, 15:1–25, 2005.
8. Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137, June 2005.
9. Seymour Papert. An Evaluative Study of Modern Technology in Education or LOGO Memo No. 26, 1975-6.
10. Loftus G. R. Rundus, D. and R. C. Atkinson. Immediate free recall and three-week delayed recognition. *Journal of Verbal Learning and Verbal Behaviour*, 9:684–688, 1970.
11. Mark Slagell Yukihiro Matsumoto, Gogo Kentaro. Ruby User's Guide translation-http://www.rubyist.net/ slagell/ruby/, 1995.

## A KidsRuby Tutorial

In this tutorial, you will learn to program through writing code which makes simple computer graphics in the KidsRuby environment. KidsRuby is a language which is based on the computer programming language Ruby, but has been made simpler for novice learners.

Please complete the various exercises, giving yourself time between each to fiddle around and experiment with the new concepts youve learnt

1. Instructions

Programs operate because they carry out the sequence of instructions they are given in the code which makes up a program.
To set things up for creating computer graphics, KidsRuby needs to be told that this is what the set of instructions are supposed to do. In order to do this, you type the following instructions:

```
Turtle.start do
      |
end
```

All the code that you type from now on should be inside the first line and the end keywords. The package used to make the graphics is called Turtle graphics, which is why the keyword turtle is needed.

Now, within the do and the end keywords, add the following command:

```
Turtle.start do
         background yellow
end
```

When you now click the [start/play] button to run (or execute) the code youve written, you should see the colour change.

Try swapping or adding to the lines of code with the following:

```
background black
  pencolor white
...blue...green...red\newline
```

2. Syntax and order matter

The order and way you write instructions in the code is very important, like words in any language: if you wrote a set of instructions such as a recipe for someone, the order would be important. The spelling is also important; your program needs to give the computer instructions in a language it understands, and it only understands a small, precisely defined set of words.

Try copying this set of commands:

```
Turtle.start do
    background yellow
    turnright 90
    forward 50
    turnright 90
```

```
        forward 50
        turnright 90
        forward 50
        turnright 90
        forward 50
end
```

The commands tell the pen what lines to draw on the background: the numbers specify either how many pixels (the unit which your screen resolution is measured in) the length of the line should be, or by how many degrees the direction of where the line is drawn should change.

Try drawing changing the order of some of the commands to see the different shapes you can make, then try misspelling a word or forgetting the space between the numbers and the words, and see what happens.

3. Loops

A useful thing to be able to do is tell computers to carry out a sequence of instructions multiple times. This saves writing out the same commands a lot of times, like in the previous code example. Programmers do a lot of reading of code, and it is neater to say "do something ten times" than to say "do something" followed by "do something" etc. ten times.

Type the following code and run it, it produces exactly the same thing as in the previous example.

```
  Turtle.start do
  background yellow
  4.times do
        turnright 90
    forward 50
  end
end
```

The highlighted part is the part which defines the loop: you need the do and end in much the same way that you need capital letters and full stops, or that you use parentheses, they show the computer the start and the end of the loop, and therefore which instructions to carry out 4 times. Try changing the number 4 to other numbers, and changing the commands within the loop.

4. Random numbers

Computers can generate random numbers, through the following command:

```
  rand(10)
```

This particular command will generate a number which is zero or larger, but less than ten. (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

Try putting some random numbers into your loop:

```
  Turtle.start do
  background yellow
  4.times do
```

```
        turnright rand(90)
    forward 50 + rand(10)
  end
end\newline
```

This will result in the degree being some random number under 90, and the length of the lines drawn to vary between 50 and 59 (because each time we are adding 0 to 9 on to 50)

5. Conditionals

Sometimes in a program you will not want all of the instructions to be carried out (executed by the computer) you will only want some of them to be, and only in certain situations. This is handled by conditionals, or if-statements.

An if statement looks like this:

```
if A
    do instruction B
else
    do instruction C
end\newline
```

The A here is the condition which will determine which of either B or C will be carried out (performed)

To make an example using something you just learnt, lets make it random. Type the following code and run it:

```
Turtle.start do
  if rand(10) < 5
        turnright 90
        forward 40
        turnright 90
        forward 40
        turnright 90
        forward 40
        turnright 90
        forward 40
      else
        turnright 90
        forward 50
        turnright 60
        forward 50
        turnright 60
        forward 50
      end
end\newline
```

This program is equivalent to saying: if a randomly generated number between zero and ten is less than 5, then draw a square, if it is not, then draw a triangle

Try changing the program a bit, and make some other conditional statements.

6. Nesting commands

To make more complicated programs, a lot of these commands will have to be executed in different combinations. Putting different commands inside other commands is called nesting commands. For example, a loop inside a loop, or a conditional within a loop. The fact that you have been typing all of your commands within the Turtle.start do . end commands is nesting.

Try typing and running the following:

```
  Turtle.start do
  background yellow
  pencolor red
  8.times do
    forward 50
          if rand(2) < 1
      turnright rand(90)
    else
    turnleft rand(90)
    end
  end
end
```

This makes use of all three more complicated programming concepts covered in this tutorial.

Try experimenting and making more crazy shapes with different combinations.

## B Sonic Pi Tutorial

In this tutorial, you will learn to program through writing code which makes simple electronic music in the SonicPi environment. SonicPi is a language which is based on the computer programming language Ruby, but has been made simpler for novice learners.

*Please complete the various exercises, giving yourself time between each to fiddle around and experiment with the new concepts youve learnt*

1. Instructions

Programs operate because they carry out the sequence of instructions they are given in the code which makes up a program.

Type the following commands into SonicPi:

```
play 50
sleep 1
play 60
```

When you now click the green button to run (or execute) the code youve written, you should hear the noises which correspond to the numbers 50 and 60. the sleep command tells the computer to wait for 1 second between carrying out the next play instruction, otherwise theyll be played really close together.

Try changing the numbers to hear the different notes which you can play, and changing or removing the sleep command.

2. Syntax and order matter

The order and way you write instructions in the code is very important, like words in any language: if you wrote a set of instructions such as a recipe for someone, the order would be important. The spelling is also important; your program needs to give the computer instructions in a language it understands, and it only understands a small, precisely defined set of words.

Try typing in some of the following individually and running the program:

```
pllay 50
play50
sleep
```

Now try this longer set of instructions (or commands)

```
play 50
sleep 0.5
play 51
sleep 0.5
play 50
sleep 0.5
play 51
sleep 0.5
play 50
sleep 0.5
```

```
play 51
sleep 0.5
```

3. Loops

A useful thing to be able to do is tell computers to carry out a sequence of instructions multiple times. This saves writing out the same commands a lot of times, like in the previous code example. Programmers do a lot of reading of code, and it is neater to say do something ten times than to say do something followed by do something etc. ten times.

Type the following code and run it, it produces exactly the same thing as in the previous example.

```
3.times do
   play 50
  sleep 0.5
  play 51
  sleep 0.5
end
```

The highlighted part is the part which defines the loop: you need the do and end in much the same way that you need capital letters and full stops, or that you use parentheses, they show the computer the start and the end of the loop, and therefore which instructions to carry out 3 times.

Try changing the number 3 to other numbers, and changing the commands within the loop.

4. Random numbers

Computers can generate random numbers, through the following command:

```
rand(10)
```

This particular command will generate a number which is zero or larger, but less than ten.
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
Try putting some random numbers into your loop:

```
3.times do
   play 50
  sleep 0.5
  play 50 + rand(10)
  sleep 0.5
  play 50
  sleep 1
end
```

This will result in the first play playing a number which is equal to 50 plus a random number of less than 10.

5. Conditionals

Sometimes in a program you will not want all of the instructions to be carried out (executed by the computer) you will only want some of them to be, and only in certain situations. This is handled by conditionals, or if-statements.

An if statement looks like this:

```
if A
    do instruction B
else
    do instruction C
end
```

The A here is the condition which will determine which of either B or C will be carried out (performed)
To make an example using something you just learnt, lets make it random. Type the following code and run it:

```
if rand(10) < 5
    play 50
    sleep 0.5
    play 50
    play 52
    sleep 0.5
    play 50
  else
    play 60
    sleep 1
    play 55
    sleep 0.1
    play 55
end
```

This program is equivalent to saying: if a randomly generated number between zero and ten is less than 5, then play one sequence of notes, and if not, then play a different one.

Try changing the program a bit, and make some other conditional statements.

6. Nesting commands

To make more complicated programs, a lot of these commands will have to be executed in different combinations. Putting different commands inside other commands is called nesting commands. For example, a loop inside a loop, or a conditional within a loop.

Try typing and running the following:

```
 3.times do
    play 50
   sleep 0.5
if rand(10) < 5
    play 50
    sleep 0.1
       play 50
  else
    play 57
    sleep 0.5
    play 50
  end
  sleep 0.5
```

```
  play 60
  sleep 0.5
  play 50
  sleep 1
end
```

This makes use of all three more complicated programming concepts covered in this tutorial.

Try experimenting and making more crazy sounds with different combinations.