INTRODUCTION

It's necessary to be able to understand the code in my Rojak pieces to be able to get the most out of looking at their scores. Originally, I planned for the annotations found in the scores to address a reader's lack of familiarity with code, but as I annotated a few of them, I found myself needing to repeatedly explain the basics. Reading code is a pair of skills: it requires an understanding of computing generally and an understanding of the programming language specifically used. My goal here is to introduce enough of the basics of each to make reading my scores possible. Then, annotations in the score can build off of this basic knowledge, where necessary.

There are many programming languages. They form families. Knowing one helps in understanding a number of them within the same family. But there are several paradigms in programming. These have names like imperative, functional, and object-oriented, which you may have heard of in passing at some time. In essence, a paradigm is a way of organizing information and carrying out procedures in code. It can be hard to shift paradigms. If you are used to solving coding problems in a functional language, you will be often lost and frustrated in an object-oriented language, and vice versa.

The Rojak pieces are written for the Sonic Pi environment, which itself is built on top of Ruby, a general purpose language. Ruby is an object-oriented language that is very cool and very expressive. I'm glad Sonic Pi is build on it. It makes coding fun, pretty fast, and quite easy to read. This last point you'll just have to trust me on for the time being, but by the end of this introduction perhaps you'll agree. One advantage to learning Sonic Pi, then, is that one is (more or less) actually learning Ruby, an elegant and powerful language.

What is Sonic Pi? The developers call it "a code-based music creation and performance tool." When you launch it and first look at it, you think, "This is a programming language." So, what's the difference? Sonic Pi is technically a domain-specific language, that is, a programming language that is designed to be used for a specific set of tasks. In our case those tasks are musical. For instance, we use code to name and store rhythms and melodies, and then use a bit more code to call this stored information up when we want to use it and play it back with a built-in synthesizer. Code is a nice way to deal with musical material. It can be easily manipulated. For instance, we transpose melodies by just adding or subtracting a value to the original pitch value. We speed up a rhythm by dividing the original duration values by a value greater than 1 (or multiplying it by a value less than 1 but greater than 0). Believe it or not, you could use Sonic Pi as a substitute for Logic Pro, Ableton Live, Pro Tools, Cubase, etc. Those DAWs all have pleasant, easy-to-use interfaces, but behind the scenes they are just doing what Sonic Pi does, calling up data about audio files and MIDI values at a certain time and playing

them back. So, it really is accurate to call Sonic Pi "a code-based creation and performance tool." But while Sonic Pi can roughly emulate Ableton Live, there is still a lot you can do in Ableton - or, at least, can do much more simply - that you can't do in Sonic Pi. However, the reverse is true as well. There's a lot in Sonic Pi that is very easy to do that is just impossible to achieve in a DAW. And, in case you were wondering, this is why I don't do my pieces in Ableton, as much as I use that environment for other tasks where it is more suitable. I really appreciate the closeness between thinking, coding, and executing musical design in Sonic Pi (or any of the other terrific environments for computer music like CSound and Supercollider). It is a place to compose rather than arrange and mix.

COMMENTING

When coding, it is important to put in comments so that readers and other coders understand your code. However, the computer needs a way to differentiate between code and comment. Comments must always be ignored by the computer on execution of the program. Different programming languages have difference conventions. In Ruby, the hashtag (#) is used. Anything after a # is interpreted as a comment.

```
# I'm a comment
bread = "wonder" # Before this hashtag is some code, after a
comment.
```

I can write comment haikus:

```
# the moon at 12 noon
# I wonder if this is caused
# by global warming
```

And longer comments can be sectioned off with =begin and =end.

```
=begin
my dog whines at night
does she need to pee so much
or is she lonely
=end
```

ASSIGNMENT

In coding, it's important to name data so that it can be referenced later. Below are two examples:

```
SCREEN_WIDTH = 480
current_melody = [60,64,66,59,72,70]
```

The first example `SCREEN_WIDTH` demonstrates a constant, the second - `current_melody` - a variable. The screen width of your device is not going to change while you use a program. Therefore, it can remain constant for the duration that the program runs. But the current melody could very reasonably change. In the first section of a piece, it could be what is given above, but in the second section, it could be replaced with a different set of values to represent the second section's melody. Constants are written in all caps to distinguish them from variables.

I need to admit that I only use variables, even when something is technically a constant. In reality, the code for my pieces isn't very long - perhaps a few hundred lines - so I don't feel a need to make sure this distinction is given. If you look at my scores and say, "Isn't that a constant? Why isn't it all caps?", then know the answer is because I'm just being lazy.

More generally, these examples demonstrate assignment. Whenever you need the value 480, you can just type in `SCREEN_WIDTH`. For example:

```
SCREEN_WIDTH = 480
hello_kitty = 10
print SCREEN_WIDTH * hello_kitty # 4800 printed to console log
```

If we execute the code above, the result of the mathematical operation will be 4800, and it will be printed in the console log, a window that gives feedback on code execution as it is running.

There is something really important to understand in the above code. There is nothing "screen-widthian" about `SCREEN_WIDTH` (or "hello-kittyian" about `hello_kitty`). `SCREEN_WIDTH` is just a bunch of characters used to represent something else, in our case the integer 480. The computer doesn't know that it has anything to do with the screen width.

Every programming language has its syntactical conventions for assignment. These aren't always required but are considered good practice so that others can read your code more easily. With assignment in Javascript, you use camel case (`theNextVariable`) - yes, it's really called camel case, I didn't just make that up - in Lisp you use hyphens (`the-next-variable`), and in Ruby you use underscores (`the_next_variable`). Traditionally, constants use all caps (`SCREEN_WIDTH`), but this isn't a consistent convention between languages. When you learn a new programming language, you hate the conventions, you think they look ugly, and then you get over it and just use them.

DATATYPES

In the two initial examples of assignment one value was an integer and the other was a list of values in brackets. These represent 2 datatypes commonly used in programming. The first is called an integer (obviously); the second is called an array. An array is just a list of values separated by commas and surrounded by brackets. It can contain any datatype, even other arrays. A few examples are given below:

```
[1,2,3]
[a,b,c]
[dog,65,"purple",[4,4,3,2,d,s]]
```

There are a few other datatypes. For our purposes you should know:

• floating-point values (or floats), which are decimal numbers like 3.2443;
• strings, which are characters surrounded by quotation marks, such as "happy" or "lucky777";
• Booleans, which are the values true and false;

Each datatype has different properties and, hence, we use them for different tasks. Having datatypes is important also because they let us check the nature of the data and confirm it is the kind we want. Imagine you fill out a webform that asks for your phone number and you type FD#-HU7^. The computer would check to make sure that each value entered is an integer, and if not, it will tell you that you made a mistake.

Note that purple and "purple" aren't the same thing. For example:

```
purple = [128,0,128]

purple.length # returns 3
purple[1] # returns 0
"purple".length # returns 6
"purple"[1] # returns "u"
```

In the example above, `purple` is a variable name and `"purple"` is a string. If we ask for the length of `purple`, the computer returns 3 because there are 3 values in the array; asking for the length of `"purple"` returns 6 because there are 6 letters in the word purple. We can get the nth value of an array or string by using [n] after the variable or data. If we ask for the value at index 1 - which is the 2nd value, not the first - you get 0 with purple and "u" with "purple".[1]

[1] Like almost all programming languages, Ruby uses zero-based indexing. This means that the first value in any group of things is at the 0-th position. [1,2,3][0] returns 1. [1,2,3][1] returns 2. Similarly, "cat"[0] returns "c" and "cat"[1] returns "a".

You might think that 2 and 2.31 are just numbers and could be treated the same, but as shown above, sometimes you really need an integer, like with digits in a telephone number. Another example: when asking for the 2nd value in the array [32,43,12,64] versus asking for the 2.31st value of that array. The second request would cause an error. With MIDI there is middle C (60) and C# (61) and D (62). There is no 60.31. (Sadly, MIDI can't do microtonality, although Sonic Pi synths can!) So, datatypes and datatype checking help reduce mistakes in code and errors in usage and help programs function correctly.

Note this isn't an exhaustive list of datatypes in Ruby, just the ones we'll encounter mostly in my pieces.

In Sonic Pi there is another "datatype" called a ring. It's not really a datatype. Effectively, a ring is just an array that can be read through cyclically. Normally, when we use arrays, we read through each value to utilize it in an iterative algorithm, applying the same algorithm to each value in the array. When the computer reaches the end of the array, it terminates the iteration. With a ring, it returns to the beginning, cycling until requested to stop. It is very good that this feature doesn't exist in general purpose programming languages. It would cause a lot of bugs because you would end up with arrays accidentally being continuously read through. This would cause stack overload and crash your program. However, it's very good that it is a feature in Sonic Pi, where speed of code execution is more controlled and we often need to cyclically read through a series of array values that represent a rhythm or melody. I don't explicitly use rings in my code because all arrays are converted to rings during execution, if necessary. So, you'll never (or very rarely) see them in the Rojak pieces. But if you come from a programming background, then you might be confused as to why the programs work even though the arrays aren't long enough to support them. The reason is because arrays convert automatically to rings during execution, when necessary.

OBJECTS

Ruby is an object-oriented language. This means that the main organizational strategy in a Ruby program is through the use of objects generated from templates called classes. That was a bit of a mouthful, so let's have an example. Imagine you own a pet shop and want to have a website listing all of your pets available for sale. You might create a set of classes called dog, cat, and hamster. These would be templates that define generically the

information you want to store for each animal. You will want to record a set of characteristics, or what in programming is called properties.[2]

```
class Dog

    name = ""   # these are your properties.
    color = ""  # each is given a default value
    weight = 0  # when instantiated, these default values
    age = 0     # are usually replaced

end
```

To define a class, you name it - in our example, it will be named Dog - and then define its properties. The word `end` defines what is called a block in Ruby. Here we have a class block.

You can now create a Dog object for each dog in the shop. Let's make one for Bobo and update its properties.

```
Bobo = Dog.new
Bobo.name = "Bobo"
Bobo.color = "brown"
Bobo.weight = 15
Bobo.age = 2
```

Then, we post on our site the following: (Note that #{} allows us to convert any datatype to a string so that it can be printed as text.)

```
"We have a new dog in our shop named #{Bobo.name}. He's a lovely
#{Bobo.color} color and is medium sized at #{Bobo.weight}kg. He's
just #{Bobo.age} years old!"
```

If someone searched your website for cats, then Bobo wouldn't show up in the search results because you would have programmed your site to filter for objects of only class Cat, and Bobo is an instance of class Dog.

If this was a game with virtual dogs, then we'd want these dogs to be able to do actions. In a class, you do this by defining methods (or what are also called functions in programming).

---

[2] The class examples are not accurate Ruby syntax. I simplify to avoid needing to explain unnecessary details. You will never encounter class definitions in my pieces but it's helpful to understand some basics about classes in order to understand what objects are.

```
class Dog

    name = ""   # these are your properties.
    color = ""  # each is given a default value
    weight = 0  # when instantiated, these default values
    age = 0     # are usually replaced

    def bark # the bark method
      print "Woof!!"
    end

    def growl # the growl method
      print "grrrrr..."
    end
end
```

Here we've defined two methods called bark and growl that print some text to the console log. If we had a virtual dog named Bobo, we could write `Bobo.growl` and "grrrrr..." would be printed to the console log.

Yes, this is a silly example, but it demonstrates how classes are used to organize and categorize data in a program and that classes are made up of properties and methods. You also can see what is called dot notation. Object_Name.Method is how we call a method on - or access a property of - a particular object.

Ruby is one of the most thoroughly object-oriented programming languages around. All of its datatypes are objects. If you were paying close attention, you would have noticed that the dot notation of objects was used in the purple/"purple" example. The length method exists for both the array and the string datatypes. You'll see a lot of this dot notation in my scores because it makes it very easy to process and manipulate arrays.

In most object-oriented languages, arrays and strings are treated like objects, but it's rare to see integers and floats treated this way. But these datatypes are also objects in Ruby, and this is really great, especially for music. For example:

```
4.times do
  # do something here
end
```

Here we call the `.times` method on the integer 4, using that to repeatedly execute the code within the do and end block 4 times. You can probably see how useful this would be with music.

Your techno hi-hats on every 16th notes:

```
16.times do
  # play hi-hat sound for the length of a 1/16 note
end
```

Four on the floor:

```
4.times do
  # play kick drum sound for the length of 1/4 note
end
```

You can nest these. For example, 4 bars of four on the floor:

```
4.times do
  4.times do
    # play kick drum sound for the length of 1/4 note
  end
end
```

IMPORTANT BUILT-IN FUNCTIONS

But how do we actual make sound in Sonic Pi? For instance, how do we get that kick drum to sound? There are 2 types of sound making in Sonic Pi (as well as in a DAW). There are synthesizers and there are audio files. To play a synthesizer sound, one simply types "play" along with the MIDI pitch value.

```
play 60 # plays middle C on the default synthesizer
```

You can define the synth used, adjust its envelop, amplitude, perform basic filtering on it, etc. I'm skipping over these features since I don't use them. I work primarily with samples. In my code you'll largely see the word "sample" used. This triggers playback of a sample.

```
sample :bd_haus # playback the built-in sample called bd_haus
```

I use a lot of samples that I've personally made. So, I need to first tell Sonic Pi where those files are:

```
mySamples = "/Desktop/FrogSamples/"
```

This creates a variable that points to the folder called FrogSamples on my desktop. This is usually placed at the very top of the score. Then, I can call up the audio files within it using integers, where 0 is equal to the first file, and 1 is equal to the second, etc.

```
sample mySamples, 3
```

This plays back the 4th sample in the FrogSamples folder.

Unless otherwise told, computers process information as fast as possible. So, in computer music, we need to always tell the computer how long to wait before continuing.

```
sample mySamples, 2
sample mySamples, 5
```

In the code above, it looks like the 3rd sample in the FrogSamples folder is played and then this is followed by the 6th sample. (Remember that the 1st sample is represented by 0.) This is technically true, but because the computer executes its tasks so quickly, we will hear these two samples played simultaneously. We need to tell it to wait after the first sample is triggered.

```
sample mySamples, 2
wait 3
sample mySamples, 5
```

Here we wait 3 beats before triggering the `sample mySamples, 5` code. For coder folks reading this: yes, it is beats, not seconds. Sonic Pi defaults to 60 BPM, but this can be changed.

```
use_bpm 60

sample mySamples, 2
wait 3
sample mySamples, 5

use_bpm 108

sample mySamples, 2
wait 3
sample mySamples, 5
```

In the code above, the first `wait 3` will pause for 3 seconds, the second `wait 3` will pause for only 1.67 seconds. For those who have learned some Sonic Pi in the past, you might have learned to use the command sleep instead of wait. Sleep is more standard but I prefer wait as a term. Both functions do the same thing.

RANDOMNESS

Harnessing randomness is part of what I do in my work. It is a common tool in the history of computer music. Randomness can get a bad rap. It's random, therefore, disorganized, right? Music is organized, and composers make decisions about what

happens when in a piece. Aren't you abdicating your responsibility as a composer? First, different forms of randomness are random in different ways. There are random walks; different distributions like Uniform, Poisson, Gaussian, Weibull, Geometric; Markov chains; and weighted probability to name some of the most common. That means, randomness has behavior. The choice to use randomness as well as the type of randomness used can have musical intention. It certainly does in my work.

Interestingly, randomness in a computer isn't actually random. Computers follow instructions, and those instructions can't be as simple as "Choose 10 random numbers between 50 and 75." Although the code may state this, what is happening behind the scenes is an algorithm that simulates randomness. We call this pseudo-randomness. The formula for this algorithm is the following:

$$X_{n+1} = (aX_n + c) \bmod m$$

where X_{n+1} is a sequence of values and

m, $0 < m$
a, $0 < a < m$
c, $0 \leq a < m$
X_0, $0 \leq X_0 < m$

If you're a musician, then the above equation probably looks a bit scary to you. Let me explain a bit and then we can look at an example. This equation has 4 components. a is the multiplier, c is the increment, m is the modulus, X_n is a seed value. The multiplier a just multiplies its value with the current seed value. Added to this is an incrementing value c. Whatever value results is divided by the modulus m. If the result of $aX_n + c$ is 15 and m is 7, then 15 ÷ 7 is executed. This yields 2, remainder 1. With a modulus we are only interested in the remainder. So, the value returned by the modulus function is 1. This value, then, becomes the new X_n and we run the equation again. Below is an example:

```
# a = 1, Xn = 3, c = 2, m = 7

(1 * 3) + 2 = 5 (mod 7) # yields 5
```

Then, we repeat this formula with all of the same values except X_n, which is replaced with the result of the previous equation.

```
(1 * 5) + 2 = 7 (mod 7) # yields 0
(1 * 0) + 2 = 2 (mod 7) # yields 2
(1 * 2) + 2 = 4 (mod 7) # yields 4
```

As you can see, this formula is used recursively, that is, it is continually rerun using data from previous calls. It stops when it meets a certain condition. In the case of this formula, the condition would be a predetermined length of the sequence. If we requested 5 values with this algorithm, then the computer would stop this procedure at this point and return [3,5,0,2,4].

This example is a simplification though. In real life use, the values for *a* and *m* must be much bigger. The modulus is usually a factor of 2, like 2^{32}, and the multiplier is usually a very large prime number. This formula is called the Linear Congruential Generator.

Maybe it seems I'm in the weeds right now in explaining how a linear congruential generator works, but I figured it is worth explaining this in more depth for 2 reasons. First, I use the linear congruential generator constantly in the "wrong" way - with small values - to create patterns rather than randomness. So, in my code, you'll see references to LCG and the values inputted to get the results of a certain array. Second, I wanted to show that if you fix the seed you will always get the same result because the computer is just executing a formula recursively. This is a really useful compositional tool. Run some code that includes randomness with a seed value of 100. Play it back. Don't like it? Change the seed to 123 or 154 or 72 until the result is what you like.[3] Then, any time you want to have that passage play exactly as you like it, just set the seed to the correct value before you execute that bit of code. In Sonic Pi, you achieve this with the built-in function `use_random_seed`.

```
use_random_seed 154
```

There are some other common built-in functions related to randomness worth mentioning here before moving on to the next topic.

```
rand(50) # return a float between 0 and 50
rand_i 50 # return an integer between 0 and 49
rrand(20,50) # return a float between 20 and 50
rrand_i 20,50 # return an integer between 20 and 50
```

Take note of the different notations used above. In `rand` and `rrand`, I used parentheses around the arguments; with `rand_i` and `rrand_i` I didn't. Both are permissible notations. I prefer to use parentheses, but sometimes I don't use them. Just understand that the different notations don't cause different results. This is the case in a number of syntactical aspects in Ruby. There is often more than one way to write the same code.

---

[3] Note that the values for *a*, *c*, and *m* are predetermined by the programming language and don't need to be inputted.

ARRAY MANIPULATION

Since arrays are objects, they also have methods. It is with array methods that one can very clearly and simply manipulate array data. Let's take the randomly generated array from above and manipulate it further to demonstrate some of the most common and useful methods.

```
rand_result = [3,5,0,2,4]
# a randomly chosen values from the array, e.g., 2
rand_result.choose
# return in a random permutation, e.g., [5,0,4,2,3]
rand_return.shuffle
rand_return.reverse => [4,2,0,5,3] # pretty obvious what it does
# return only the first 3 values of the array
rand_return.take(3) => [3,5,0]
# return only the last 3 values of the array
rand_return.last(3) => [0,2,4]
```

They can also be used in series:

```
# take the last 3 values of the array and reverse their order.
rand_return.last(3).reverse => [4,2,0]
```

Their names are usually very descriptive and you can figure out what's happening to the data quite easily.

ITERATION

The other important use of arrays is iteration. Imagine we want to add 1 to every value in our randomly generated array and have each result printed to the console log. That's pretty easy to do.

```
rand_result = [3,5,0,2,4]
rand_result.each do |n|
    print n + 1
end
```

Here we use the `.each` method. For each value in the array, the computer takes it, assigns it to the variable n - that's indicated with `|n|` - and uses it in the program written between the do/end block. Here we just add 1 to n and print it to the console log.

How could you use this in music?

```
[3,5,0,2,4].each do |n|
    sample mySamples, n
```

```
    wait 0.5
end
```

In this example, we iterate through each value in the array and use that value to select samples in the mySamples path (in the FrogSamples folder, remember?). Then, we wait half a beat until moving on to the next value in the array.

We can combine some of the things from earlier to make a small passage of music that repeats. In this case, we will use the array [3,5,0,2,4] as values added to 60 (middle C) to get a 5-note melody

```
[3,5,0,2,4].each do |n|
    play 60 + n # We will have the notes 63, 65, 60, 62, 64.
    wait 0.5
end
```

Let's repeat this 4 times, updating the array so that every value has 1 added to it - a half-step transposition - each repetition.

```
rand_result = [3,5,0,2,4] # define a variable rand_result

4.times do

    rand_result.each do |n|
      play 60 + n
      wait 0.5
    end

    # update rand_result to a new array
    rand_result = rand_result.collect do |n|
      n + 1 # add 1 to each value from the original rand_result
    end

end
```

Note that the original `rand_result` variable is defined outside of the `4.times` block. That's important. If it was within the block at the top, then `rand_result` would reset to the original values with each repetition and we would never hear the transposition. Also note the method used here to update `rand_result` is called `.collect`. With `.each`, the result of the procedure on each n value is outputted at the end of the procedure; with `.collect`, these results are collected and only outputted as an array when there are no more values in the array to process and the iteration is terminated.

```
[3,5,0,2,4].each do |n|
  print n + 1
end
```

With this code, we would see the following in the console log:

```
4
6
1
3
5
```

But with this code, which uses collect instead of each -

```
[3,5,0,2,4].collect do |n|
  print n + 1
end
```

- we would see this in the console log:

```
[4,6,1,3,5]
```

Hence, we are updating the current `rand_result` with a new array that takes the current version and adds 1 to each value. Once that is complete, the whole process is repeated until it has been repeated 4 times, at which point the process terminates.

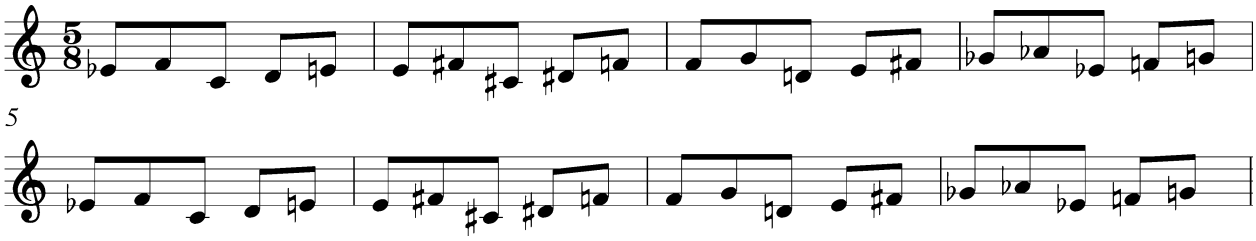
Let's go even one step further. Let's repeat this process twice.

```
2.times do

  rand_result = [3,5,0,2,4]

  4.times do

    rand_result.each do |n|
      play 60 + n
      wait 0.5
    end

    rand_result = rand_result.collect do |n|
      n + 1
    end

  end

end
```

Note that `rand_result` is declared as a variable with [3,5,0,2,4] within the `2.times` block but outside of the `4.times` block. When the `4.times` block is completed, `rand_result` will have been set to [7,9,4,6,8]. This needs to be reset. So, when the

`2.times` block is invoked again, the `rand_result` is reset to [3,5,0,2,4] - that's why it's inside the `2.times` block - and we get a repetition of the chromatically ascending 5-note pattern.

In music, it would look like this:



As mentioned in the first footnote, Ruby, like almost all programming languages, uses zero-based indexing. The first value in an array is the 0th value, that is, it is at index 0. In the array [3,5,0,2,4], 3 is at the 0th index and 4 is at the 4th index. To access a value at a particular index, we just put the index value in brackets after the variable or data.

```
[3,5,0,2,4][1] => 5
nums = [3,5,0,2,4]
nums[2] => 0
```

We can iterate through an array and access the index value for each array value using .each_index. For example:

```
[3,5,0,2,4].each_index do |i|
  print i
end
```

This will return the following in the console log:

```
0
1
2
3
4
```

Finally, a Sonic Pi original is the `.tick` method. This is like `.each`, but it allows us to tick through array values only when they are requested in a program, and it automatically converts arrays to rings, as mentioned previously, so that they can be indefinitely cycled through, which is good if you have 5 rhythmic values and 6 pitch values and want to repeat them until they realign, as below:

```
notes = [60,62,63,65,67,68]
rhys = [1,1,0.5,0.5,1]

30.times do
  play notes.tick(:n)
  wait rhys.tick(:r)
end
```

Note that each tick needs a name placed within parantheses. This should be a symbol, which is basically a variable that start with a colon. We repeat 30 times so that the array of 5 values repeats 6 times total and the array of 6 values repeats 5 times total. Doing so, we arrive at the point where these two components would resynchronize.

CONDITIONALS

As a program runs, it's important for the program to be able to make decisions based on given rules. These are called conditions. You can imagine them as driving instructions. If there's no traffic in your lane but a lot in the next lane, then stay in your lane. If there's a lot of traffic in your lane but not much in the next one, switch to the next lane. But if both lanes have a lot of traffic, then exit the highway. Or in code:

```
if your_lane == not_much_traffic && other_lane == lots_of_traffic
  stay_in_lane
elsif your_lane == lots_of_traffic && other_lane == not_much_traffic
  change_to_next_lane
else
  exit_highway
end
```

Note that == is not =. = is used for assignment while == means "equal to". && means "and" (and || means "or", which is not used here but is also important to know). In our example, we are checking the status of both lanes, so we need &&. Conditions use the Boolean datatype. When you write:

```
if my_age == 48
```

that means "if it is true that `my_age` equals 48". So, the computer is waiting for a true or false answer to the statement "`my_age` is equal to 48". The traffic example had 3 states. It used `if`, `elsif` (else if), and `else`. `Else` isn't a condition. It is just the last option. It really means "if none of the above is true, then do this." It's also possible to just have `if` and `else`, that is, only 2 options.

```
if next_note == 65
  play 65
```

```
else
  play 72
end
```

In this example, the computer checks the value of an incoming note called `next_note`. If it equals 65 (an F), then that note is played; otherwise, the note 72 (a C) is played.

Sometimes you have more than 2 or 3 choices. In this scenario, you use a `case` statement (that's the name in Ruby but is maybe more commonly known as a switch statement).

```
panning = 0 # set the panning variable to 0

case next_note
when 60 # if middle C
  panning = -0.5
when 62 # if D above middle C
  panning = -0.25
when 64 # if E above middle C
  panning = 0
when 65 # if F above middle C
  panning = 0.25
when 67 # if G above middle C
  panning = 0.5
else # if none of those previous values
  panning = rrand(-1.0,1.0)
end
```

In the example above, the value of `next_note` determines the panning. (Panning in Sonic Pi is expressed between -1.0, only from the left speaker, to 1.0, only from the right speaker.) If `next_note` is C, D, E, F, or G (starting at middle C), then `panning` is set according to the values given for each case. Otherwise, panning for the next note is determined randomly. If `next_note` equalled 66, then the computer would check all of the cases, see that none of them match, and execute the code given under `else`.

MAKING FUNCTIONS

All programming languages come with built-in functions. Math operators are a good example. Every programming language has functions to add, subtract, multiple, and divide. Every language has functions to manipulate arrays. Every language has functions to print to the console. As we've seen in this introduction, Sonic Pi is a domain-specific language. It is built on top of the general purpose language Ruby and has additional functionality that serves electronic music purposes. But as your tasks become more specific in your program, you usually need additional functionality. No programming environment can predict exactly what every person will need for their individual tasks.

Hence, it is possible to create functions. In Sonic Pi this is achieved with `define`. For example, a (very silly) function that adds 3 to any number it receives.

```
define :add3 do |n|
  return n + 3
end
```

After the word define, we name the function with a symbol (so it must start with a colon). Then, we create a block by using `do` and `end`. After the do, we declare any variables to be used in the function. In this case, we will have one, it will be called `n`. Then, we code our program within the block. In this example, the function will return the value of `n + 3`. Examples of this in use:

```
add3(3) # returns 6
add3 5 # returns 8
```

Note that the variable `n` isn't technically called a variable. When defining a function, it's called an argument. A function can have a bunch of arguments. They need to be supplied when the function is invoked, otherwise the code won't run. (Ok, actually it's more complex than this. Function arguments can be made optional or be given default values, but that's for Computer Science 102, not Computer Science 101, so we aren't covering it.)

The decision to make a function is similar to the decision to make a constant. If you are going to repeatedly need a certain algorithm to process data, then you make it a function (or a method in a class) rather than cut-and-paste that bit of code for the function all over your program. Coding is really a process involving simplification and clarification, a balancing act between design and expression. This is technically called refactoring. Tools like assignment and the ability to design classes and functions make this possible.

We can use a function to play sound as well. In my piece *Giraffe Cup*, I wrote a function called `giraffeKick` that plays back a bunch of samples simultaneously to recreate the sound of a kick drum. Below is an example in which a melody and rhythm is played back repeatedly until they resynchronize.

```
define :isorhythm do |mel,rhy|

  # find the LCM between the lengths of the mel and rhy arrays
  num_reps = mel.length.lcm(rhy.length)

  # use the LCM from above as the number of repetitions
  num_reps.times do
    play mel.tick(:n)
```

```
    wait rhy.tick(:r)
  end

end
```

This function takes two arrays, finds the least common multiple between the lengths of these two arrays and assigns that value to the variable `num_reps`. Then, `num_reps`, an integer, is used with the `.times` method to repeat the `mel` and `rhy` cycles until they resynchronize.

To invoke the function and have the music play back, you write something like:

```
isorhythm([60,62,63,65,67,68],[1,1,0.5,0.5,1])
```

In this example, since the first array has 6 values and the second has 5 values, the LCM is 30. So, the function would play 30 notes, cycling through the values in each array, and then terminating at a time when they would resynchronize.

This example is built from an earlier one, which you may recall. Here the function allows us to work more abstractly with the code's underlying compositional principle called isorhythm. The computer, rather than the programmer, calculates the number of repetitions for any pair of pitch and rhythm values it receives and executes the musical passage. If you were writing a piece built largely from different isorhythms - perhaps generating hundreds of them in various layers over the course of 10 minutes - then you would want to use this function rather than copy-and-paste the earlier code, individually calculate the LCM, and substitute in the updated values for each isorhythm. By doing so, you would also be expressing the design of the composition more clearly.

CONCLUSION

What is offered above is not a complete introduction to computing or to Sonic Pi. But this introduction should give you a good basis for understanding the code I use in my Rojak pieces. Originally, I decided to write this introduction because I was uncertain of what assumptions in knowledge to make when annotating my Rojak scores. This introduction doesn't exhaust what I use in my work, but does cover what is most commonly used. And it gives me clarity on the knowledge I can assume from those looking at the scores. When anything is used in my pieces that is not address in this introduction, I know I need to explain it in more depth. I hope this introduction gives readers more confidence to explore my scores and see what is inside the music's design.