DESIGNING ART, MUSIC AND ANIMATIONS

Glen Bull, Jo Watts and N. Rich Nguyen
DESIGNING ART,
MUSIC, AND ANIMATIONS

Editors
Glen Bull, Jo Watts, and N. Rich Nguyen

Technical Editor
Gina Bull

Art Editor
Alexis Kellam

Published by
Association for Advancement of Computers in Education (AACE)

Copyright © 2021 by Glen L. Bull

No part of this publication may be reproduced in whole or in part, or stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission of the author. For information regarding permission, contact Glen L. Bull, School of Education and Human Development, University of Virginia, Charlottesville, Virginia.
Acknowledgements

*Designing Art, Music, and Animations* stems from a collaboration between the Department of Computer Science, the Department of Music, and the School of Education and Human Development at the University of Virginia. Modules with instructional videos and resources that support topics in the book were originally developed with support from a National Science Foundation grant (DRL-1842342, Glen Bull, P.I.). These resources are available on the *Make to Learn* web site at:

https://www.maketolearn.org/creating-art-animations-and-music/

To support this work, a programming environment, *TuneScope*, was developed that extends the music creation capabilities of *Snap!* through addition of sampled musical instruments such as pianos, guitars, and drums. Rich Nguyen, an assistant professor of Machine Learning in the Department of Computer Science, provided oversight for program development. Eric Stein, a computer science student working under the direction of Rich Nguyen, is the lead developer for *TuneScope*. A second computer science student, Harsh Padye, participated in the project, developing extensions supporting capture and recording of input from MIDI keyboards among other contributions.

The art chapters were developed by Glen Bull, Jo Watts, and Alexis Kellam. Jo Watts, manager of the *Make to Learn Laboratory*, coordinated the overall project. Alexis Kellam, a student in the School of Education and Human Development, served as art editor and provided illustrations for the book. She also provided leadership for development of instructional videos in support of the project.

The music chapters were developed by Glen Bull, Jo Watts, Luke Dahl, and Rachel Kim-Shao Gibson. Luke Dahl is an associate professor of Music Composition and Music Technology. Rachel Gibson’s undergraduate degree in Technology in Music and Related Arts (TIMARA) is from the Oberlin Conservatory of Music. She currently is enrolled in the instructional technology graduate program in the School of Education and Human Development. We would like to acknowledge and thank the sculptor, Bathsheba Grossman, and the illustrator, Peter Reynolds, for their contributions to the effort.

Most of all, we would like to acknowledge an enormous debt to Brian Harvey and Jens Möenig for creation of the computing environment, *Snap!*, that makes our work possible. We appreciate their support and encouragement throughout this effort and thank them for the many questions that they answered. Brian and Jens created a world and a community that has facilitated and enabled creative work by many others, including this working group.
Preface

This book was developed to accompany a parallel course, *Designing Art, Music, and Animations*. The goal of the course is to provide an introduction to creativity at the intersection of digital technology and the arts. It is the third iteration of this sequence. Based on pilot implementations, the course and accompanying text have been substantially revised. In the current implementation, there are five chapters that focus on the arts and five subsequent chapters that focus on music.

Each of five arts chapters provides an introduction to a different artist – the sculptors Alexander Calder and Bathsheba Grossman, the nineteenth century post-impressionist artist, Georges Seurat, and the twentieth century artists Mark Rothko and Jason Pollock. Each of these artists works in a different style and in different mediums. These styles provide opportunities to emulate these artistic approaches through the medium of digital technologies.

The five music chapters culminate in development of a code-based Digital Audio Workstation (DAW). These chapters introduce individual musical notes, musical scales, musical chords, drum tracks, and lyrics and vocals.

The course and accompanying book is a collaboration among the School of Education and Human Development, the Department of Computer Science, and the Department of Music at the University of Virginia. The course is approved as an elective in the Bachelor of Arts in Computer Science program at the University of Virginia. The primary objective is to facilitate creativity and exploration of the arts through coding.

The educational computing language *Snap!* is used as the foundation for the course. This computing language was developed at the University of California, Berkeley, with support from the National Science Foundation and is used in undergraduate courses for non-computer science majors. It was jointly developed by Brian Harvey and Jens Möenig with the goal of making advanced computational concepts accessible to nonprogrammers.

TuneScope, developed to support the music portion of the course, is an extension of *Snap!* that incorporates additional music commands. TuneScope uses the W3C Web Audio application interface (API) to extend the music capabilities of *Snap!*.

Brian Harvey notes, “Languages in the *Logo* family, including *Scratch* and *Snap!*, take the position that our mission is to bring programming to the masses.” We share that goal.
# Table of Contents

Preface ................................................................................................................................. 4  
Introduction to the Snap! Coding System ........................................................................ 6

**Section A. Art**

1. Creating Artistic Patterns through Computer Graphics .............................................. 18  
2. Exploring Color ............................................................................................................ 33 
3. Creating Colorful Patterns ......................................................................................... 49 
4. Simulating an Impressionist Painting ......................................................................... 61 
5. Animated Art ................................................................................................................. 69

**Section B. Music**

6. Animated Art and Music ............................................................................................... 88 
7. Musical Notes and Scales ......................................................................................... 97  
8. Musical Chords ........................................................................................................... 123  
9. Building a Drum Track ............................................................................................... 139 
10. Lyrics and Vocal Tracks ............................................................................................ 151

**Appendices**

A. TuneScope User Guide .............................................................................................. 169 
B. Alignment with APCS Principles ............................................................................ 179
An Introduction to the Snap! Coding System

This initial module provides an orientation to the Snap! programming language. Snap! was developed at the University of California, Berkeley, and is used in computer science (CS) for non-CS majors at that university. Because of its widespread use, a user community has developed around this programming language. The capabilities of this language make it well-suited to explorations in art and music.

Topic 0.1 Securing a Snap! Account

A free Snap! account can be obtained from the University of California, Berkeley web site:

https://Snap.berkeley.edu/Snap

Secure a Snap! account before continuing. The Snap! reference manual is available here:


Section II of the reference manual provides information about securing a Snap! account and saving projects. Review this section before continuing to the explorations that follow below.

Snap! help forums maintained at the University of California, Berkeley, are available here:

https://forum.Snap.berkeley.edu/

Assistance with questions that are not addressed in the reference manual can be obtained through the Snap! forum.

Topic 0.2 The Snap! Workspace

In a typical Snap! session, blocks of code are dragged from the Code Block Palette on the left-hand side of the screen to a Script Area in the middle of the screen. Blocks of code are snapped together to create scripts. Clicking a group of code blocks causes the script to run, performing an action. Often these actions involve movement of sprites on a Stage at the right-hand side of the screen.

A sprite corral beneath the stage indicates which sprite is currently selected. Each sprite has its own separate script space.
**Topic 0.3 Snap! Menus**

There are several menus that can be accessed in the top left-hand corner of the Snap! screen. The Snap! icon in the top left-hand corner can be used to access the Snap! reference manual.

The *File* menu to the right of the Snap! icon can be used to save projects and open projects that have been previously saved. This menu can also be used to access costumes for sprites, sounds, and libraries of additional Snap! code blocks.

The *Login* menu is to the right of the *File* menu.

A user must be logged into Snap! in order to save projects. Therefore it is a good idea to log in to Snap! at the beginning of every session.

A *Settings* menu is to the right of the *Login* menu. This menu provides a number of options for customizing Snap! For example, the *Zoom Blocks* option can be used to increase the size of the code blocks so that they can be more easily viewed in presentations.

**Exploration 0.3**

If you have not already, log into Snap! Explore some of the options available under the different menu settings.
Topic 0.4 The Code Block Palette

Scripts are created in Snap! by snapping blocks of code together (such as the **Move 10 Steps** code block in the illustration). The types of code blocks available are displayed in a **Code Block Palette** at the top left-hand side of the screen. For example, the **Motion** code blocks are currently highlighted in the palette below. Other categories of code blocks include **Looks**, **Sound**, **Pen**, **Control**, **Sensing**, **Operators**, and **Variables**. Each category is a different color (e.g., **Motion** code blocks are blue). Click on the different categories (Motion, Looks, etc.) to access the code blocks associated with that category.

![Code Block Palette](image)

The **Motion** code blocks direct the movement of sprites (actors that can move about the stage on the right-hand side of the screen.) The **Looks** code blocks control the appearance of sprites. The **Sound** code blocks are used to play sounds. The **Pen** code blocks control the color and thickness of the turtle’s pen. The **Control** code blocks provide control structures such as the **Repeat** command. The **Sensing** code blocks are used to sense the status of Snap! objects and monitor external inputs such as the keyboard and the microphone. The **Operators** code blocks provide mathematical and logical functions. The **Variables** palette is used to create and modify variables.

**Exploration 0.4 The Code Block Palette**

Click on each of the categories in the **Code Block Palette** to get a sense of the types of commands that are found under each category.

Topic 0.5 A Digital Clock

Sound is produced by the back-and-forth motion of a vibrating object. The rate at which the back-and-forth motion occurs is an important characteristic of sound. Two closely related terms used to describe this characteristic are **frequency** and **period**.

**Frequency.** The rate at which a back-and-forth motion occurs over a given amount of time. The number of back-and-forth events that occur within a given time period is known as frequency.

**Period.** The duration of time for a single back-and-forth event is known as its **period**.

These terms are used to characterize sound, but are also used to define other non-auditory events. The motion of the rotating flywheel of an engine can be described in terms of **revolutions per minute**. For example, the flywheel might be described as rotating at a rate of 60 revolutions per minute (rpm). In this instance, the frequency of motion could be described as **60 rpm**. If the frequency is 60 rpm, this means that one complete rotation would occur in one second. Therefore, the period in which a single
rotation occurs could be described as 1 second. In this example, frequency and period are related. If the frequency increases, the period in which one rotation occurs will decrease.

The concepts of frequency and period can be applied to the back-and-forth vibration of an object that produces sound in a similar manner. The time scale for vibration that produces sound is typically events per second rather than events per minute, however.

A digital clock is at the heart of every digital computer. The digital clock is used to control the timing and rate at which computer instructions are executed. It can also be used to create a digital timer that can be used to time events.

**Topic 0.6 Rotating the Turtle**

The examples that follow make use of the digital clock to illustrate ways in which code blocks from several different code block palettes are used. In this example a timer will be constructed that will keep track of seconds as the turtle rotates. The user will be able to stop the timer (and the turtle) by pressing the space bar.

The code blocks in the *Motion Palette* are blue. Drag the **Turn Right** code block into the script area and enter 90 degrees as the input to this code block. Then click the code block. Each time the code block is clicked, the screen turtle should turn right 90 degrees.

**Exploration 0.6 Rotating the Turtle**

Explore some of the other code blocks in the *Motion Palette*. What does the **Move** code block do?
Topic 0.7 Controlling the Timing of the Turtle’s Rotation

Go to the Control Palette and drag a **Wait** block onto the script area. Snap it into place beneath the **Turn** block.

Then drag a **Forever** block from the Control Palette into the script area.

Drag the combined **Turn** and **Wait** blocks into the open space within the C-shaped **Forever** block.

The combined blocks should look like this.
Click the combined group of code blocks. (A group of code blocks that have been snapped together in this way is referred to as a *script.*) When the script is executed, an outline or halo will surround the script to indicate that the program is running. The screen turtle should turn right by 90 degrees one time per second. (*Forever* means that the turtle will continue to turn until the user stops the script.)

```
forever
turn 90 degrees
wait 1 secs
```

Click the red *Stop* button in the top right-hand corner of the screen to stop the script (or click the highlighted script).

**Exploration 0.7 Controlling the Timing of the Turtle’s Rotation**

Explore some of the other uses of variables. How could a timer be created that counts down the seconds?

**Topic 0.8 Tracking Elapsed Seconds**

Variables provide a way to keep track of minutes and seconds as the timer counts. To create a *Seconds* variable to track seconds, select the *Variables Palette* and click the *Make a Variable* button.

Enter the variable name *Seconds* in the dialog box that appears. (Leave the default option of “for all sprites” selected.) This variable will be used to track elapsed seconds.
A variable watcher *Seconds* will appear on the stage. All of the variables that have been created are listed at the top of the *Variables Palette*. When the checkbox beside each variable is selected, the *Variable Watcher* will appear on the stage. The variable watcher can be used to monitor the status of the variable.

To count seconds, the seconds variable will need to increase the count each time a second elapses. Drag the code block **Change __ by 1** into the script area. Select the variable *Seconds* from its drop-down menu.

Drag the **Change Seconds by 1** block into the **Forever** block. Click the group of blocks to execute the script. The variable *Seconds* will increase by one each time the turtle turns.
The revised script now functions as a timer. The code block “**Set Seconds to __**” can be placed before the **Forever** block to ensure that the timer always begins with a starting value of 0. (This step is called *initialization* because it establishes the initial value of the variable.)

**Exploration 0.8 Tracking Elapsed Seconds**
Create a timer that tracks elapsed seconds.

**Topic 0.9 Stopping the Timer**
A second script can be added to stop the timer when the space bar is pressed. Begin by dragging the **When** code block from the **Control Palette** into the script area.

Select the **Sensing Palette** and drag the **Key Space Pressed** block into the hexagonal slot in the **When** block.

Then add the **Stop All** block from the **Control Palette**.
Finally, add the **Green Flag** block to the other script. The green flag block will enable the user to start the program by clicking the green flag in the upper right-hand corner of the screen. Pressing the space bar on the keyboard will stop the timer.

**Exploration 0.9 Stopping the Timer**

Explore some of the other code blocks in the *Sensing Palette*. What does the **Mouse Down** code block do? How could this code block be incorporated into the timer?

**Topic 0.10 Counting Revolutions of a Rotating Turtle**

To count the number of revolutions of the turtle that occur within a given time period, add a *Revolutions* counter. Use the **Make a Variable** button in the *Variables Palette* to create a variable named *Revolutions*.

Begin by setting *Seconds* and *Revolutions* to an initial state of zero. Point the turtle straight up. This process of establishing the conditions for an initial state is known as *initialization*. 
The **Direction** code block reports the direction in which the turtle is pointing. In the illustration below, the turtle is pointed to the right, at an angle of 90 degrees.

The **Direction** block can be combined with an **If** code block to increase the **Revolutions** counter by one each time turtle returns to a heading of 0 degrees after completing a full rotation. (The **Equals** code block used in this example is found under the green **Operators** palette.)

The **If** code block is placed within the **Forever** code block. Each time the turtle completes a rotation, the **Revolutions** counter is increased by one.
The completed procedure initializes *Seconds* and *Revolutions* when the green flag is clicked. It also starts the loop that rotates the turtle and counts revolutions and seconds until the space bar is pressed.

In this illustration, the turtle completed 15 revolutions in 60 seconds. The frequency of rotation, therefore, can be described as *15 revolutions per minute*. Dividing 15 revolutions into 60 seconds yields 4 seconds. The *period* of one rotation, therefore, is 4 seconds.

**Exploration 0.10 Counting Revolutions of a Rotating Turtle.**

Change the **Wait** time to 0.5 seconds. How many revolutions occur in one minute when this change in timing is made? How does this affect the period of time that it takes each revolution to occur?

**Topic 0.11 Saving the Script**

This is a good point to save the script. If the **Save** option (under the **File** menu) is used when the script is first saved, it will be given the default name of “Untitled.” Since that name is not descriptive, use the **Save As** selection the first time that the script is saved.

This will produce a dialog box that allows the project to be given a name. In this instance, the project has been named, “Timer.” The project can be saved on a local computer. The advantage of saving the project in your Snap! account in the cloud is that it can be later retrieved from any computer with an Internet connection. (You must be logged into Snap! in order to save projects.)
After the project is saved, the project name should replace the default name of “Untitled”. After the project is saved and given a name, the Save option under the File menu can also be used to save future changes and revisions that may be made to the script.
1. Creating Artistic Patterns through Computer Graphics

Glen Bull, Jo Watts, and Alexis Kellam

Many artists now use computer graphics to create artistic patterns. Bathsheba Grossman is representative of a new generation of artists who use computers to extend the type of art that they create. The design below was created by Bathsheba Grossman (https://bathsheba.com/crystal/calabiya/) using a computer-generated pattern.

These artists use computer code to create art. Some of the techniques that they use are available to anyone with an interest in exploring art in this way.

Topic 1.1 Algorithmic Thinking – Finding a Pattern

The first robotic turtles were developed at M.I.T. These robots consisted of Plexiglas hemispheres that rolled about the floor. They were called “turtles” because the Plexiglas hemispheres resembled the shell of a turtle. A pen in the belly of the turtle could be lowered to draw patterns on paper placed on the floor.

Robotic turtles are still used to draw patterns on paper today. However, a turtle on the screen of the computer can be used to create a design more quickly. Screen turtles typically consist of a triangle on the screen of the computer. The Pen Up and Pen Down code blocks are used to raise and lower the pen in the belly of the screen turtle.

The explorations that follow begin with simple patterns. These simple patterns are then used to create more complex patterns.

A square, for example, is one of the most basic patterns that can be created. A square is created by drawing a line and then turning 90 degrees. For example, if the turtle begins in the position shown below (pointing right):
then executing the code blocks Move 100 Steps followed by Turn 90 Degrees causes the turtle to draw a line and then turn right.

Repeating the Move and Turn code blocks four times causes the turtle to draw a square.

However, repeating the pattern four times achieves the same result more efficiently. Computers are efficient at repeating a pattern many times. However, the human coder must recognize the pattern in order to take advantage of the computer’s capabilities in this way.

Computing Concepts

The example above illustrates the process of repeating a series of steps to draw a square. This can be done most efficiently by identifying the pattern involved in drawing the square. The pattern in this case consists of a Move command followed by a Turn command: Move 100; Turn 90 Degrees.

This pattern provides a recipe for drawing a square. In computer science, the term algorithm is used to describe a pattern that can be used as a recipe for creating the steps of a program.
The process of repeating the steps of the pattern several times is known as iteration. In the case of the square, the steps of the pattern (Move Forward and Turn) are repeated four times (i.e., four iterations).

An algorithm can be written in multiple ways and still accomplish the same task. A square can be drawn in eight steps; the same pattern can be drawn more efficiently by using a Repeat code block in combination with one Move code block and one Turn code block.

**Exploration 1.1 Algorithmic Thinking – Finding a Pattern**

Identify the pattern used to create other shapes such as triangles (three-sided figures) and polygons (five sized figures). Assemble and run code blocks that draw shapes such as triangles and polygons.

**Topic 1.2 Abstraction – Making a Custom Code Block**

The limits of human memory place limits on creation of computer code. A human can only retain a limited number of code blocks in memory at one time. Experienced coders circumvent these limits through abstraction. Once code has been developed and tested, it can be abstracted through creation of a custom code block.

Clicking the plus sign near the top of the command palette invokes the option to make a custom block.

A dialog box appears when the Make a Block option is invoked. The dialog box includes a space to enter the name of a new custom block. (Note: in most instances the default selection of “for all sprites” is used.)
When the entered name is confirmed by clicking “OK,” a block editor appears. The working code previously developed and tested can then be dragged into the block editor and snapped into place beneath the header block at the top of the block editor.

Once this choice is confirmed by clicking “OK,” the newly defined custom block appears at the bottom of the other commands in the command palette.

The new custom block can then be dragged into the script area and used in the same way as any other code block.

Creating a custom block reduces the burden on the coder’s memory. Instead of remembering the correct order of an entire sequence of code blocks, the coder now only needs to remember the name of a single custom block. For this reason, it is important to create a meaningful name for the custom block so that its purpose is clear.
The introduction to a famous computer science textbook, the *Structure and Interpretation of Computer Programs*, asserts in the preface that “Programs must be written for people to read, and only incidentally for machines to execute” (Abelson & Sussman). Elegance in programming is a balance between efficiency (writing code in as few lines as possible) and readability (creating custom code blocks with meaningful names). Readability ensures that code can readily be maintained and reused in other programs.

**Computing Concepts**

A procedure is a group of program instructions that have been given a name. In the example above, the instructions for drawing a square are assigned the name “Square”.

In Snap! a custom code block is created by assigning a name to a group of instructions. Custom code blocks can be used in Snap! in the same way that built-in code blocks are used.

The term *abstraction* refers to the process of naming a concept without explaining the details of the concept each time. The process of assigning a name to a group of instructions to create a procedure is one form of abstraction. Once a code is tested and encapsulated in the form of a procedure, the programmer no longer needs to remember the specific details of the procedure. This enables programmers to focus on other program details and create much more complicated programs than otherwise would be possible.

In the example above, a custom code block named Square replaced a longer sequence of instructions. Replacing an entire sequence of instructions with a named procedure yields several benefits.

1. It makes the top level of the program more concise and easier to read.
2. If meaningful names are used, it makes the program easier to follow.
3. The cognitive load on the programmer is reduced by replacing a longer sequence of code with a procedure name. This enables the programmer to devote attention to other details of the program.

**Exploration 1.2 Abstraction – Making a Custom Code Block**

Create custom code blocks for other shapes such as *Triangle* and *Pentagon*.

**Topic 1.3. Generalization – Creating Inputs for Custom Code Blocks**

One of the goals of effective programming is creation of re-usable code. Inputs to custom code blocks make these code blocks more generalizable.

The previously created **Square** code block made a square that was 100 turtle steps on each side. In other circumstances, a square that was 50 steps on a side or 200 steps on side might be needed. One approach to addressing this need consists of creating a new custom block – such as **Small Square** and **Big Square** – for each new application.

However, a more effective approach would be to incorporate an input into the procedure that made it possible to use a single code block to draw a square of any size.

Right-clicking the **Square** code block reveals a drop-down menu that offers the option of editing the code block.
Selection of the *Edit* option accesses the *Block Editor*. If the plus sign (“+”) beside the name *Square* is clicked, another dialog box appears that offers the option of entering an input name.

Once an input name is selected (*Size* in this instance), the choice can be confirmed by clicking “OK.”

Once this choice is confirmed, the input name (*Size*) will appear to the right of the block name (*Square*).
This orange oval representing the input *Size* can then be dragged into the input slot of the **Move** code block.

Once this revision of the custom code block is confirmed, an input slot will appear to the right of the name of the code block.

The input slot makes it possible to use the same code block to create squares of three different sizes, rather than creating three different code blocks for the same purpose.

Use of inputs to create more generalizable procedures greatly increases the efficiency of coding.

**Computing Concepts**

Procedures are implemented in the form of custom code blocks in Snap!. In Snap!, a code block can be executed (i.e., called) by clicking it.

Procedures can have inputs. For example, an input to the **Square** procedure can be used to specify the size of the square. In computer science, the term *parameter* can be used to describe an input such as size that can be varied in this manner.

Use of an input such as size allows the procedure to be generalized. This enables the procedure to be re-used to draw squares with a wide range of different sizes.

In the example above, the numbers “50”, “100”, and “200” are used as inputs to the procedure. The *size parameter* allows the procedure to be generalized, enabling **Square** to be used to draw squares of different sizes.

**Exploration 1.3 Generalization – Creating Inputs for Custom Code Blocks**

Create inputs for other shapes such as the Triangle and Pentagon code blocks.

**Topic 1.4 Building Complex Code Block by Block**

Effective coding begins with construction of basic building blocks that are used to create more complex procedures.
For example, if a series of squares is drawn, turning a bit after each square is drawn, the following pattern results.

![Image of a circle drawn by repeating a block of commands]

In this manner, the Square code block becomes a building block for construction of another custom block, **Spin Square**.

Surprisingly complex patterns can be generated through construction of a library of custom code blocks in this manner.

**Computing Concepts**

An effective coding strategy breaks a larger problem into a series of smaller sub-problems. Procedures are created to solve each of the smaller sub-problems. The larger problem is then solved by combining the procedures that were developed to solve each of the sub-problems.

In the example above, in order to create the **Spin Square** procedure, the larger problem was separated into two smaller problems: (1) drawing the square and (2) turning the square. The **Spin Square** code block is based on the procedure **Square**. Development of the **Square** procedure makes the code easier to understand when it is incorporated into a more complex program in the future.

**Exploration 1.4 – Building Complex Code Block by Block**

Incorporate the **Triangle** and **Pentagon** code blocks into other code blocks to create complex patterns.

**Topic 1.5 Using Loops to Vary a Parameter**

The **Repeat** block represents one type of loop in which the same action is repeated over and over again. A **For** block is another type of loop. One advantage of a **For** loop is that the variable \( i \) changes each time that the loop is executed.

In the example below, the variable \( i \) has a value of “1” the first time the loop is executed. The value “1” is multiplied by 10 in the input to the **Square** code block. Consequently, a square with a size of 10 steps on a side is drawn the first time the loop is executed.
The second time that the loop is executed, a square is drawn that is 20 steps on a side. Since \( i = 2 \) during the second execution of the loop, the input to the \textbf{Square} block becomes \( 2 \times 10 \) (i.e., 20).

The last time that the loop is executed, a square that is 100 steps on a side is drawn. In this manner, a series of nested blocks is drawn using just two code blocks, the \textbf{For} loop combined with the \textbf{Square} code block.

The ability to vary the input of a code block each time that the loop is executed can therefore yield some powerful effects in an efficient way. Almost every program of any complexity will make use of one or more loops in this manner.

\textbf{Computing Concepts}

A variable is a name that can be assigned a value. For example, the name \textit{Size} might be assigned a value of “40”. In Snap!, the \textbf{Set} command can be used to assign a value to a variable.

The \textbf{Operators} palette in Snap! provides mathematical and logical operators that can be used to evaluate an expression such as “\( 2 + 2 \)” (which equals “4”) or “\( \text{Size} = 40 \)” (which returns a value of “True” if the variable \textit{Size} has been set to “40”).

In the example above, the multiplication operator is used to multiply the variable \( \textit{i} \) by “10” to draw a series of squares with increasingly larger sizes.

The term \textit{incrementing} is used to describe the process of changing the value of a variable during each iteration of a loop.

\textbf{Exploration 1.5 – Using Loops to Vary a Parameter}

Explore creation of nested shapes such as triangles and pentagons through use of loops.

\textbf{Topic 1.6 Fabricating Art with Patterns}

Tools such as inkjet printers, laser cutters, and other fabrication tools can be used to translate digital designs into ornaments, jewelry, and sculptures in the same manner as digital artists.
Print your design with an ink-jet printer. If you have access to a laser cutter or 3D printer, use the pattern to create a three-dimensional ornament.

Graphics drawn with the turtle can be exported as Scalable Vector Graphics (SVG) files for higher resolution output. To access the Scalable Vector Graphic feature, turn on the Log Pen Vectors option in Settings.

After a Snap! program has been used to draw a design, place the mouse cursor on any part of the design and right-click to access the menu with the “SVG Export” option.

The SVG file can then be imported into other graphics programs such as Silhouette Studio. (Silhouette Studio Business Edition includes an option to import SVG files.)
Foil Quill is a third party option available for the Silhouette die cutter that can be used to emboss foil patterns onto materials such as card stock.

The result is an embossed foil pattern obtained by using an SVG vector pattern generated by Snap!.

**Exploration 1.6 – Fabricating Art with Patterns**
Design a card with your own unique artistic design.
Appendix A
Code Blocks introduced

A. Motion Palette

Move and Turn

The Move code block is used to move the turtle forward. (If a negative number is used as an input, the turtle moves backward.)

![Move and Turn Diagram](image)

The Turn code block is used to rotate the heading of the turtle.

B. Pen Palette

Pen Up and Pen Down

The Pen Up and Pen Down code blocks are used to raise and lower the pen in the belly of the screen turtle.

![Pen Up and Down Diagram](image)

C. Control Palette

Repeat

The Repeat code block consists of a “C-shaped” block. Other code blocks places within the Repeat command are repeated the specified number of times.

![Repeat Diagram](image)

For Loop

The For code block consists of a “C-shaped” block that can be used to repeat a sequence of actions while changing the value of a variable (such as “i” in the example below) during each iteration of the loop.

![For Loop Diagram](image)
Appendix B

New APCS Principles

This appendix lists Advanced Placement Computing Standards (APCS) associated with each topic.

Topic 1.1 Algorithmic Thinking – Finding a Pattern

The section illustrates the process of repeating a series of steps to draw a square.

Algorithms

The term algorithm refers to a set of instructions that accomplish a task. A computer algorithm refers to a set of instructions that can be implemented by a computer.

CS Principle AAP-2.A Developing Algorithms

An algorithm that uses sequencing can be expressed without using a computer language. For example, before approaching the code, a student should understand that to make a square, they will need to write a procedure that creates a single side and a right angle and then repeat that procedure four times. After understanding that, the concepts can be transferred to assembling code blocks.

CS Principle AAP-2.B Programming Algorithms

Represent a step-by-step algorithmic process using sequential code statements.

CS Principle AAP-2.J Iteration

A computer often needs to repeat the same series of steps several times. The term iteration refers to a repeated sequence of steps. In the example in the preceding section, a series of steps is repeated to create an artistic pattern.

Application

In Topic 1.1, an algorithm for drawing a square is developed. Development of this algorithm depends on recognition that the Move code block and the Turn code block can be combined and repeated to form a square.

Topic 1.2 Abstraction – Making a Custom Code Block

A procedure is a group of program instructions that have been given a name. In the example above, the instructions for drawing a square are assigned the name “Square”.

CS Principle AAP-3.A Procedures

A procedure is a named group of programming instructions. In Snap! a custom code block is created by assigning a name to a group of instructions. Custom code blocks can be used in Snap! in the same way that built-in code blocks (known as primitives) are used.


Assigning a name to a group of instructions to create a procedure is one form of abstraction. Once a code sequence is tested and encapsulated in the form of a procedure, the programmer no longer needs to remember the specific details of the procedure, enabling the programmer to focus on other program details.

Application
In the example in this section, a custom code block named **Square** replaced a longer sequence of instructions. Replacing an entire sequence of instructions with a named procedure yields several benefits. First, it makes the top level of the program more concise and easier to read. Second, if meaningful names are used, it makes the program easier to follow. Finally, it reduces the cognitive load on the programmer by replacing a longer sequence of code with a procedure name, enabling the programmer to devote their attention to other details of the program.

**Topic 1.3. Generalization – Creating Inputs for Custom Code Blocks**

**CS Principle AAP-3.A Parameters for Procedures**

Procedures frequently have inputs. For example, an input to the **Square** procedure can be used to specify the size of the square. In computer science, the term *parameter* can be used to describe an input such as size that can be varied in this manner.

**CS Principle AAP-3.C Developing Procedures**

Developing abstractions helps manage the complexity of a procedure. For example, use of an input such as size allows the procedure to be generalized. This enables the procedure to be reused to draw squares with a wide range of different sizes.

**Application**

In Topic 1.3, the numbers “50”, “100”, and “200” are used as inputs to a procedure. The **size** parameter allows the procedure to be generalized, enabling **Square** to be used to draw squares of different sizes.

**Topic 1.4 Building Complex Code Block by Block**

**CS Principle AAP-3.D Selecting Appropriate Procedures to Use in Programs**

An effective coding strategy breaks a larger problem into a series of smaller sub-problems. Procedures are created to solve each of the smaller sub-problems. The larger problem is then solved by combining the procedures that were developed to solve each of the sub-problems.

Procedural abstraction allows a solution to a large problem to be based on the solutions of smaller sub-problems. This is accomplished by creating procedures to solve each of the sub-problems.

**Application**

In Topic 1.4, the larger problem was separated into two smaller problems: (1) drawing the square and (2) turning the square to create the **Spin Square** procedure. The **Spin Square** code block is based on the procedure **Square**. Development of the **Square** procedure makes the code easier to understand when it is incorporated into a more complex program in the future.

**Topic 1.5 Using Loops to Vary a Parameter**

**CS Principle AAP-1.A Variables**

A variable is an abstraction inside a program that can hold a value. Each variable has associated data storage that represents one value at a time, but that value can be a list or other collection that in turn contains multiple values.

Using meaningful variable names helps with the readability of program code and understanding of what values are represented by the variables.

**Application**

Variables can be used to change the length of a shape’s sides or the angle of its corners.
CS Principle AAP-2.C Mathematical Expressions

Mathematical expressions, sometimes called operators, enable the computer to perform mathematical calculations. Expressions are evaluated to produce a single value.

Expressions are part of most programing languages and include addition, subtraction, multiplication, division, and modulus operators.

Application

My modifying a variable with a mathematical expression, the shape drawn can be changed with each iteration.

CS Principle AAP-2.K Iterative Statements

Iteration is a repeating portion of an algorithm that repeats a certain number of times or until a given condition is met.

Application

Iteration can be used to repeat statements that will draw a single shape. It can also be used to repeat procedures that will draw shapes differently with each iteration.

CS Principle CRD-2.B Programs

A program is a collection of statements that performs a specific task when run by a computer. Statements with a program are called code segments. Code segments often incorporate procedures to simplify the code and make it easier for humans to read and write. A program can be described broadly by what it does, or in more detail by both what the program does and how its statements accomplish this function.

Application

In this section, a program is used to draw a sequence of inset squares. The program consists of an iterative loop, a variable, a procedure, and a multiplication operator. The procedure draws a square using the variable from the loop and a multiplication operator to set the square’s size. Each time the loop repeats, the variable increases to the next integer and the procedure draws a square. This process continues until ten squares have been drawn.
2. Exploring Color

Glen Bull, Jo Watts, and Alexis Kellam

An understanding of the Snap! color framework explored in this chapter is required for subsequent art modules. These modules include one that simulates an impressionist painting, another module in which a paint program is developed, and a module involving creation of an animated coloring book. Each of these modules makes use of color.

Creating Art through Coding

Mark Rothko was an American abstract painter who explored color usage through his artwork. His signature colorfield paintings depict irregular and rectangular areas, known as multi-forms, that encouraged color contemplation and appreciation. The majority of these multi-forms were horizontally painted blocks. Occasionally the orientation of the rectangular multi-forms varied.

Using the pen up and pen down line drawing techniques from Chapter 1 in conjunction with the color palette and size features of the turtle discussed below, Rothko-inspired creations can be made in Snap!
**Topic 2.1 Setting the Turtle’s Size and Color**

The previous module introduced the ability of the sprite turtle to draw by raising and lowering a pen in its belly. The color of the line drawn by the turtle can be changed by using the **Set Pen Color** code block.

![Set Pen Color](image)

When the pen color is changed, the body of the turtle also changes to the same color.

![Set Pen Color](image)

The **Set Size** code block is used to increase the size of the turtle. This makes the color easier to see.

![Set Size](image)

**Computing Concepts: Mathematical Operators**

Programming languages use mathematical expressions like addition, subtraction, multiplication, and division. These expressions are known as mathematical operators. They enable the computer to perform mathematical calculations.

Mathematical operators are found in the green **Operators** palette in Snap!. These blocks report the mathematical result.

![Mathematical Operators](image)

**Exploration 2.1 Exploring the Snap! Color Space**

Use the **Set Pen Color** code block to explore the range of different colors available in the Snap! color space.

**Topic 2.2 Functions: Finding the Turtle’s Pen Color**

The values of the pen colors available in Snap! range from 0 to 255. The current pen color of the turtle can be determined by using the **Pen Hue** reporter block. For example, the shade of green in the illustration below corresponds to **Pen Hue** number “31”.

![Pen Hue](image)
In Snap! code blocks that perform an action such as the Move block have the shape of a rectangle.

In contrast, code blocks that report a value are known as reporter blocks. These code blocks have an oval shape. The oval Pen Hue block reports the value of the turtle’s current pen color.

In other computing languages, code that reports a value is known as a function. The term “function” is also used in mathematics. Mathematical functions and functions in computer science are similar but not exactly the same. Using the term reporter block to describe a code block that reports a value eliminates any ambiguity or confusion about the role of this type of block.

The Pen Hue reporter block reports the value of the pen color to more than a dozen decimal places.

This is more precision than will be required in subsequent modules that make use of color. Another type of reporter block, the Round reporter block, can round the pen color value to the nearest whole number.

The Pen Hue block reports that the blue pen color of the turtle is “67.2916666666666”. This number is supplied as an input to the Round reporter block. The Round reporter block, in turn, reports that the nearest whole number is “67”.

It is common to nest one reporter block within another to perform a series of operations in this fashion.

Computing Concepts: Functions

In mathematics, a function defines a relationship between one variable (the independent variable) and another variable (the dependent variable). For example, the expression “X = 3Y” defines a relationship in which Y is three times greater than X. In computing, a function returns a value.

In Snap!, a reporter is a code block that reports a value. For example, the Round reporter block accepts an input such as “3.7” and reports the nearest whole number “4”. In this example, the number “3.7” is the independent value and the number “4” is the dependent value. A reporter block has an oval shape, in contrast to the rectangular shape of a procedural block that performs an action such as Move 100.

Exploration 2.2 Finding the Turtle’s Pen Color

Change the turtle’s pen color by using Set Pen Color to change its color. Then use the Pen Hue block to identify the value of the pen color.

Explore the range of color values that are perceived as the same color. How much does a color value have to change before the color is perceived as different? For example, does changing the color value from 37 to 40 make a perceptible difference in the perceived color?
Topic 2.3 Variables and Loops: Varying the Turtle’s Pen Color

There are several color variables in the HSL color model: hue (color), saturation (shade), and brightness (light level). The values for each of these variables range from 0 to 255.

A Snap! variable can be created and combined with a loop to vary the turtle’s pen color. The Make a Variable option is found under the orange Variables section of the Snap! command palette.

In this illustration, a variable named Color has been created and made available to all sprites. (The default option of “for all sprites” will be selected in most of the variables created in subsequent modules.)

Once a variable has been created, the name of the variable will appear directly beneath the Make a Variable button.
When the checkbox beside the variable name is selected, the value of the variable appears in a *watcher* on the stage. The *watcher* is a display that shows the value of the variable.

Right-clicking the watcher displays several options, including an option to create a slider that can be used to adjust the value of the variable.

The default value of the slider is 0 to 100. However, options are available (accessed by right-clicking the watcher) to adjust the minimum and maximum values of the slider.

If the oval reporter block containing the name of the variable is dragged from the command palette into the script area, clicking on the reporter block will also display the current value of the variable.

If the *Color* variable block is dragged into the input of the *Set Pen Hue* code block, the *Set Pen Hue* block can be used to set the pen color to the current value of the variable *Color*.

Placing the *Set Pen Hue to Color* block inside a *Forever* loop allows the color of the sprite to be updated as the *Color* slider is moved back and forth. This provides a convenient way to quickly identify the number associated with a given color.
Computing Concepts: Variables

A variable is a place-holder that can be given any value. In example above, a variable named Color is created. The value of this variable is controlled by a slider. Using meaningful variable names makes the program easier to read.

Exploration 2.3 Variables and Loops – Varying the Turtle’s Color

Use a slider to change the turtle’s color. Then identify the minimum and maximum values associated with the following colors: red, orange, yellow, green, blue, and purple.

<table>
<thead>
<tr>
<th>Color</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orange</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yellow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Green</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Purple</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is the range of values associated with each color band similar or different?

Topic 2.4 Other Color Characteristics: Varying the Turtle’s Shade

The shade of a color is affected by the saturation parameter in the HSL color model.

A variable named Shade can be combined with the Set Pen Saturation code block to adjust the shade of a color. For example, if the Pen Hue is set to 164 and the Pen Saturation is set to 255, the color is perceived as dark blue. However, if the Pen Hue is set to 164 and the Pen Saturation is set to 27, the color is perceived as light blue.

Computing Concepts: Variables

Use of a variable such as Color makes it easier to explore a range of values. Linking the variable to a slider bar enables a user to quickly sweep through a range of colors.

Exploration 2.4 Other Color Characteristics – Varying the Turtle’s Shade

Use a slider to change the turtle’s shade. Then identify the values associated with the following light and dark shades for the following colors: red, green, and blue.

What value is associated with a shade that is completely white (i.e., that has no perceptible color)?
**Topic 2.5 Other Color Characteristics: Varying the Turtle's Light Level**

The light level (luminosity) of a color is affected by the *brightness* parameter in the HSL color model. A variable named *Light Level* can be combined with the *Set Pen Brightness* code block to adjust the light level of a color. For example, if the *Pen Hue* is set to 164 and the *Pen Brightness* is set to 255, the color is perceived as blue. However, if the *Pen Hue* is set to 164 and the *Pen Brightness* is set to 50, the color is perceived as dark blue.

**Computing Concepts**

Use of sliders combined with variables for color, shade, and light level creates a user interface that enables users to interactively explore the relationships among these variables.

**Exploration 2.5 Other Color Characteristics – Varying the Turtle’s Light Level**

Use a slider to change the turtle’s light level. Then identify the values associated with the following normal and dark shades for the following colors: red, green, and blue.

What value is associated with a light level that is completely dark (i.e., that has no perceptible color)?

**Topic 2.6 Art Blocks**

The sample illustration in the style of Rothko is composed of bars and blocks of varying colors.

A block can be created in Snap! by using the *Square* procedure developed in the previous section (*Creating Artistic Patterns*) as a starting point. The primary difference is that while the *Square* code block has a single input, *Size*, the custom *Block* code block has two inputs, *Width* and *Height.*
A user might easily forget whether the *width* or *height* input is entered first. To assist users, the *Title Text* option can be used to add descriptive titles after each input. If the plus sign (“+”) after the input *Width* (shown as an orange oval in the illustration below) is clicked, an additional dialog box will appear. Enter the name *Width* and select *Title Text* to create a descriptive title after the input.

In the example below, the custom code block *Block* has been used to draw a rectangular block that is 100 steps wide and 20 steps high.

*Block* 100 Width 20 Height

Rothko art is primarily composed of solid blocks and bars. The basic *Block* procedure can be used as a starting point for creating a solid bar filled with color.
A **Solid Bar** code block can be constructed in two steps:

1. First develop a **Go to Block Center** custom code block. This code block places the turtle in the center of a block.

   The **Go to Block Center** code block changes the horizontal position (X) of the turtle by the width of the block divided by two. The vertical position (Y) is changed by the height divided by minus 2.

   ```
   Go to Block Center 100 Width 20 Height
   ```

   This change in the vertical and horizontal position of the turtle places the tip of the turtle in the center of the rectangular block.

2. Once the **Go to Block Center** code block has been developed, it can be combined with the **Block** and **Fill** code blocks to create a **Solid Bar** code block.

   ```
   Solid Bar 120 Width 30 Height
   ```

   The **Solid Bar** code block can be used to create a solid bar filled with color.
Exploration 2.6 Art Blocks

It is good practice to leave the turtle at the same position at the end of the procedure as it was at the beginning of the procedure. Leaving an object in the same position as before is known as state transparency. The custom Solid Bar code block described in this section does not leave the turtle in its original position. Modify the Solid Bar procedure so that the turtle is left in its starting position.

Explore creation of several solid bars of varying sizes and colors. After creating several bars, identify other ways in which the Solid Bar code block might be refined based on your exploration.

Topic 2.7 Creating Art in the Style of Rothko

Once a solid bar has been created to your desired dimensions and color, the bar can be turned into a Costume. In order to do this, right click on the bar in the stage area.

Choose the option Pen Trails. This turns all pens trails into a new costume for the currently selected sprite. The bar can now be found under the Costumes tab for the sprite.

Converting the bar into a Costume allows it to be manipulated with code. It is helpful to give each bar costume a name that makes it easily identifiable based on its visual features. This can be done by right clicking the newly created bar costume and selecting the Rename option.
Rothko art typically consists of several bar variations. In order to emulate that feature, several bar Costumes can be created using different colors, sizes, and orientations. Rothko uses color intentionally in his artwork in order to present an aesthetically pleasing piece. The color of the bar that is being created can be selected using the Set Pen Color code block.

Clearing the stage before creating a costume ensures that no other unwanted graphics become part of the costume.

To increase or decrease the size of the turtle after creating a costume, use the Change Size block (found under the Looks palette).

To change the orientation of the turtle and its associated costume, use the Point in Direction block.

The Stamp code block can be then used to stamp an image of the block onto the stage. You can move the turtle around the stage in order to position stamped blocks in desired locations. Positioning the bar costumes on the stage in an aesthetically pleasing artistic pattern art in the style of Rothko to be created.

In order to switch between costumes to select bars of different sizes and colors, use the Switch to Costume code block (found under the Looks palette).

A dropdown menu will list each bar costume (hence the importance of assigning a meaningful name to each costume).
To save an image of the art created, right click the stage area and select “pic…” A copy of the artwork will then be saved locally on the computer’s desktop.

The saved image will have a “.png” file extension.

**Exploration 2.7 Creating Art in the Style of Rothko**

Create an artistic pattern in the style of Rothko by manually dragging the turtle to different locations and stamping bars of different sizes and colors on the stage.

Then create the same design by using a script to move the turtle to different locations on the stage. How can regularities in the pattern of bars be used to create more efficient scripts by incorporating a **Repeat** code block to take advantage of the regularities?
Appendix A
Code Blocks Introduced

A. Motion Palette

Change <X | Y> By

The **Change X** and **Change Y** code blocks change the horizontal and vertical position of the turtle by the specified amount.

![Image of change x by 10 and change y by 10]

Point in Direction

The **Point in Direction** block causes the turtle to point in the specified direction.

![Image of point in direction 90°]

B. Looks Palette

Set Size

The **Set Size** block is used to set the size of the turtle.

![Image of set size to 400%]

Change Size

The **Change Size** code block changes the size of the turtle by the specified amount.

![Image of change size by 10]

Switch to Costume

The **Switch to Costume** block switches the turtle’s shape to the specified costume.

![Image of switch to costume with costume options]
C. Pen Palette

Set Pen Color

The **Set Pen Color** block provides access to a color palette that can be used to set the pen color of the turtle. The pen color attribute determines the color of lines drawn by the turtle.

Set Pen `<Characteristic>`

The **Set Pen Characteristic** block is used to set the value of selected characteristics of the turtle’s pen such as *Hue*. Values can be entered directly by typing in a number, or they can be provided in the form of a variable such as *Color*.

Pen `<Characteristic>` Reporter Block

The **Pen `<Characteristic>`** reporter block reports the current attributes of the turtle’s pen state such as Pen Hue. These attributes are accessed via a drop-down menu.

Clear

The **Clear** code block clears the stage by erasing any lines drawn by the turtle.

Stamp

The **Stamp** block causes a copy of the turtle’s current shape to be stamped onto the stage.

D. Operators Palette

Arithmetic Operators

Mathematical reporter blocks report the results of mathematical operations such as addition, subtraction, multiplication and addition. Note: Reporter blocks can be recognized by the oval shape of the code block.
Round

The **Round** reporter block reports the nearest whole number of a value. Values can be entered directly by typing in a number or provided as an input from another reporter block such as *Pen Hue*.

![Round Block Examples]

E. Variables Palette

**Set Variable**

Variables are one of the basic building blocks of almost all programs. The **Set** code block is used to set the value of a variable to a specified value.

```
set Color to 50
```

New variables can be created using the *Make a Variable* button that is located below the command palette tabs.

![Make a Variable Button]

Once a variable has been created, it appears in the drop-down menu of the **Set** code block.
Appendix B
New APCS Principles

Topic 2.2 Functions: Finding the Turtle’s Pen Color

**CS Principle CRD-2D** Identify Outputs

Program outputs are any data sent from a program, often to another program or to a device. Program outputs are often based on a program’s input or current state.

**Application**

In this section, the hue of the turtle is initially found using the **Pen Hue** reporter block. Reporter blocks will always produce an output. In this instance, the reporter block outputs the turtle’s current state.

Topic 2.3 Variables and Loops: Varying the Turtle’s Pen Color

**CS Principle AAP-1.A** Variables and Assignments

*(Introduced in Topic 1.5)*

**CS Principle CRD-2.C** Identify Inputs

Program inputs are data sent to a computer for processing by a program. Inputs usually affect the output produced by a program.

**Application**

A variable named **Color** is created in this section. The value of this variable is controlled by a slider. The variable is then used as an input to change the color of the turtle.
3. Creating Colorful Patterns

Glen Bull, Jo Watts, and Alexis Kellam

The Snap! color framework explored in the previous chapter can be used to create colorful patterns filled with polka dots and stripes. This exploration, in turn, will lead to simulation of impressionist paintings in the next chapter.

Snap! sprites move about an area known as the Stage. The ability to move the sprites across the stage will play an important role in creation of artistic patterns.

Creating Art through Coding

Jackson Pollock was an American painter in the abstract expressionist movement. Pollock was known for his paint pouring technique that left his canvas covered with paint in a uniquely nontraditional way.

Creating these whimsical and colorful projects in Snap! can be reminiscent of the vibrant paint work done by Pollock. Varying the size of the dots or the thickness of the lines in combination with the randomized positioning of the pen and changes in the pen hue can allow you, the artist, to create art in the style of Pollock.

The illustrations above were created through use of these techniques. The methods used are described in the sections that follow.
Topic 3.1 The Stage

The *Stage* is the space in which the turtle can move about. The default stage is 360 turtle steps high and 480 turtle steps wide.

The **Go to X _ Y _ block** can be used to send the turtle to any part of the stage. The X coordinate adjusts the horizontal position of the turtle and the Y coordinate adjusts the vertical position of the turtle.

**Exploration 3.1 The Stage**

Explore the dimensions of the stage by using the **Go To** code block to move the turtle about the stage. What are the X and Y values associated with the center of the stage and each of its four corners (top left, top right, bottom left, and bottom right)?

Topic 3.2 Moving to a Random Position

The **Pick Random** reporter block can be used to pick a random number.

By picking a number between the left side of the stage (-240) and the right side of the stage (+240), the **Go To** block can be combined with the **Pick Random** block to send the turtle to a random position on the horizontal axis. The process can be duplicated for the vertical (Y) axis, using -180 and 180 to send the turtle to a random position on the horizontal and vertical axes.

This capability will be used to place polka dots at random locations across the stage.
Effective coding encapsulates code that will be reused in a custom code block once the code has been tested and verified to work properly. This reduces the burden on human memory, and makes it possible to create more complex programs than otherwise would be possible.

It is important to assign a meaningful name to a custom code block when it is created. A meaningful name enables the programmer to use the custom code block without the necessity of remembering all of the details of how the code inside the custom block works. In this instance, because the custom code block will send the turtle to a random position on the stage, the custom code block will be named **Go to Random Position**.

**Computing Concepts**

Generation of random numbers has many applications in computing, ranging from generation of white noise in the field of audio to selection of a random position for the turtle.

**Exploration 3.2 Moving to a Random Position**

Create a **Go to Random Position** custom code block, and use it to move the turtle about the stage. Verify that the turtle randomly moves to every part of the stage given a sufficient number of jumps.

**Topic 3.3 Drawing Dots in Random Locations**

The **Move** block moves the turtle about the stage. For example, **Move 10** moves the turtle forward ten turtle steps. However, when the **Move 0** block is executed, the turtle stays in the same place and does not move forward. If the **Pen** is down, the turtle will draw a dot.

The **Set Pen Size** block controls the size of the dot.

**Move 10** moves the turtle forward ten turtle steps. However, when the **Move 0** block (i.e., “move 0 steps”) is executed, the turtle stays in the same place and does not move forward. If the **Pen** is down, the turtle draws a dot in that spot (as though a blob of ink squirted out).

The script that makes a dot can be refined by lifting the pen after the dot is drawn. This will ensure that the turtle only draws dots as it moves about the screen and does not draw lines that connect the dots when the turtle is moved.
A custom **Dot** code block can be created once this code is tested and verified. The **Dot** block, in turn, can be combined with the **Go to Random Position** block to place random dots across the screen.

![Image of code blocks and dots](image)

A **Repeat** block can be used to control the number of dots that are placed on the stage.

**Computing Concepts**

**Iteration.** The term *iteration* in computing refers to a repeated series of steps. In this instance, the **Repeat** block is used to repeatedly place the turtle at different positions on the stage.

**Exploration 3.3 Drawing Dots in Random Locations**

Create a **Dot** custom code block. Combine the **Dot** custom block with the **Go to Random Position** block to create random dots of varying sizes across the stage.

Estimate how many times the turtle has to make a dot in order to completely color in the entire stage. Then verify this estimate through repeated moves.

**Topic 3.4. Dots of Different Colors**

Random colors can be generated by assigning a random number as an input to the **Set Pen Hue** block.

![Image of code block](image)

This code can be used as the basis for a custom **Pick Random Color** code block.

If the **Pick Random Color** block is combined with the **Random Position** and **Dot** code blocks, this sequence of code blocks can be used to place random dots of different colors across the screen.
Computing Concepts

Creation of custom code blocks improves the readability of a program. Rather than displaying all of the lines of code contained within a custom code block, the reader sees the central concept or abstraction of the procedure’s function. This also facilitates re-use of code, which helps manage complexity.

Exploration 3.4 Dots of Different Colors

Create a Random Color custom code block. Combine the Random Color custom block with the Go to Random Position and Dot blocks to create random dots of different colors across the stage.

Topic 3.5 Polka Dot Backgrounds

A polka dot background can be created by using the Repeat block to generate the desired number of dots across the stage.

The Hide code block can be used to hide the turtle. Hiding the turtle speeds up the drawing process.

The Warp code block causes a polka dot background to appear almost instantaneously. The Warp block suspends other background operations until execution of the script is completed. This causes operations to take place more quickly.
The decision about whether to include the Warp block is an esthetic one in this instance. In some cases, it may be enjoyable to watch the dots appear one by one. In other cases, it may be satisfying to see the polka dot background appear instantly.

**Exploration 3.5 Polka Dot Backgrounds**

Create a **Polka Dot** custom code block that generates a polka dot background on the stage.

**Topic 3.6 Confetti Backgrounds**

A confetti background can be created by replacing the dots with randomly generated lines. A Line code block can be created by moving forward a specified number of steps (with the pen down) and then moving back the same number of steps.

Replacing the **Dot** code block with a **Line** code block produces a striped background.

Turning the turtle a random amount each time produces a confetti background.

The number of variants that can be created is endless. The choices for different variations is dictated by the esthetic sensibility of the coder.
**Exploration 3.6 Confetti Backgrounds**

Create a **Confetti** custom code block that generates a confetti background on the stage. Then explore other variants and patterns.

**Topic 3.7 Creating a Customized Color Palette**

There are times when it may be useful to create a customized color chart that includes a specific array of colors. For example, the chart below shows a sequence of colors and associated pen numbers associated with each color.

<table>
<thead>
<tr>
<th>Color</th>
<th>Pen #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>4</td>
</tr>
<tr>
<td>Orange</td>
<td>9</td>
</tr>
<tr>
<td>Gold</td>
<td>11</td>
</tr>
<tr>
<td>Yellow</td>
<td>17</td>
</tr>
<tr>
<td>Lime</td>
<td>20</td>
</tr>
<tr>
<td>Green</td>
<td>37</td>
</tr>
<tr>
<td>Cyan</td>
<td>48</td>
</tr>
<tr>
<td>Aqua</td>
<td>54</td>
</tr>
<tr>
<td>Blue</td>
<td>61</td>
</tr>
<tr>
<td>Purple</td>
<td>73</td>
</tr>
<tr>
<td>Violet</td>
<td>78</td>
</tr>
<tr>
<td>Magenta</td>
<td>87</td>
</tr>
<tr>
<td>Gray</td>
<td>66</td>
</tr>
</tbody>
</table>

The pen numbers associated with the colors selected can be placed in a list.

```
list 4 9 11 17 20 37 48 54 61 73 78 87 66
```

The list of pen numbers can then be assigned to a variable.

```
set Color Chart to list 4 9 11 17 20 37 48 54 61 73 78 87 66
```

The **Item** code block can then be used to select a random pen number from the color chart.

```
item 1 of Color Chart
```
The method can then be used to set the **Pen Hue** to a pen number randomly selected from a specific palette of colors.

**Exploration 3.8 Color Chart**

Create a customized color chart and combine it with the **Polka Dot** and **Confetti** programs to create art work in the style of Jason Pollock.
Appendix A

Code Blocks Introduced

A. Motion Palette

Go To X _ Y _

The Go To X _ Y _ code block combines the Set X and Set Y commands in a single code block.

B. Looks Palette

Hide and Show

The Hide and Show blocks are used to hide the turtle or show the turtle again if it hidden.

In some cases, the turtle can be useful as a guide to show the steps involved in creating a drawing. The turtle can create a drawing faster when it is hidden.

C. Operators Palette

Pick Random

The Pick Random block picks a random number between two specified values.

D. Control Palette

Warp

Code blocks placed within a Warp block will execute more quickly.

The increased speed is achieved by suspending execution of other scripts until the code within the Warp block is completed.
E. Variables Palette

Lists

Lists are one of the basic building blocks of Snap! A list can be used to organize a collection of related values. In the example below, the values associated with a range of colors have been placed in a list.

```
list 4 9 11 17 20 37 48 54 61 73 78 87 66
```

Lists are often assigned to variables. In the following example, the values associated with a series of colors have been assigned to a variable named *Color Chart*.

```
set ColorChart to list 4 9 11 17 20 37 48 54 61 73 78 87 66
```
Appendix B
New APCS Principles

Topic 3.2 Moving to a Random Position

CS Principle AAP-3.E Random Values

The code block Pick Random generates and returns a random number between two values (i.e., within a range that falls between a low number and a high number). Each result within this range is equally likely to occur. Generation of a random number in a program means each execution may produce a different result.

Application

In this section, Pick Random is used to send a turtle to a given random position on the stage.

Topic 3.3 Drawing Dots in Random Locations

CS Principle AAP-2.J Iteration

Iteration is a repeating portion of an algorithm. Iteration repeats a specified number of times or until a given condition is met.

Application

In this section, the Repeat block calls two different procedures. The programming instructions in the first procedure move the turtle to a random position on the stage. The programming instructions in the second procedure places a dot at the turtle’s location. In this section, the Repeat block iterates 100 times.

Topic 3.5 Polka Dot Background

CS Principle AAP-3.F Simulation

Simulations are abstractions of more complex objects or phenomena for a specific purpose. A simulation is a representation that uses varying sets of values to reflect the changing state of a phenomenon. Simulations often mimic real-world events with the purpose of drawing inferences, allowing investigation of a phenomenon without the constraints of the real world. The process of developing an abstract simulation involves removing specific details or simplifying functionality. Simulations can contain bias derived from the choices of real-world elements that were included or excluded. Simulations facilitate the formulation and refinement of hypotheses related to the objects or phenomena under consideration. Random number generators can be used to simulate the variability that exists in the real world.

Application

The blocks used in this topic allow the user to quickly iterate examples of art simulating the style of painter Jackson Pollock.

Topic 3.7 Creating a Customized Color Palette

CS Principle AAP-1.D Data Abstraction

Data abstraction provides a separation between the abstract properties of the data and the concrete details that the data represents. Data abstractions manage complexity in programs by giving a collection of data a name without referencing the specific details of the representation. Developing data abstraction within a program can make it easier to develop and maintain.
Application

A color palette is a collection of colors used to create art. In Snap/ each color is represented by a numerical value. Rather than referencing the number of each color as it is needed, users can group numerical values of colors into a list. That list can then be used as a color palette, with the artist calling individual items from the color palette instead of having to remember the numbers of all of the colors used.
Chapter 4. Simulating an Impressionist Painting

Glen Bull, Jo Watts, and Alexis Kellam

The image on the left is a photograph of a window box with flowers. The image on the right is an impressionist painting of the flowerbox created through a technique known as pointillism. This method creates a painting by placing tiny dots on the canvas to create an impression of the original painting.

In the previous module, dots were placed across the stage to create a polka dot background. The same method can be used to create an impressionist painting.

**Topic 4.1 Using a Sprite as a Canvas**

A photograph is imported into Snap!, where it becomes the costume of a sprite that serves as the canvas for the painting. This is the first module in which more than one sprite is used. One sprite will be used as the canvas for the painting. A second sprite will be used as a paint brush to create the painting on the canvas.

Begin by adding a new turtle sprite that will become the canvas.

Once the second sprite is created, change its name to *Canvas*. 
Once this is accomplished, two sprites will exist – the original sprite named “Sprite” and a second sprite named “Canvas.”

Select the *Costumes* tab of the *Canvas* sprite and import a photograph by dragging it into the space below the *Costumes* tab.

The photograph will become the new costume worn by the *Canvas* sprite. Drag the *Canvas* sprite to the center of the stage.
Once the Canvas sprite is centered on the stage, uncheck the “draggable” option so that the Canvas sprite cannot be moved from this position by accident.

**Exploration 4.1 Using a Sprite as a Canvas**

Select an existing photograph (or take a new photograph) and import it to use as the costume for the Canvas sprite. This photograph will be used as the basis for an impressionist painting.

**Topic 4.2 Using a Sprite as a Paint Brush**

The original sprite (by default named “Sprite”) will be used as a paint brush. Rename the original sprite, using “Paint Brush” as its new name. When this step is completed, two sprites will be visible in the sprite corral beneath the stage: one named “Paint Brush” and a second named “Canvas.”

The Canvas sprite may cover up the Paint Brush sprite on the stage. If that is the case, the Paint Brush sprite will not be visible. If the Paint Brush sprite is not visible, use the **Go to Front Layer** code block to bring the Paint Brush sprite to the front layer so that it is visible. (Note: The Paint Brush sprite must be selected in the Sprite Corral when this command is executed.)
When the *Paint Brush* sprite is in the front layer, the **Hue at Myself** reporter block can be used to identify the color of the canvas beneath the *Paint Brush* sprite.

The **Hue at Myself** reporter block can be combined with the **Set Pen Hue** block to change the color of the *Paint Brush* sprite to the color beneath the *Paint Brush* sprite.

This method can be extended to include *saturation* and *brightness* as well as the hue.

This will ensure that the *Paint Brush* sprite reflects the exact shade and color of the canvas beneath it. Once this code has been verified and tested, it can be encapsulated in the form of a custom code block, **Reflect Color**.

The **Reflect Color** custom code block causes the *Paint Brush* sprite to reflect the color of the canvas beneath it.
Computing Concepts

In the example above, a custom code block named Reflect Color replaced a longer sequence of instructions. Replacing an entire sequence of instructions with a named procedure yields several benefits. First, it makes the top level of the program more concise and easier to read. Second, if meaningful names are used, it makes the program easier to understand.

Exploration 4.2 Using a Sprite as a Paint Brush

Create a Reflect Color custom block that causes the Paint Brush sprite to reflect the color of the underlying Canvas sprite. Drag the Paint Brush sprite around the Canvas and observe the changes in its color when the Reflect Color code block is executed.

Topic 4.3 Painting the Canvas

After the Paint Brush and Canvas sprites have been created, the Reflect Color block can be combined with the Random Position custom block to move the paint brush to a random position on the stage, reflecting the color beneath the paint brush at the new position. The Paint Brush sprite is then used to place a dot at that position.

The Go to Front Layer block is involved prior to initiation of the Repeat block to ensure that the Paint Brush sprite is in the top layer (in front of the Canvas sprite) when the process is begun.

Once the code is created and tested, it can be used to create a custom Paint Canvas block. The Paint Canvas block can be repeated as many times as desired to create the desired impression of the underlying photograph.

Computing Concepts

In this section, the main Paint Canvas procedure calls the Go to Random Position, Reflect Color and Dot code blocks to paint the canvas. Through this process, a series of relatively straightforward procedures are used as building blocks to create more complex procedures. Testing each building block before it is used in another procedure makes the debugging process more efficient.

Exploration 4.3 Painting the Canvas

Create a Paint Canvas custom block that causes the Paint Brush sprite to create a pointillist painting. Explore the effect of invoking the Paint Canvas block with different dot sizes.
Topic 4.4 Further Explorations in Simulation of Impressionist Paintings

The **Paint Canvas** block created with the **Dot** block is just a starting point for exploration of impressionist effects. Just as the **Dot** block was replaced with the **Line** block in the previous module to create a confetti background, the **Dot** block can be replaced with the **Line** block in **Paint Canvas** to create a different effect.

Other possibilities include substitution of a starburst in place of a line or a dot. These substitutions can be used to create effects such as the impression of looking at the photograph through frosted glass. Explore the range of effects that can be achieved through different combinations and extensions.
Appendix A
Code Blocks Introduced

A. Looks Palette

Go to Front Layer

The **Go to Front Layer** block moves the currently selected sprite to the front layer of the stage.

B. Sensing Palette

Hue at Myself

The **Hue at Myself** reports the color beneath the currently selected sprite.
Appendix B
New APCS Principles

Topic 4.3 Painting the Canvas

CS Principle CRD-2.A The Purpose of Innovation

The purpose of computing innovations is to solve problems and pursue interest through creative expression. An understanding of the purpose of a computing innovation provides developers with an improved ability to develop that innovation.

CS Principle DAT-2.A Data Processing

Information is the collection of facts and patterns extracted from data. Data provides opportunity for identifying trends and making connections. Digitally processed data may show correlation between variables.

CS Principle DAT-2.D Data Extraction

Programs can be used to process data to acquire information. Some processes can be used to extract or modify information from a data set, such as by transforming every element of a data set, combining data, filtering data, or creating visualizations of data.

CS Principle AAP-4.A Algorithmic Efficiency

Efficiency is an estimation of the amount of computational resources used by an algorithm. An algorithm's efficiency can be measured by determining the number of times a statement or procedure executes before the desired result is achieved. Different correct algorithms can have different efficiencies.

CS Principle CRD-2.E Program Testing

A development process can be intentional or exploratory in nature. A development process that is iterative requires refinement and revision based on feedback, testing, and reflection throughout the process. A development process that is incremental is one that breaks the problem into smaller pieces and masks sure each piece works before adding it to the whole.

Application

In this module, a set task is creating Pointillist artwork from a photograph. In this task, the photograph contains all the data needed. Through an incremental development process, tools to extract and manipulate that data can be created. After determining the efficiency of the program, it may be necessary to use exploratory development to determine how efficiency can be increased. This may involve increasing or decreasing either the number of dots places on the canvas or the size of the dots.
5. Animated Art

Glen Bull, Jo Watts, and Rachel Gibson

Disney’s film, Fantasia, released in theaters in 1940, explored animated art. The renowned orchestra conductor, Leopold Stokowski, collaborated with Disney on development of themes based on “instrumental coloring.”

The film was recognized with a special Oscar at the Academy Awards for creating a new form of music visualization, thereby extending use of film and animation as an art form. Disney subsequently collaborated with a number of artists, including the surrealist Salvador Dali, on explorations in animated art.

**Topic 5.1 Animating Objects**

The programming language Snap! lends itself to exploration of abstract animated art. In the exploration that follows, a mosaic of shifting shapes and colors will be animated. This animation begins with selection of several shapes. The following shapes were created for the animation. Each shape was assigned to a different sprite.
A **Drift** procedure causes each shape to drift across the screen. The **Drift** code block picks a random point within the bounds of the stage, and causes the sprite to glide toward the specified point.

A parallel **Spin** procedure rotates the shape as it drifts across the stage.

If each of these blocks is attached to a **Green Flag** block, clicking the green flag in the upper-right hand corner of the stage will cause both procedures to run in parallel.

If these procedures are created in the script space for the sprites associated with each shape, clicking the green flag will cause each of the shapes to spin as they drift across the screen.

**Computing Concepts**

Procedures can run either sequentially or parallel to one another. Sequential procedures complete each block of code they contain in order and one at a time. Parallel procedures complete multiple blocks of code simultaneously to accomplish tasks that would be impossible with only sequential code. In this section, the **Green Flag** block runs the drift and spin procedures in parallel, while the code within those procedures runs sequentially.
**Exploration 5.1 Animating Objects**

Create shapes for each of three sprites. Develop procedures that cause the sprites to turn and move as they drift across the screen.

**Topic 5.2 Exploring Red-Green-Blue (RGB) Color**

Shifting the colors of the shapes as they drift across the screen emulates another element of Fantasia’s “instrumental coloring.” Each shape on the computer screen is constructed from tiny dots known as picture elements or pixels. The illustration below shows an image composed of four pixels magnified ten times. An image consists of a mosaic of pixels. This image consists of one black pixel, one red pixel, one green pixel, and one blue pixel.

The color of each pixel is controlled by values (ranging from 0 to 255) that specify the amount of red, green, and blue in the picture element. This method of specifying color is known as Red-Blue-Green (RGB) color. A custom code block, Set Color, can be used to set the sprite’s current costume to the red, green, and blue values supplied as an input to the procedure. In the illustration below, the values of 128 (Red), 255 (Green), and 255 (Blue) supplied as inputs to the procedure result in a cyan color.

Details of construction of the Set Color block are provided in the Appendix.

**Computing Concepts**

It’s important to make sure inputs to a procedure produce the expected outputs. In this section, the color of a sprite is determined by adjusting the red, blue, and green color values. These values are represented by a list of inputs. To test whether or not the block is working properly, the individual inputs can be set to values with specific known outputs. For example, if the red value is set to 255 and the other values are set to 0, the sprite should turn red when the Set Color block is activated. If the blue value is set to 255 and the other values are set to 0, the sprite should turn blue. Since red and blue make purple, setting both red and blue to 255 and the remaining input to 0 should cause the sprite’s color to change to purple when the block is activated. If the inputs to a block do not match the expected outputs, the procedures within the block should be re-evaluated for errors.

**Exploration 5.2 Exploring Red-Green Blue (RGB) Color**

Use the Set Color code block to change the color of a sprite’s costume.
**Topic 5.3 Adjusting Colors**

The colors obtained with varying combinations of red, green, and blue can be explored through creation of variables for each of these values. When the checkbox beside each variable is checked, a display called a *watcher* will show the current value of the variable on the stage.

Right-clicking the on-screen variable watcher accesses an option to create a slider control for the variable and assign minimum and maximum values to it.

When the *Set Color* code block is placed in a *Forever* loop with these variables assigned as inputs, adjusting the sliders controls the respective red, green, and blue values of the sprite’s current costume.

This method can be used to create an *Adjust Color* code block that can be used to quickly determine the relative amounts of red, green, and blue that must be used to achieve a desired color.
A user interface is a means by which the user and a computer system interact. Perhaps the most prominent aspects of the Snap! user interface is the manner in which code blocks can be snapped together in the coding area to create statements, procedures, and programs. When creating a program, it's often a good idea to consider how users will interact with it. For example, keyboards are a type of user interface that allow the user to directly enter text. Video game controllers allow the user to interact with a computer through a series of buttons and joysticks. In this section, variables are displayed as sliders, making it easier for users to adjust the color of the sprite. Each color could be adjusted manually, but employing a user interface with slider controls makes it easier for users to quickly adjust colors without having to spend time editing code directly. What are some other user interfaces you encounter in your life every day?

**Exploration 5.3 Adjusting Color**

Create a set of sliders that can adjust the red, green, and blue shades in a sprite’s costume. Identify a family of several colors that go well together, and record the RGB values for each of these colors.

**Topic 5.4 Varying Color and Movement**

Once the appropriate color ranges have been selected, the **Set Color** code block can be used to pick random colors for each of the shapes as they drift across the screen.

Fading the shape from one color to another as it moves across the screen is another method of varying color. This can be achieved by holding two of the RBG values constant while using a loop to vary the third color input. For example, if the red and green inputs are held constant (with Red set to a maximum value of 255 and Blue set to the minimum value of 1), varying the Green input from 1 to 255 will cause the shape’s color to fade from red to yellow.

**Exploration 5.4 Varying Color and Movement**

Create a block that fades a sprite’s costume from one color to another as it moves across the screen.
**Topic 5.5 Transparency**

A custom **Set Transparency** code block can be used to set an image to a desired level of transparency. The underlying details of this code block are described in the *Appendix*.

```
Set 50 Transparency (1 to 100%)
```

In the same way that sliders were created to adjust the red, green, and blue values of the pixels, a variable can be created with an associated slider to adjust the transparency of an image.

```
Adjust Transparency
```

The **Adjust Transparency** code block consists of a **Forever** loop that continually sets the transparency value of the image’s pixels to the value set by the *Transparency* slider.

```
\(\text{Adjust Transparency} + \)

forever

Set Transparency Transparency (1 to 100%)
```

In the same way that a procedure can be created to cause the shape to fade from one color to another (for example, from red to yellow) as it moves across the screen, a procedure can be created to cause the shape to fade from opaque to transparent as it drifts across the screen.

```
\(\text{Fade Transparency} + \)

forever

for \(i = 100 \text{ to } 1\)

Set \(i\) Transparency Transparency (1 to 100%)
```

**Exploration 5.5 Varying Color and Movement**

Create a block that fades the transparency of sprite’s costume from opaque to transparent as it moves across the screen.
Topic 5.6 Pulsing Shapes

Shapes can be created that appear to pulse by momentarily increasing the size of the shape and then decreasing the size a moment later.

An extension of this procedure can play a sound each time that the shape pulses.

Until now, the code blocks for each sprite have been placed in the script area for that sprite. However, it is also possible to create a single custom code block that issues commands to several sprites.

One of the advantages of this approach is that revising the code only involves editing the code in a single location rather than in several different places.

Computing Concepts

In the first section of this chapter, the Green Flag block was used to launch parallel procedures. In this section, shapes are made to pulse in a procedure that will be incorporated as part of a larger program. Since the procedure will be used in a program that's already running, the Green Flag
block cannot be used here. Instead, the **Tell** block alerts different sprites that it's time to execute their procedures in parallel with one another. These sprites will continue to run their procedures in parallel while the overall program continues to run its statements and procedures in series.

**Exploration 5.6 Pulsing Shapes**

Create a block that causes a sprite’s costume to pulse as it moves across the screen.

**Topic 5.7 Pulsing Sounds**

At the beginning of *Fantasia*, artists and musicians collaborated to play Bach’s Toccata and Fugue in D Minor synchronized with pulsing shapes that moved across the screen.

To create a similar effect in Snap!, a series of musical notes and phrases were imported into Snap!. In this instance, the musical phrases were imported into the sound files associated with the Stage. The first phrase was named “Toccata 1”, the second phrase in the musical sequence was named “Toccata 2”, etc.
Any of the imported sounds can be replayed by telling the Stage to play the desired sound; in the example below, the Stage has been asked to play the sound named “Toccata 1”.

In order to play the sounds in order as the shapes pulse sequentially (i.e., one after another), a variable named \textit{List of Sounds} containing the names of each of the sounds was created.

Telling the Stage to play the sound that is Item 1 in the \textit{List of Sounds} has the same effect as telling the Stage to play the sound named “Toccata 1”.

In order to move through the list while keeping track of the current sound, a second variable named \textit{Current Sound} was created. This variable is initially set to 1; each time a sound is played, the variable is incremented (i.e., increased by 1).

Once a method for incrementing the current sound has been developed, a \textbf{Play Current Sound} code block can be created that plays the current sound and then increments the current sound.
When the *Green Flag* is clicked to start the program, a **Pulse Shape 1** procedure pulses the shape and plays the current sound. It then sends a message (*Next Sound*) to Shape 2 asking it to play the next sound.

When *Shape 2* receives this message, the **Pulse Shape 2** code block pulses the shape, plays the current sound, and sends a *Next Sound* message to *Shape 3*.

After repeating the process, *Shape 3* sends a *Next Sound* message to *Shape 1*, completing the loop.

**Computing Concepts**

Relational operators compare the values of two inputs and then report whether the statement expressed is true or not. Relational operators contain the symbols =, ≠, <, >, ≤, and ≥. For example, if the integers 4 and 3 were put into the = operator such that the statement read 4=3, the operator would output the value “false”. If the statement instead read 4=4, the operator would output the value “true”.

Relational operators are often used in conditional statements. Conditional statements change the order in which procedures in a program run, depending on whether or not set conditions are met. For example, given the statement “if it’s raining, take an umbrella”, you would know when you’re supposed to take an umbrella. You would also know that if it’s not raining, an umbrella may not be needed. The same is true of computer code. When a procedure sees a conditional **IF** block, it will check to see if the conditions described in the operators section of the block have been met. If the conditions have been met, it will execute the code within the block before moving to the next
statement. If the conditions have not been met, it will skip the code in the block and continue with the rest of the procedure.

**Exploration 5.7 Pulsing Sounds**

Create a list of sounds. Develop a code block that causes each sprint in turn to play the next note in the list.
Appendix A

Code Blocks Introduced

A. Motion Palette

Glide

The Glide block causes the turtle to move smoothly to a specified position over a specified period of time.

![Glide block example](image)

B. Looks Palette

Pixels of Costume Current

The Pixels of Costume Current code block displays the Red, Green, and Blue (RGB) values for each pixel. In the graphic below, each of the four rows represents one of the four pixels in the image.

![Pixels of Costume Current example](image)

This code block can be combined with the Switch Costume code block to control the colors of the pixels in a costume. (See Appendix B.)

Change Size

The Change Size code block is used to change the size of the turtle.

![Change Size block example](image)

Sound Palette

Play Sound Until Done

The Play Sound until Done block plays the selected sound and waits until the sound is done.

![Play Sound until Done block example](image)
Control Palette

Tell Code Block

The **Tell** block tells a sprite or the stage to perform a specified action. This enables one actor such as a sprite to tell another entity such as the stage to perform an action.

![Tell block diagram](image)

Send and Receive Message

The **Send [Message]** block enables one sprite (such as Turtle 1 in the illustration below) to send a message to a second sprite (Turtle 2 in this example).

![Send and Receive Message](image)

When the second sprite (Turtle 2) receives the message, it performs the specified action (**Move 50 Steps** in this instance).
Appendix B

Adjusting the Color of a Sprite’s Costume

Picture Elements (Pixels)

Each shape on the computer screen is constructed from tiny dots known as picture elements or pixels. The illustration below shows an image composed of four pixels magnified ten times. An image consists of a mosaic of pixels. This image consists of one black pixel, one red pixel, one green pixel, and one blue pixel.

The color of each pixel is controlled by values (ranging from 0 to 255) that specify the amount of red, green, and blue in picture element. This method of specifying color is known as Red-Blue-Green (RGB) color.

RGB Colors

The **Pixels of Costume Current** code block displays the RGB values for each pixel. In the graphic below, each of the four rows represents one of the four pixels in the image.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>2</td>
<td>255</td>
<td>0</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>255</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>255</td>
<td>255</td>
</tr>
</tbody>
</table>

The first three columns (A, B, and C) control the red, green, and blue values of each pixel. The fourth column (D) controls the transparency of the pixel, ranging from completely opaque (255) to completely transparent (0). Because the first pixel, represented by Row 1, has values of 0 in each column, it is completely black (i.e., has no color). The second row has a maximum value of 255 in the first column (i.e., the red column) and 0 in the green and blue columns. Therefore, this pixel is completely red.
Changing the Colors of Pixels

The **Switch to Costume** code block can be used to change the values of pixels in the current costume. All of the elements in a list can be changed by arithmetic operators. For example, if one is added to a list of numbers from one through four, a second list of the numbers two through five results.

Therefore, if the first three columns of values for the picture elements are multiplied by zero, the red, green, and blue values for those pixels are set to zero. (Multiplying the fourth column controlling transparency by one leaves its values unchanged.)

Once multiplication is used to reset the color values to zero, the addition operator can be used to add the desired amounts of red, green, and blue color. (Adding zero to the fourth column controlling transparency leaves its values unchanged.)

The **Switch to Costume** code block can then be used to switch the pixels to the desired color values.
Values of 128 (Red), 255 (Green), and 255 (Blue) result in a cyan color.

This method of changing the value of a table of numbers is known as a matrix operation. Fortunately, the complexity of the underlying matrix operations involved in adjusting the color of a sprite’s costume can be encapsulated in the form of a custom code block.

The Set Color code block sets the sprite’s current costume to the red, green, and blue values supplied as an input to the procedure.

This makes it possible to adjust the colors of any costume to the specified RGB values.

**Adjusting Transparency**

In addition to specification of the red, green, and blue values for each pixel, a fourth parameter controls the transparency of the pixel. When the transparency parameter is set to 0, the pixel is completely transparent (i.e., invisible). When the transparency parameter is set to 255, the pixel is completely opaque. Values between 1 and 255 result in varying degrees of transparency. In the magnified image below, the checkboard squares that are white or gray are pixels that have a transparency value of 0. Pixels that are darker have a higher level of opacity.
To adjust the transparency of the image, an operation is required that leaves all of the pixels that have a value of zero unchanged, while adjusting the values of the pixels that have a non-zero value. This can be accomplished through multiplication. In the example below, the first three parameters (red, green, and blue) are multiplied by 1. This leaves their values unchanged. The fourth parameter, transparency, is multiplied by 255. This operation makes all of the pixels with a non-zero value complete opaque, while leaving the pixels which have a transparency value of zero untouched (since multiplying zero by 255 results in zero (i.e., $255 \times 0 = 0$).

Once all of the pixels with non-zero transparency values are set to maximum opacity, the desired level of transparency can be specified. In the example below, the transparency levels of the pixels have been set to 50%. Since multiplying 255 times 0.5 results in a rounded result of 128, the resulting level of transparency is halfway between completely opaque and completely transparent.

The **Set Transparency** code block employs this method to enable a sprite’s costume to be set to a specified level of transparency.

In the example below, the transparency level of the image has been set to 50 percent.

The advantage of creating a custom **Set Transparency** code block is that it is no longer necessary to remember the details of the internal pixel structure and parameters while focusing on other aspect of the program.
Appendix B
New APCS Principles

Topic 5.1 Animating Objects

CS Principle CSN-2.A Parallel Computing

Parallel computing is a method of computing where the program is broken into multiple smaller computing operations, some of which are then performed simultaneously. This is different from the programs used in previous modules, which relied on sequential computing. In sequential computing, operations are performed in order and one at a time.

Application

In Snap!, the Green Flag block can be used to start multiple programs or statements in parallel with one another. With parallel computing, multiple sprites can run multiple code segments at the same time, without having to wait for each other to finish. This would be impossible using only sequential computing.

Topic 5.2 Exploring RGB Color

CS Principle CRD-2.J Identify Expected Outputs

In the development process, testing uses defined inputs to ensure that an algorithm or program is producing the expected outcome. Programmers use the results from testing to revise their work. Defined inputs used to test a program should demonstrate the different expected outcomes that are at the extremes of input data.

Application

When adjusting RGB color settings, each variable can be set to its minimum value (0). If the variables are changed one at a time to their maximum values (255), students should be able to accurately predict resulting color of the sprite. Once students have verified that the variables produce the desired results independently, they can try adjusting two or more sliders to try blending colors. This will illustrate the importance of verifying that a program is working as expected with regard to its inputs and outputs.

Topic 5.3 Adjusting Color

CS Principle CRD-2.F User Interface

When a program is being developed, it’s important to understand not only the program, but also the concerns and interest of the people who will use it. Program requirements describe how a program functions and may include a description of user interactions that the program must provide. Some ways to investigate these needs include user testing, interviews, and direct observation.

Application

In the previous section, it was demonstrated how colors can be changed by adjusting the numerical values of the RGB settings. In that section, these adjustments were made manually. As the procedure to set the color gets integrated into the larger program, it may be easier for users if they didn’t have to directly edit the code to change colors. In this module, slider controls are created to allow users to more easily adjust sprite colors.

Topic 5.7 Pulsing Sounds

CS Principle AAP-2.E Relational Operators

Relational operators include the following symbols: $=, \neq, <, >, \leq, \geq$. These are used to test the relationship between two variables, expressions, or values. A comparison using a relational operator
evaluates to a Boolean value. For example, \( a = b \) evaluates to true if “\( a \)” and “\( b \)” are equal; otherwise, it evaluates to false.

Application

In this topic, the greater than symbol is used to determine whether the length of the input list has been exceeded. If the length of list has been exceeded, the procedure begins again with the first item in the list.

CS Principle AAP-2.H Conditional Statements

Conditional statements, or “if-statements”, affect the flow of a program by executing different statements when different conditions are met. If the condition of the if-statement is determined to be true, the code included in the if statement will be executed. If the condition is determined to be false, the program or procedure will ignore the if statement and continue to the next statement in the code sequence.

Application

In this section, a procedure repeats a sequence of sounds using a \textbf{For} loop. Sounds are numbered sequentially, and the code in the loop plays the sounds in order. To make the sequence repeat, a conditional statement is added, telling the procedure that if the number of the sounds set to play is greater than the number of sounds that exist, it should start back over with the first sound.
6. Combining Art and Music

Glen Bull, Jo Watts, and Rachel Gibson

The film, *Fantasia*, begins with a series of abstract swirling images with a conductor leading an orchestra playing Bach’s *Toccata and Fugue in D Minor*. Animated shapes and abstract images move in time to the music.

In the previous module, *Animated Art*, individual sounds such as chimes or heartbeats were played in time with pulsing shapes. Playing an entire music composition requires an approach that is more efficient than recording sound samples for each individual note in a music composition.

TuneScope ([www.TuneScope](http://www.TuneScope)) is a code-based Digital Audio Workstation that can be used to compose and play music. For those interested in the technical details, TuneScope is an extension of the educational programming language Snap! (developed at the University of California, Berkeley) that uses the W3C Web Audio application interface (API) to generate musical notes and instruments.

**Topic 6.1 The Building Blocks of Music**

An understanding of the basic building blocks of music is helpful in creating music in TuneScope. The Western chromatic musical scale consists of twelve notes. On a piano, these notes consist of seven white keys and five black keys. Pressing a piano key causes a hammer to strike a piano string, which vibrates at a rate that is specific to each note. For example, the note *C* in the middle of the piano keyboard vibrates at a rate of approximately 261 times per second. Because this note is in the middle of the piano keyboard, it sometimes called *Middle C*. 
Higher rates of vibration are perceived as higher pitches. After the twelve notes of the scale are played, the pattern repeats itself. In music the twelve notes of the chromatic scale are referred to as an octave. Middle C begins the fourth octave on the piano keyboard. The next C in the sequence begins the fifth octave. Therefore, Middle C is sometimes written as C4 and the C that begins the next octave can be written as C5. The note C5 vibrates at twice the rate of C4. A piano keyboard that can be used to explore the fourth and fifth octaves can be accessed at:

https://tunescope.org/index.html#project:Username=make2learn&ProjectName=Piano

Pressing a piano key on the computer screen plays the note associated with that key.

When written, notes usually appear on a series of five parallel horizontal lines, known as the staff. Notes are grouped into measures. A measure consists of the notes between two vertical lines on the musical staff.

The time signature is the convention used in Western musical notation to signify the number of beats in a measure (also referred to as a musical bar) and to describe which note value is counted as a beat. The time signature is written in the form two numbers stacked on top of each other: 4/4, 3/4, etc. The first number specifies the number of beats in a measure and the second number indicates the value of the note that is counted as a beat. For example, a 4/4 time signature indicates that there are four beats per measure and that each beat is a quarter note. A 3/4 time signature indicates that there are three beats per measure and that each beat is a quarter note. A 6/8 time signature indicates that there are six beats per measure and that each beat is an eighth note. A 4/4 time signature is sometimes described as common time because it is one of the most common time signatures.

Modern Digital Audio Workstations, such as Ableton, allow musicians to record and edit songs with the notes for each instrument represented by separate tracks. A track is generally the music for a single instrument. These tracks can then be mixed together to create a full song. TuneScope also can be used to create and play tracks of music created using computer code in the form of Snap! code blocks. This module introduces the way that individual musical notes are represented and sequenced in TuneScope. The next module explores groups of musical notes known as scales. The modules that follow after that explore musical chords, drum tracks, and construction of musical lyrics.

Exploration 6.1 The Building Blocks of Music

Use the piano keyboard created with sprites to explore the relationship of notes within the fourth and fifth octaves of the chromatic scale.
6.2 Musical Notes

TuneScope can be used to access sampled instruments that can be used to play musical notes. These musical instruments are selected through the **Set Instrument** code block. This code block enables a musical instrument to be selected from a dropdown menu.

Once an instrument has been selected, the **Play Note** code block can be used to play a musical note.

A dropdown menu in the TuneScope **Play Note** code block can be used to select musical notes. (Note designations can also be typed directly into this input slot.)

Once a note has been selected, a second dropdown menu is used to select the duration of the note. The note duration determines the length of time that the note is played.

In most western music, a *quarter note* is considered to be one beat. Moving up the menu doubles the number of beats in the note. Moving down the menu halves the length. For example, a half note is two beats, while a whole note is four beats. In the other direction, an eighth note is half a beat. Dotted notes equal the duration of the named note plus the duration of the next shortest note. For example, a dotted half note would be equal in duration to a half note plus a quarter note, or three beats. Triplet
notes divide the duration of two of the named notes into three equal durations. For example, a quarter note triplet would be three evenly timed notes played within the normal duration of two quarter notes.

**Exploration 6.2 Musical Notes**

Explore the voices of different musical instruments that can be accessed through the **Set Instrument** code block. Use the **Play Note** code block to play back notes using different instruments.

**Topic 6.3 Combining Notes**

The **Play Note and Wait** code block waits until one note is completed before beginning the next note. Several **Play Note and Wait** code blocks can be combined to play a series of notes. For example, the opening notes of the blues song *Crossroads* are G♯5, A♯5, and A♯5

![Play Note and Wait](image)

The first note (G5) lasts for a sixteenth note, the second note (A#5) lasts for one-eighth note, and the third note (A#5) lasts for a quarter of a note. The music notation “♯” is pronounced “sharp”. For example, the note A# is pronounced, “A Sharp”. The sharp keys are the black keys on a piano keyboard. (Note: “A♯” can also be written as “B♭”, pronounced as “B Flat”.)

**Exploration 6.3 Combining Notes**

Create your own three-note sequence in TuneScope using the **Play** code block. You can either use notes from a favorite song or create your combination of notes.

**Topic 6.4 Playing a List of Musical Notes**

Since the music visualization in *Fantasia* begins with *Toccata and Fugue in D Minor*, this seems like an appropriate place to begin with musical visualization in Snap! The **Set** command can be used to create a variable with a list of notes.

![Set List of Notes](image)

Once the variable has been created, the list can be edited and additional notes can be added by directly hand editing the list.
A loop can then be constructed to play each of the notes in the list.

Computing Concepts

When working with a list, the For loop block makes it possible to repeat a procedure multiple times using a different item from the list as the input for each iteration. For example, in this section, six Play Note and Wait blocks could have been assembled in sequence to play the six notes in the list. Instead, the For loop block simplifies the process by only requiring one Play Note and Wait. Each time the loop repeats, the variable \(i\) changes by 1, causing the procedure inside the loop to play the next note in the list. After \(i\) equals the length of the list (6), the loop stops.

Exploration 6.4 Playing a List of Musical Notes

Create a list of notes. Then use the Play Note List code block to play the list of notes.

Topic 6.5 Creating a Musical Visualization that Pulses with Sound

All of the musical tools needed to create a musical visualization that pulses with sound have now been assembled. In the previous module, the Pulse code block increased the size of each shape and then decreased each one after a short delay. The Pulse Shapes code block then told each shape to pulse in turn.
The updated version replaces the code block **Pulse** with the code block **Pulse Shapes with Sound**. The **Pulse Shapes with Sound code** block replaces the delay in **Pulse** with a custom code block, **Play Current Sound**.

A variable, **Current Note**, tracks the current note in the list of notes played. A **Setup** procedure is run and sets the **Current Note** to 1 when the program begins.

The **Play Current Note** code block plays the current note in the list of notes.

The **Increment Current Note** code block then increases the value of the variable **Current Note** by 1. If the end of the list of notes is reached, the variable is reset to 1.

**Exploration 6.5 Creating a Music Visualization that Pulses with Sound**
Create a music visualization in which each shape plays the next note in a list of notes

**Topic 6.6 Recording a Sound Sample**

The computer’s microphone can be used to capture a sample of a naturally occurring sound. A sound recorder in Snap! is accessed through the *Sounds* tab to the right of the *Scripts* tab. (Note: on a Macintosh, you will be asked to give permission to access the computer’s microphone the first time the sound recording capability is accessed.)

Once a sound has been recorded, it can be assigned any name that is appropriate. In this instance, a ringing sound was created by tapping a fork on a water glass.

Additionally, any sound saved as a .wav or .mp3 file can be imported into Snap! by dragging into the area below the *Sound* tab.

**Exploration 6.7 Recording a Sound Sample**

Use the Snap! *Sound Recorder* to record a sound sample. Then import a sound acquired from FreeSound or a similar sound repository.
Appendix A
TuneScope Blocks Introduced

Set Instrument

The **Set Instrument** block is used to select a musical instrument.

Play Note

The **Play Note** code block is used to play a musical note.

Play Note and Wait

The **Play Note and Wait** code block waits until one note is completed before beginning the next note. Several **Play Note and Wait** code blocks can be combined to play a series of notes.
Appendix B
New APCS Principles

Topic 6.4 Playing a List of Musical Notes

CS Principle AAP-2.O Iterative Procedures that Traverse a List

When a procedure uses a list as an input, each iteration of that procedure can use a different item from that list to produce a corresponding output. Traversing a list can be a complete traversal, where all elements in the list are accessed, or a partial traversal, where only a portion of elements are accessed.

Application

In this topic, a variable is created and set to be an editable list of notes. A procedure then iterates to play each note within the list in sequence order.
7. Musical Scales

Glen Bull, Jo Watts, Rachel Gibson, and Luke Dahl

The term musical scale refers to a sequence of notes. Many different musical scales have been developed throughout history. The Western chromatic musical scale consists of twelve notes. Other scales have been developed that employ different numbers of notes.

The overwhelming majority of popular music today is composed using groups of notes drawn from within the notes of the chromatic scale. The keys of today’s pianos are tuned using this scale.

### Topic 7.1 The Twelve-Note Chromatic Scale

A span of notes on the piano keyboard that begins with one note (such as the note “C” in the illustration below) and ends in the same note is known as an octave. The octave that begins with middle C is in the fourth octave (counting from the left) on the keyboard.

Therefore, middle C can be written as “C4”. In this example, the letter refers to the musical note and the number refers to the octave. A custom **Chromatic Scale** code block can be created to report the twelve notes of the chromatic scale.
This code block will be used for construction of major and minor scales that are subsets of the chromatic scale.

**Exploration 7.1 The Twelve-Note Western Chromatic Scale**
Create a custom code block that uses the Chromatic Scale code block described in this section as an input to play the notes of the scale in order. Modify the code block so that it plays the notes of the scale in order beginning with the note “E”.

**Topic 7.2 Major Scales**
Many other scales have been formed that are subsets of the full twelve notes of the chromatic scale. For example, some series of seven-note scales formed from the full twelve-note chromatic scale are known as major scales. These seven-note consist of notes that are judged to be harmonious in combination with one another.

The C Major scale consists of the seven white keys. These keys are numbered 1, 3, 5, 6, 8, 10 and 12 in the illustration below.
There are twelve major scales, one for each note in the chromatic scale. The notes for each major scale are always drawn from the same relative position in the chromatic scale. For example, the notes that form the D Major scale are drawn from the same relative positions of the chromatic scale. This scale consists of the notes numbered 1, 3, 5, 6, 8, 10, 12 (beginning with D).

In other words, the C Major scale was formed from the seven white notes on the piano keyboard. These notes are judged to sound harmonious in combination with one another. Once this initial selection was made, the pattern was repeated for each of the subsequent notes in the chromatic scale, creating the D Major scale, the E Major scale, etc.

Using the same pattern of notes (1, 3, 5, 6, 8, 10, 12) means that while the absolute frequencies shift as the scales change, the relative frequencies of one note to another within a scale remain the same. This makes it easy for musicians to transpose tunes from one scale to another, and facilitates musical exploration and invention.

From a computing standpoint, this regularity also facilitates creation of a custom code block that reports the seven notes in any major scale. For example, the notes of the C Major scale in the fourth octave are shown in the illustration below.

For those who are interested, details of how the Major Scale code block is constructed are provided in the appendix of this module.

Computing Concepts: Strings as Variables

In computing science, a string is an ordered sequence of characters. As opposed to a list, where each item in the list is a separate value, a string acts as a single value. For example, "(A, B, C) is a list, but "ABC" is a string.

In TuneScope, scales are represented by lists of notes, but notes are represented by strings of characters.
**Exploration 7.2 Major Scales**

Create a custom code block that plays the first, third, and fifth note of a major scale. For example, in the C Major scale, the notes C, E, and G would be played. Then modify the procedure so that it can play the same sequence of notes in any major scale. Try playing this note sequence in several scales. What is perceived as different when the scale changes? What does not appear to change when the scale changes? Try playing three note sequences from several scales one after the other; for example, C, E, and G followed by F, A, and C followed by G, B, and D.

**Topic 7.3 Minor Scales**

Much of the music in medieval times was commissioned and performed in churches and monasteries. A primary goal in commissioning this work was to create music that sounded as harmonious (i.e., heavenly) as possible. However, in contemporary music, songs that only included harmonious combinations of notes would soon be considered saccharine and boring.

Other scales, such as the minor scales, can result in music that can be more dissonant than the major scales. This lends music tension, drama, and interest. Minor scales are formed by replacing the fifth, eighth, and tenth note of the chromatic scale with the fourth, nineth, and eleventh notes. Just as with the major scales, although the absolute frequencies of notes in each minor scale shift, the relative positions of frequencies of notes in the minor scale remain the same.

These notes of the minor scales form ratios that are higher than the notes that form the major scales. Hence these scales are perceived as more dissonant than the major scales.

The **Minor Scale** code block reports the minor scales for each of the twelve notes in the chromatic scale.
Details of the construction of the **Minor Scale** code block are provided in the *Appendix* of this module. This code block is used in construction of a code block that reports the minor chords, described in the following section.

**Exploration 7.3 Minor Scales**

Create a custom code block that plays the first, third, and fifth note of a minor scale. For example, in the C minor scale, the notes C, Eb, and G would be played. Then modify the procedure so that it can play the same sequence of notes in any minor scale. Try playing this note sequence in several scales. How does the note sequence in the minor scale compare with the previously played note sequence in the major scale?

**Topic 7.4 Composing through Selection of Random Notes within a Scale**

The art modules (i.e., the first six modules) in the Art & Music sequence emulated the styles of different artists – such as the post-impressionist artist, Georges Seurat, and the twentieth century artists Mark Rothko and Jackson Pollock – through constrained randomness. For example, in the illustration below, the color, size and transparency of randomly placed dots and lines have been used to emulate the style of Pollock.

![Illustration](image)

In this example, a pleasing pattern has been achieved by constraining randomness. The creator has chosen to focus on the cooler portion of the color palette, primarily using blue and green colors. The key to creating engaging art in this manner involves constraining the colors and shapes in some manner.

In the same way that the color, size and transparency of dots and lines were randomly generated within constrained boundaries to create art in the style of Jackson Pollock, musical styles can be emulated by selecting random musical notes within constrained boundaries. Musical sequences can be created by randomly selecting notes from a constrained group of notes. The eight notes selected for a given scale (i.e., the C major scale, the D major scale, etc.) are ones that are likely to sound harmonious when played in combination with one another. Therefore, randomly selecting notes from a given scale has a higher likelihood of creating pleasing combinations than notes randomly drawn from all of the possible combinations.

The **Item Random** code block can be used to randomly select a note from a specified scale. In the example below, the note B4 has been randomly selected from the C Major scale.
Notes randomly selected from notes in the C major scale can then be assigned to a list of notes.

The preceding code can be used to construct a **Select Random Notes** block that can be used to quickly generate a list of notes randomly selected from the C major scale. This randomly generated list can be further edited by hand if desired to refine the sequence.

A loop similar to ones used a number of times in the art modules can then be used to play the list of notes generated in this manner.

Once the loop is constructed and verified to work properly, it can be used as the basis for a custom **Play Notes** block.
The **Select Random Notes** procedure can be modified to select notes from different scales. Many other possible groupings of notes can be explored in this manner. For example, the genre of music known as the blues often incorporates sharp notes and flat notes to create a melancholy sound. Groups of notes that include a number of sharps and flats could be used as a basis for selection.

Thus far, each note has been the same duration. The **Select Random Note** code block can be further enhanced by adding random durations to a **List of Durations**.

**Exploration 7.4 Composing through Random Selection of Notes within a Scale**

Create a custom code block that constrains the randomness beyond simply picking notes from the C Major scale by creating certain rules for picking the notes. Set up your block so that every time an E is selected, the next note picked will be D. In addition, every time a C is selected, the next note will be E. Compare how your sequences sound before and after you constrain the randomness. What other rules could be added?

**Topic 7.5 Direction and Movement of Musical Sequences**

The tune generator that you have developed approximates melodies that a composer might create. However, the random nature of the notes generated results in discontinuities between notes. When composers create music, they consider another element – musical motion. **Stepwise** motion is one type of musical motion. This type of motion occurs when one note moves directly to the next neighboring note. In a C Major scale, C moving to D or G to A is stepwise motion. **Skipwise** motion is another type of musical motion. Skipwise motion occurs when a note moves to another note that is not adjacent. In a C Major scale, C moving to G or C to A are examples of skipwise motion.

The **Interval between Note 1 and Note 2** reporter block shown below reports the number of steps between two notes within a major scale. (Details of the construction of this code block are provided in the *Appendix*.) For example, there is one step in the musical interval between C and D in the C Major scale.

However, there are four steps in the musical interval between C and E.
The musical intervals between a series of notes can be easier to visualize if they are graphed. If the turtle moves zero steps, it creates a dot the same size as the pen width at the location of the turtle.

The **Graph Notes** procedure below sets the pen color to red when the musical interval between two notes in a major scale exceeds a specified threshold.
This causes musical skips between two notes to be drawn in red. In the graph below, the notes in the first half of the sequence are drawn in blue, while several of the notes in the second half are drawn in red.

Once you have generated a melody that you like, you can enhance it by directly editing the list of notes created. When you look at your own melody, identify any steps or skips between consecutive notes. If you see many skips in your melody, consider changing them to be steps (for example, C to G might become C to D).

The direction of motion is another element to consider in musical composition. Often composers create a series of notes that move in a specific direction – ascending or descending – to create a sense of musical motion. Consider the direction of your melody’s pitch. Does the pitch get higher more often than lower? In the graph above, the pitch is ascending in the first half of the sequence. In the second half of the sequence, the notes are alternating between high and low pitches. In cases like this, consider adding or removing notes to create a general direction for notes to travel. Perhaps the melody gets gradually higher in pitch, then lower in pitch, and eventually ascends. Editing your melody to incorporate musical motion can make it more dynamic.

Once you have explored creation of musical motion by hand editing the lists of notes generated, consider how you might refine the music generation code to incorporate musical motion in the notes generated.

**Exploration 7.5 Direction and Movement of Musical Sequences**

Create a custom code block that randomly selects 10 notes that only ascend through a combination of skips and steps. Then, modify this code block so that after 10 ascending notes are selected, 10 more notes are randomly selected in a descending motion, either through skips or steps.

**Topic 7.6 Music Intervals within a Scale**

Composers often create tunes consisting of notes that are primarily drawn from within a major or minor musical scale. In these cases, it is convenient to reference the numbers of the notes within that scale. The C Major scale, for example, consists of the seven white keys on a piano, numbered 1 through 7 in the illustration below.
The first, third, and fifth notes within any major or minor musical scale sound particularly harmonious when played in combination with one another. For example, C, E, and G in the C Major scale sound harmonious when played in combination with one another.

The **Major Scale Note Position** block provides the note associated with any position in any major scale. A parallel block provides this information for the minor scales.

This block can be used in combination with the **Play Note and Wait** block to play the selected note.

**Computing Concepts:** Strings as Variables

*Indexing* is used to quickly access data from a list. Each item in a list has both a value and an index number. In this section, the notes in a scale are the values, but their positions in the scale are their index numbers. For example, the C Major scale is a list of seven notes. The note E has an index number of 3, since it's the third note in the list.
**Exploration 7.6 Musical Intervals within a Scale**

Create a custom code block that randomly selects notes of different positions from several major scales. Then, try the same procedure with several minor scales. Afterwards, within your code block, aim to incorporate skips, steps, and ascending and descending motion, similar to Exploration 7.7.

**Topic 7.7 Major Triads**

In the previous section, the first, third, and fifth notes were identified as ones that are particularly harmonious when played in combination with one another. This sequence of three notes will always sound harmonious in any of the twelve major scales or any of the twelve minor scales.

The regularity of the structure of musical notes and scale can be combined with the strength of code to make use of this regularity. A **Major Triad** code block can be built to report the first, first, and fifth notes in a major scale.

For example, the first, third, and fifth notes of the C Major scale in the fourth octave are: C4, E4, and G4.

The **Play Notes** code block (introduced in Topic 8.7 above) can be used to play the triad of notes generated by the **Major Triad** reporter block.
Many songs are composed with four beats per measure. Four quarter notes in one measure is known as 4/4 time. This means that there are four quarter notes in a measure. To fill out the fourth beat in the measure, a fourth note can be appended to the triad.

It is common practice, for example, to repeat one of the notes in a triad in a higher octave to create a fourth note.

A parallel Minor Triad reporter block can be created that reports the first, third, and fifth notes in a minor scale.

**Topic 7.8 Scale Progressions**

In the same way that certain notes within a scale sound harmonious in combination with one another, certain combinations of scales also sound harmonious in combination with one another. For example, a progression involving the C Major scale, the F major scale, and the G major scale has been used in many popular songs. There are many variations such as a progression of the major triads (first, third, and fifth notes) of these scales.

Commonly used scale progressions that begin with the C Major scale include the following:

- C Major - G Major – F Major
- C Major – F Major – G Major
- C Major – F Major – A Minor – G Major
- C Major – G Major – A Minor – F Minor
There are also several commonly used scale progressions that begin with the A Minor scale:

A Minor – F Major – C Major – G Major

Exploration 7.8 Scale Progressions

Create and play a sequence of notes using one of the scale progressions described in the section above.

Topic 7.9 Composing a Track

Musical tracks can be used to combine sequences of notes to create a song. The **Note** reporter block is the basic unit. Notes can be combined into *measures*. Measures, in turn, are combined to form *musical tracks*.

The **Note** code block, labeled with a musical note symbol “♩”, consists of a note and a duration.

The **Measure** block is used to group sequences of notes. (A musical measure consists of the notes within one bar of music.)

In the most common time signature, 4/4 time, there are four quarter notes within a measure. Since four quarter notes equal one whole note, the combined durations of all of the notes within a measure must equal a whole note.

The **Validate Measure** block can be used to determine if the combined durations of notes within a measure are correct.

If the durations are not correct, the **Validate Measure** will report the discrepancy.

Completed measures are assembled into tracks.
The **Play Tracks** block is used to play music tracks. The example below consists of a single track, but in most cases several tracks with different musical instruments would be played in parallel.

For example, a piano track and a guitar track are played in parallel in the example below.

**Topic 7.10 Sections**

When a group of measures will be repeated several times (for example, in the chorus of a song), they can be grouped into sections using the **Section** block.

If a section is very long, it may be useful to place the section within a named reporter block.
Once sections have been named, they can be repeated within a track. This avoids duplication and makes the structure of the music easier to follow.

**Topic 7.11 Backing Tracks**

A backing tracking is often combined with a melody to create a song. Several guidelines may be helpful in constructing an initial melody track to accompany the backing track.

1. Select notes in the same scale as the notes in the measures of the backing track. However, pick a different octave than the octave used for the backing track.

2. Begin with an ascending or descending sequence from that scale that does not have jumps or skips.

3. Once you have an initial sequence for a given measure, edit the notes by hand until you are satisfied with the results. You can also hand edit notes in the backing track as needed to obtain a result that is pleasing to you.

4. Often the notes in the melody are of shorter duration than the notes in the backing track. For example, if the notes in the backing track are quarter notes, the notes in the melody might be eighth notes.

These suggestions are just a starting point. As you gain more experience creating music you will get a better sense of your own musical taste.

Timing also is a significant element of music. In 4/4 time, there are four quarter notes in a measure. The notes in each measure of the melody must add up to a whole note, which is equivalent to two half notes, four quarter notes, or eight eighth notes.

The initial melody track created likely will have a mechanical sound, much like a mechanical music box. This is in part because to simplify construction of the melody, the notes have all been the same duration. Varying the duration of the notes of the melody can help create a more natural sound. The only constraint is that (in 4/4 time), the duration of the notes in each measure must add up to a whole note. Further discussion of timing is provided in the *Drum Kit* module.

**Exploration 7.11 Backing Tracks**

Use the techniques described in this module to create a melody and an accompanying backing track. Use the **Play Tracks** block to play the two tracks together.

**Topic 7.12 Musical Patterns**

Almost all music incorporates patterns that repeat. Sometimes the repetition involves repeating patterns of notes. Repetition of lyrics is also a recurring element of music.

Today technology often plays a role in creating and extending musical patterns, sometimes producing effects that would be difficult to achieve otherwise. The musical pioneer Les Paul began experimenting with use of tape recorders to achieve these types of effects shortly after the second World War. Les Paul recorded a section of music on an audio tape. He then played back the music and recorded it on an acetate record. He could then play the record and overlay the original music.
track with a second layer of music. Les Paul later developed multitrack tape recorders that enabled him to achieve similar effects within a single tape recorder.

The timing and juxtaposition of the overlaid music tracks enabled Les Paul to achieve various echo and reverberation effects. Digital computers are now used to achieve similar effects.

A musical canon is a piece of music in which the instrumental parts or vocals play or sing the same music starting at different times. (A round is a type of canon. However, in a round each voice starts at the beginning again when it finishes so that the piece goes “round and round.”)

Pachelbel’s Canon is one of the most famous examples of this type of musical repetition. A violin begins to play the music. After a delay of two measures, a second violin begins to play the same music. After a further delay of two additional measures, a third violin begins to play the same music.

The same sequence of notes is repeated three times by three violins in Pachelbel's Canon.

The effect of doubling and tripling the music with an offset greatly enhances the effect of the music. Although Pachelbel’s Canon was composed in 1694, this music is still popular and is often played at weddings today.

In one demonstration, Vihart used three paper tape music players to graphically illustrate the delayed repetition of the music in Pachelbel’s Canon. As a paper tape encoding the musical notes passed through each of three music box play heads, the same music was repeated with a delay.
Pachelbel also employed repetition in the musical chords that accompany the melody. When a violin trio plays Pachelbel’s Canon, a cello traditionally plays the chords that accompany the melody.

**Measure 1**: D Chord, A Chord, B Minor Chord, F# Minor Chord  
**Measure 2**: G Chord, D Chord, G Chord, A Chord

The core set of violin notes can be offset by inserting rests at the beginning of the lists of notes for each violin.

The violin tracks with the specified number of offsets for each violin can then be combined the repeating cello track, achieving essentially the same result in TuneScope as Vihart achieved with paper-tape music boxes.

The **Tempo** block shown in the example above can be used to set the beats per minute of the tune. **Tempo** refers to the overall speed at which a song is played. The tempo can significantly change the way that a song is perceived.

**Exploration 7.12 Musical Canons**

Recreate the melody for Pachelbel’s Canon in TuneScope using multiple violin tracks with offsets. Then modify the composition to create a variation of the Canon.
Appendix A
TuneScope Blocks Introduced

**Chromatic Scale**

The **Chromatic Scale** block reports the twelve notes of the chromatic scale.

![Chromatic Scale Diagram]

**Major Scale**

The **Major Scale** block reports the seven notes of a major scale in a specified octave.

![Major Scale Diagram]
**Minor Scale**

The **Minor Scale** block reports the seven notes of a minor scale in a specified octave.

![Minor Scale](image)

**Interval between Note 1 and Note 2**

The **Interval between Note 1 and Note 2** reporter block reports the number of steps between two notes within a major scale.

![Interval between C4 Note 1 D4 and Note 2 in the C Major Scale](image)

**Major Scale Note Position**

The **Major Scale Note Position** block provides the note associated with any position in any major scale. A parallel block provides this information for the minor scales.

![Major Scale Note Position](image)

**Note**

The **Note** block reports a note paired with a note duration.

![Note](image)

**Measure**

The **Measure** block reports a group of notes with their corresponding durations.

![Measure](image)
**Validate Measure**

The **Validate Measure** block checks to make sure the notes in the measure are properly formatted. In then checks to make sure that the total durations of the notes in the measure match the time signature. If the note durations do not match the time signature, the **Validate Measure** block reports how many beats too long or too short the measure is.

**Section**

The **Section** block reports a list of measures.

**Track**

The **Track** block groups a set of notes and assigns an instrument to play them.

**Play Track**

The **Play Tracks** block can then be used to play the tracks created in this manner.
Appendix B

Constructing Code Blocks to Report the Notes in the Major and Minor Scales and the Intervals between Notes within a Scale

Major Scales

The major scales consist of the following seven notes within the twelve-note Western chromatic scale: 1 3 5 6 8 10 12. A major scale cannot have two notes consisting of the same letter. For example, if sharps were used in the F major scale, the letter “A” would be used twice:

F G A A♯ C D E

Therefore, “B Flat” is used in place of “A Sharp” to avoid duplicating the letter “A”.

F G A Bb C D E

More details about the process of constructing the major scales are provided in Appendix B along with a Table of Major Scales. This table was used to construct a reporter block consisting of a List of Major Scales.

The information in the List of Major Scales is then used to construct a Major Scale reporter block that reports the seven notes associated with any major scale.
For example, this block reports that the following notes are associated with the *E Major* scale.

![E Major Scale]

The Major Scale & Octave reporter block reports the specified octave as well as the note in each major scale.

![F Major Scale 4 Octave]

For example, the following notes are associated with the *G Major* scale in the fourth octave. Each octave begins with the note “C”. Therefore, beginning with the note “C”, the octave shifts from the fourth octave to the fifth octave.

![G Major Scale 4 Octave]

An index, **Major Scale Index**, was constructed to indicate the point the point at which the notes in each scale shift from one octave to the next. For example, in the case of the G Major Scale, the index indicates that first three notes are in the fourth octave, while the remaining notes are in the fifth octave.

![G Major Scale Index]
The **Major Scale Index** reporter block retrieves the required information from the **List of Major Scales** reporter block.

**Minor Scales**

The procedure to report the position of the notes in the minor scale is almost identical to the similar procedure that reports the position of the notes in the major scale except that a “4”, “9”, and “11” replace “5”, “10”, and “12” in the list of positions in the chromatic scale.

<table>
<thead>
<tr>
<th>Chromatic Scale</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C#</td>
<td>D</td>
<td>D#</td>
<td>E</td>
<td>F</td>
<td>F#</td>
<td>G</td>
<td>G#</td>
<td>A</td>
<td>A#</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C Minor Scale</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>D</td>
<td>D#</td>
<td>F</td>
<td>G</td>
<td>G#</td>
<td>A#</td>
<td></td>
</tr>
</tbody>
</table>

**Intervals between Notes**

The Interval reporter block reports the interval between any two notes in a major scale. For example, there is an interval of “1” between E4 and F4. Further, since the direction of the movement is descending, the interval is reported as “-1”.

![Interval Example]
The *Interval* reporter block builds a table of the notes within that scale for the first eight octaves. It then locates the position of Note 1 and Note 2 within that list and subtracts the difference to obtain the interval between the notes.

Since the interval between notes determines whether two notes are harmonious or dissonant when played together, this information can be useful in constructing sequences of notes and chords.
Appendix C – Major Scales

The major scales consist of the following seven notes within the twelve-note Western chromatic scale: 1 3 5 6 8 10 12

*Enharmonic notes* are notes that use the same key on the piano and which have the same frequency, but which use different names depending on what scale they are part of. For example, a C# is the same frequency and piano key as a Db.

We use enharmonic notes because a major scale cannot have two notes consisting of the same letter. For example, if sharps are used in the F major scale, the letter “A” is used twice:

    F G A A♯ C D E

Therefore, “B Flat” is used in place of “A Sharp” to avoid duplicating the letter “A”.

    F G A B♭ C D E

In the table below, scales that begin with a sharp or flat are duplicated; for example, the “C Sharp” scale is also written as “D Flat”. Two scales, D# and G#, include double sharps (C## and F##). Therefore, the Eb Major and Ab Major scales are usually used in place of these two scales.

<table>
<thead>
<tr>
<th>Table of Major Scales</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
</tr>
<tr>
<td>C#</td>
</tr>
<tr>
<td>Db</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>D#</td>
</tr>
<tr>
<td>Eb</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F#</td>
</tr>
<tr>
<td>Gb</td>
</tr>
<tr>
<td>G</td>
</tr>
<tr>
<td>G#</td>
</tr>
<tr>
<td>Ab</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>A#</td>
</tr>
<tr>
<td>Bb</td>
</tr>
<tr>
<td>B</td>
</tr>
</tbody>
</table>
Appendix D
New APCS Principles

Topic 7.2 Major Scales

CS Principle AAP-1.C Strings as Variables

In computing science, a string is an ordered sequence of characters. As opposed to a list, where each item in the list is a separate value, a string acts as a single value. For example, "(A, B, C) is a list, but "ABC" is a string.

Application

In TuneScope, notes are represented using their scientific pitch notation. Each note consists of a letter name and an octave number. For notes that are sharp or flat, a modifier symbol is included between the note name and the octave number (eg. C4 or C#4). These strings may be entered manually or assembled from drop down menus.

Topic 7.6 Music Intervals within Scales

CS Principle AAP-2.N List Indexing

Indexing is used to quickly access data from a list. Each item in a list has both a value and an index number. Index numbers reference the position of their corresponding values in a list. For example, in the table below, the value of the last item in the list is “Banana”, and its index number is “3”.

<table>
<thead>
<tr>
<th>List Value</th>
<th>Apple</th>
<th>Orange</th>
<th>Banana</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Number</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Assigning an index number to each value makes it easier to not only identify the positions of items in a list, but also to manipulate and evaluate them.

Application

In TuneScope, all musical scales are zero indexed, as shown below.

<table>
<thead>
<tr>
<th>C Major Scale</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Number</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

This indexing makes it possible to determine the position of each note in the scale relative to each other note. For example, to determine how many steps it is from E to A in the C Major scale, 3 is subtracted from 6, showing that A is 3 notes above E.

Adding octave numbers to the note names makes it possible to calculate positions across multiple octaves.
The melody of a song consists of a sequential series of musical notes played one after another. A musical chord consists of several notes played together. A repeated progression of chords often is used as a backing track that repeats as the melody is played over it. This pattern, known as a chord progression, creates a sense of movement.

Some instruments, such as the piano, allow the musician playing it to play chords with one hand and melody with another. With other instruments, such as the guitar, it can be difficult to play both chords and melody at the same time. Therefore, many bands employ two guitarists, one to play chords and another to play melodies. Other instruments, like brass and woodwinds, are physically incapable of playing more than one note at a time, making playing chords on these instruments impossible. Of course, a computer can be programmed to play many chords and melodies, as you are discovering in this module.

### Topic 8.1 Parallel and Sequential Notes

The **Play Note and Wait** code block was introduced in the previous module (*Notes and Musical Scales*). This code block waits until one note ends before beginning the next.

![Play Note and Wait](image)

In the illustration above, the note G4 is played for one-sixteenth of a beat, followed by A4 for an eighth of a beat. On a piano, the first key would be struck, and then musician would then wait for one-sixteenth of a beat before striking the next piano key.

In contrast, a chord consists of several notes played together at the same time. The **Play Note** code block does not wait until one note ends before beginning the next note.

![Play Note](image)

On a piano, the musician would strike all four piano keys corresponding to the notes A3, C3, F3, and A4 at the same time.

Some combinations of notes sound more pleasing when played together. For examples, notes whose frequencies form small integer ratios such as 3:2 and 4:3 sound better than some of the other possible combinations of notes when played together. These combinations of notes that sound pleasing when played together are often combined to form chords.

### Exploration 8.1 Play Notes

Use the **Play Note and Wait** code block to create combinations of notes that sound pleasing. Identify two or three combinations of notes that could be used to form chords.
Topic 8.2 Play Chord Block

Rather than using four separate Play Note code blocks to play the four notes of the chord, a single Play Note code block can be combined with a loop to play the notes in a list.

A more general form of this script can be created through development of a custom code block, Play Chord.

The list of notes that form the chord are provided as an input to the Play Chord code block.

While each of the notes in the chord should be played together, the Play Chord block should wait until the chord is completed before continuing to the next chord. This can be accomplished by adding a Rest block at the end of the procedure.

Essential Knowledge

CS Principle 3.2 Data Abstraction

Data abstraction can be created using lists. Developing a data abstraction to implement in program can result in a program that is easier to develop and maintain.

An element is an individual value in a list that is assigned a unique index.
An index is a common method for referencing the elements in a list or string using natural numbers.

The use of lists allows multiple related items to be treated as a single value. Lists are referred to by different names, such as array, depending on the programming language.

**CS Principle 3.10 Lists**

The Length of List code block reports the number of elements in a list.

**CS Principle 3.8 Iteration**

**CS Principle 3.13 Developing Procedures**

**Application of Essential Knowledge**

In the example above, a list of notes is used to create a musical chord. A *For* loop is used to develop a custom code block named *Play Chord*. The custom code block has two inputs: *Chord* and *Note Length*.

**Exploration 8.2 Play Chord Block**

Put one of the combinations of notes developed in *Topic 9.1* together in a list. Use the chord list created in this way as an input to the custom *Play Chord* code block.

**Topic 8.3 Major Chords**

The previous module (*Musical Notes and Scales*) introduced the major and minor scales. These scales are seven-note subsets of the full twelve-note chromatic scale. Major chords are three-note combinations of notes within a seven-note major scale. For example, the C Major scale consists of the seven white keys, numbered one through seven in the illustration below.
The C Major chord consists of the first, third, and fifth notes of the C Major scale (i.e., the notes C, E, and G).

This pattern is repeated for all of the other major scales. For example, the D Major chord is composed of the first, third, and fifth notes of the D Major scale (i.e., the notes D, F#, and A).

A block that reports the major chord for any major scale simply reports the first, third, and fifth items of that scale.

For example, the D Major chord consists of D, F#, and A. In the example below, the fourth octave has been specified for this chord.
The **Play Chord** code block plays a group of notes that form a chord.

The **Major Chord** reporter block can now be used in combination with the **Play Chord** code block to play any of the twelve major chords in any octave.

**Exploration 8.3 Major Chords**

Try combining the note C from one octave with the notes E and G from different octaves. How does that affect the quality of the chord produced? Try assembling several sequences of chords. For example, try combining a C Major chord, followed by a F Major chord, followed by a G Major chord. What other sequences of chords sound good in combination with one another?

**Topic 8.4 Minor Chords**

Minor chords are formed from notes that are a subset of the notes of a minor scale. For example, in the case of the C Minor scale, the first, third, and fifth notes form a C Minor chord (that is, the notes C, Eb, and G).

The following procedure reports the notes of a minor chord - a list consisting of the first, third, and fifth notes of the minor scale.
For example, in the case of the C Minor scale, the notes C, Eb, and G form the C Minor chord.

On a piano keyboard, this chord consists of the combination of keys shown in the illustration below. The shift from a C Major chord to a C Minor chord involves replacement of the note E with the note below it in the scale. This causes the minor chord to be perceived as more dissonant because the ratios formed are higher.

**Exploration 8.4 Minor Chords**

Assemble several sequences of chords that include both major and minor chords. For example, try combining a C Major chord, followed by an A Minor chord, followed by a F Major chord, followed by a G Major. What other combinations of major and minor chords might sound good?
Topic 8.5 Four Note Chords

A total of 24 chords – 12 major chords and 12 minor chords – have been described thus far. These chords, major and minor, form the foundation for all other chords.

The foundational chords contain three notes. Four-note chords are also very common. Addition of a fourth note changes the way that the chord sounds. Some chords can contain as many as ten different notes, but these rare chords are typically only found in experimental forms of classical and jazz music.

For example, the seventh note of a minor scale is often added to a major chord to create dissonance in genres such as the blues. For example, a C Major chord consists of the notes C, E, and G. The seventh note of the C Minor scale is B♭.

The Add Note to Chord code block can be used to append the seventh note in the C Minor scale to a C Major chord.

Because this type of chord is formed by adding the seventh note of a minor scale to a major chord, it is often referred to as a Seven Chord. A Seven Chord code block can now be made to combine a major chord with the seventh note of the corresponding minor scale to form a Seven Chord.

Other methods of forming Seven chords can be found in the Appendix.

Another common way of creating four-note chords is to repeat the two notes of the chord in a different octave. For example, the notes C4 and G4 might be repeated in the fifth octave to create the chord combination C4, G4, C5, G5.
**Exploration 8.5 Four-Note Chords**

Create several four-note chords. Add a note from a minor scale to a major chord to create a four-note chord. For example, try combining G, B, D (from the G Major scale) with F (from the G Minor scale). Try creating chords that double the first and second notes of the chord in a different octave. For example, try combining G4 and E4 with G5 and E5.

**Topic 8.6 Chord Progressions in the Major Scale**

A chord progression consists of a series of chords, often used as a backing track that accompanies a melody consisting of individual notes. A common chord progression consists of the chords associated with the first, fourth, and fifth note in a major scale.

![Chord Diagram]

For example, the notes C, F, and G are the first, fourth, and fifth notes in the C Major scale. Therefore, the C Major chord progression would consist of the C Major chord, the F Major chord, and the G Major chord.

Because a Major Scale chord progression is established by the first, fourth, and fifth notes in the scale, the corresponding chords are often referenced as the I Chord, the IV Chord, and the V Chord. Roman numerals are used to reference chords to differentiate chords from notes. The I-IV-V chord progression for the C Major scale is summarized in the table below.

<table>
<thead>
<tr>
<th>Chord Numbers</th>
<th>I</th>
<th>IV</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Major Chord Progression</td>
<td>C Chord</td>
<td>F Chord</td>
<td>G Chord</td>
</tr>
</tbody>
</table>

A chord progression beginning with a D Major chord would consist of the D Major Chord, the G Major Chord, and the A Major chord. Since these chords maintain the same relative positions in the D Major scale, the chord numbers remain the same.

<table>
<thead>
<tr>
<th>Chord Numbers</th>
<th>I</th>
<th>IV</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>D Major Chord Progression</td>
<td>D Chord</td>
<td>G Chord</td>
<td>A Chord</td>
</tr>
</tbody>
</table>

There are many other chord progressions, but the I-IV-V chord progression is the basis of much of the popular music that employs the Western chromatic scale.
The **Major Octave Chord** code block can be used to identify the chords along a major scale.

Chords can be described by using either the name of the chord (C Major chord) or the number of the chords (the I chord). Both chords consist of the same notes regardless of whether the name of the chord or the number of the chord is used.

However, there is one advantage to referring to chords by number. A loop can be constructed to play a sequenced of numbered chords.

This, in turn can be used to construct a code block that can be used to play any sequence of chords.

This block can be used as the basis for construction of a procedure to play any progression of major chords.
Once a code block has been constructed that can play a progression of numbered chords, the chord progression can be readily shifted from one scale to another.

Because of this ease in shifting from one scale to another, musicians will sometimes refer to the “I Chord” or the “IV Chord”. Using this notation, the chords for the first verse of “Over the Rainbow” would be represented in the following way:

I iii IV i IV i vii ii vii i

**Exploration 8.6 Chord Progressions in the Major Scale**

Try experimenting with chord progressions composed of various major chords. For example, create a custom code block that plays a V-IV-I chord progression in any scale.

**Topic 8.7 Chord Progressions in the Minor Scale**

Minor chords are referenced with lower-case Roman numerals to distinguish them from major chords, which are referenced with upper-case Roman numerals.

A chord progression that begins in a minor scale often consists of two minor chords followed by a major chord. For example, a chord progression that begins with a C Minor chord would be followed by an F Minor chord succeeded by a G Major chord.

<table>
<thead>
<tr>
<th>Chord Numbers</th>
<th>i</th>
<th>iv</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Minor Chord Progression</td>
<td>C Chord</td>
<td>F Chord</td>
<td>G Chord</td>
</tr>
</tbody>
</table>

Construction of a block that reports the notes associated with chords in the minor scale parallels the methods used to construct a similar code block for the major scale, except that the minor scales are used in creation of this reporter block.

This block, in turn, can be used to construct a custom minor chord progression block. This block parallels the major chord progression block, except that it uses the minor scale.

A blues chord progression can be formed by adding a fourth, minor chord to the end of the chord progression. While there are songs that follow a strict I-IV-V or i-iv-v chord progression, moving directly from one chord to the next before repeating the pattern, most songs that use those chords introduce more variation to the pattern. Some songs might add more chords to the sequence (e.g., I-IV-V-IV), while other songs might change the order completely (e.g., IV-V-I-I).
**Exploration 8.7 Chord Progressions in the Minor Scale**

Try experimenting with chord progressions composed of various major and minor chords. For example, try playing a chord progression consisting of G Major, D Major, and A Minor chords. Using the Roman numeral rotation, this can be written as I-V-ii.

**Topic 8.8 Complex Chord Progressions**

The twelve-bar blues is one of the most influential chord progressions. The blues is a musical genre that can be played with just three chords. Because thousands of songs can be played with these three chords, it is a form that is accessible to musical novices.

The blues originated with Black musicians in the nineteenth century. The blues influenced many of the popular music forms that followed, including jazz, rhythm and blues, rock and roll, pop, and rock music. The most common blues format consists of a line sung over the first four bars, repetition of the line over the next four bars, followed by a concluding line over the last four bars. These measures follow a I-IV-I-V-IV-I progression. The I chord is played for four measures, the IV chord is played for two measures, the I chord is played again for two measures, the V chord is played for one measure, the IV chord is played for one measure, and then the I chord is played for the last two measures.
The song *Crossroads*, recorded by Robert Johnson in 1936, is a good example of this form.

I went to the crossroad. Fell down on my knees.
I went to the crossroad. Fell down on my knees.
Asked the Lord above, "Have mercy, now, save poor Bob if you please."

Ooh, standin' at the crossroad. Tried to flag a ride.
Ooh-ee, I tried to flag a ride.
Didn't nobody seem to know me, babe. Everybody pass me by.

Four bars of musical chords are played as each line of the song is sung. When *Crossroads* is played in the key of C, the following chords accompany each line.

<table>
<thead>
<tr>
<th>C Chord</th>
<th>C Chord</th>
<th>C Chord</th>
<th>C Chord</th>
</tr>
</thead>
<tbody>
<tr>
<td>F Chord</td>
<td>F Chord</td>
<td>C Chord</td>
<td>C Chord</td>
</tr>
<tr>
<td>G Chord</td>
<td>F Chord</td>
<td>C Chord</td>
<td>C Chord</td>
</tr>
</tbody>
</table>

This same progression can be used with minor chords as well. To create a melancholy sound, any of these chords can be played as their four-note *Seven Chord* variant.

**Exploration 8.8 Complex Chord Progressions**

Try building a chord progression like the one above. Try changing some of the major chords to seven chords. How does that change the overall sound of the chord progression?

**Topic 8.9 Parallel Chords and Melodies**

In the previous module, a backing track was created to accompany a melody played in a parallel track. The backing track consisted of the first, third, and fifth notes of the of the C Major scale, the A Minor scale, the F major scale, and the G major scale. The *Track Chords* block, marked with a ♫, can be used to play sequences of notes as chords rather than as individual notes played sequentially.

A variable with this sequence of chords can be created in the following way:

Once the parallel lists of chords and chord durations have been created, they can be supplied as inputs to the *Track Chords* block:
The **Track Chords** block can then be provided as an input to the **Play Tracks** block. (The **Play Tracks** block was first introduced in the previous module, *Notes and Scales.* ) Once the chord track has been tested, a melody can then be added as a second track.

**Exploration 8.9 Parallel Chords and Melodies**

Use the techniques described in this module and the previous module (*Musical Notes and Scales*) to create a melody and an accompanying backing track consisting of a chord progression. Use the **Play Tracks** block to play the two tracks together.

**Topic 8.10 Musical Loops**

In music it is common to create loops in which the same sequence of notes repeats over and over again. The **Track Loop ♫** block provides the capability to repeat a sequence of chords for as long as the melody track is playing.

A track that loops must be combined with at least one melody track. The length of the melody track then determines how long the looping track will play.

**Exploration 8.10 Musical Loop**

Use a looping chord progression as a backing track and combine it with a melody track.
Appendix A
TuneScope Blocks Introduced

**Play Note**

The **Play Note** block plays a note for a given duration but doesn’t wait for that note to end before passing control to the next block.

![Play Note Block](image)

**Play Chord for Duration**

The **Play Chord for Duration** block plays a chord for a given duration and waits for the chord to finish playing before passing control to the next block.

![Play Chord Block](image)

**Major Chord**

The **Major Chord** reporter block reports the notes in a major triad chord.

![Major Chord Block](image)

**Minor Chord**

The **Minor Chord** reporter block reports the notes in a major triad chord.

![Minor Chord Block](image)
Add Note to Chord

The **Add Note to Chord** block reports a chord paired with an additional note appended.

![Add Note to Chord](image1)

Major Octave Chord

The **Major Octave Chord** block reports a chord from a selected position within a major scale.

![Major Octave Chord](image2)

Minor Octave Chord

The **Minor Octave Chord** block reports a chord from a selected position within a major scale.

![Minor Octave Chord](image3)

Chord

The **Chord** block reports a chord paired with a note duration.
Chord Section

The Chord Section block reports a series of measures containing chords.

Chord Track

The Chord Track block reports a series chords and assigns an instrument to play them.

Appendix B
Seven Chords

A Seven Chord can be formed by adding the seventh note of either a major or minor scale to a major or minor chord, creating four possible ways in which this type of chord can be formed (summarized in the table below).

<table>
<thead>
<tr>
<th>Original Chord</th>
<th>7th Note of</th>
<th>Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Major Chord</td>
<td>Major Scale</td>
<td>Cmaj7</td>
<td>The first 7th chord to appear regularly in classical music. Also known as the Dominant 7th chord. Used extensively by blues musicians and barbershop quartets.</td>
</tr>
<tr>
<td>C Major Chord</td>
<td>Minor Scale</td>
<td>C7</td>
<td>This chord is rarely used in rock and pop music. It is predominantly found in jazz, classical, and flamenco. It is also the chord notoriously used in Alfred Hitchcock's film Psycho, colloquially known as the Hitchcock Chord.</td>
</tr>
<tr>
<td>C Minor Chord</td>
<td>Major Scale</td>
<td>CmM7</td>
<td></td>
</tr>
<tr>
<td>C Minor Chord</td>
<td>Minor Scale</td>
<td>Cm7</td>
<td></td>
</tr>
</tbody>
</table>
9. Building Drum Tracks

Glen Bull, Jo Watts, Joe Garofalo, and Rachel Gibson

A drum track establishes the rhythm or beat for the music. For that reason, the drum is often a key member of a band, establishing the tempo or rhythm that other musicians follow. A rotating object offers mechanical way of creating a pattern that repeats.

The San Francisco Exploratorium collaborated with the Lifelong Kindergarten group at M.I.T. to create a series mechanical sound machines that could be programmed to produce repeating patterns in this way. Drum patterns can also be created in TuneScope using rotating digital objects.

**Topic 9.1 Visualizing Drum Patterns**

The circle below can be visualized as the face of a clock. As the second hand sweeps around the circle, it plays a snare drum each time it strikes a green dot and plays a bass drum sound each time it strikes a blue dot.
In music with a 4/4 time signature (i.e., four beats per measure) the drum sounds generated by the blue dots occur in time with the on-beat notes while the drum sounds generated by the green dots occur in time with the notes that are off the beat. This is a classical drum pattern frequently found in music.

The rotating second hand of the clock face consists of a line created in the costume Paint Editor. The rotation center should be set to the origin of the line as shown in the illustration below.

The rotation of the line is accomplished by placing a Turn block inside a Forever loop.

The Point in Direction block can be used to precisely position the line in a specified direction. Another sprite, such a the blue dot shown in the illustration below, can then be placed on the tip of the line.
The code for the *Blue Dot* sprite plays a drum sound when it is touching the line. A delay at the end prevents the drum from playing multiple times as the line passes through the circle,

```blocks
when touching Line ?
Set Instrument To snare
Play C4 For Sixteenth Note Length and Wait
wait 1 secs
```

The size of the blue dot can be increased while the drum sound is playing to provide visual as well as auditory feedback.

```blocks
when touching Line ?
set size to 110 %
Set Instrument To snare
Play C4 For Sixteenth Note Length and Wait
set size to 100 %
wait 1 secs
```

A circle placed beneath the dots can help reinforce the concept of repeating circular motion. The circle does not affect sound production – rather, it is added as a visual aid. A circle can be created in the paint editor in the same way that the line was created. The Set Size block can be used to adjust the size of the circle until its radius is the same size as the line. Once the circle is position in the right location, the **Stamp** block can be used to stamp the shape on the stage beneath the dots. Once the circle has been stamped onto the stage, the sprite with the circle costume can be hidden or deleted.
The blue sprites have been placed at 0 and 180 degrees in the illustration below and the green dots have been placed at 90 and 270 degrees. This placement produces the common four-beat drum pattern found in many genres of music.

Now that the larger circle is set, the smaller, colored circles can be plotted onto it. These circles can be made in the Costumes tab of TuneScope as well. The circles can essentially be plotted anywhere on the larger one, but placing two at 0 and 180 degrees and the other two at 90 and 270 degrees will produce a common four-beat drum pattern found in many genres of music.

As the line sweeps around the circle, the specified drum sound will play as it touches each dot.

**Exploration 9.1 Visualizing Drum Patterns**

Continue to visualize patterns by adding additional circles. Try adding a larger circle that encapsulates the current one. Make the line longer so it reaches the new circle and plot more colored dots onto it, with each corresponding to a different drum sound.

**Topic 9.2 Exploring Drum Patterns with a Musical Circle**

In Western music based on the twelve-note chromatic scale with a 4/4 time signature, the first beat of each measure usually coincides with the strongest accents in the melody. Therefore, a chord change often occurs at this point. The third beat is the point with the second greatest potential for a chord change. Anything that occurs at these points is said to be “on the beat.” Events that occur on the second or fourth beat are said to be “off beat.” Some genres, such as dance hall, reggae, and ska, reverse this pattern.

The most basic element of a drum pattern is the bass drum. In many forms of music, the bass drum controls the rhythm and sets the tempo. The most common drum patterns in Western 4/4 music place the bass drum on the beat. This is like the pattern shown in the previous topic.

Within a rhythmic pattern, shifting the regular accents associated with the pattern is called “syncopation”. Syncopated music can disrupt the listeners expectations and create a pleasing
sensation when the rhythm resolves to its original pattern. This technique spread in America from gospel choirs, eventually finding its way into rock and roll, rockabilly, and funk. Syncopation is usually added by adding a snare drum on the off beats of a measure.

The colored circles can be moved around the larger circle to create a variety of drum patterns. More circles can be added to make increasingly complex patterns that explore rhythms from around the world.

In this example, there are three different colored circles, each with a corresponding instrument. This visualization is a rhythm often found in Greek Hasapiko music. All of the drums fall either on the beat or on the off-beat.

This visualization is a Cuban cinquillo rhythm. Note that while most of the drums fall on the beat or off-beat, there are two that fall between beats.

This visualization displays a rhythm that is often found in traditional Persian music. This rhythm is played in 6/8 time. In 6/8 time, there are six eighth notes per measure instead of 8. This allows the rhythm to be divided into both two and three beats. The red beats
Exploration 9.2 Exploring Drum Patterns with the Rhythm Visualizer

Implement the different rhythms shown above. After watching John Varney’s TED Ed Talk, pick out a few more rhythms from the video to implement, hear, and explore. Try to find other rhythms from around the world to try out on the musical circle.

Topic 9.3 Creating Drum Tracks with a Drum Machine

Many professional musicians and DJs use drum machines to build drum tracks. A drum machine is sometimes known as a drum sequencer. One type of interface for a drum machine resembles the illustration below.

There are two rows of 16 squares. This particular drum machine is set to sixteenth notes, so there are 16 possible beats to be played. The top of the grid indicates where the quarter notes land within the measure. Drum machines are set up in a linear way, which means that it reads the grid from left to right.

To play the kick drum on each quarter note the squares at the first, second, third, and fourth quarter note location are turned on. This is shown in the example below. In this example, the kick drum plays on the first, second, third, and fourth quarter note of a measure.

The following example illustrates a simple blues rhythm in which the kick drum is played on the first and third quarter notes, while the snare drum is played on the second and fourth quarter notes.

This is one of the most basic, yet foundational rhythms that nearly all other blues and rock rhythms are constructed from. In the topics that follow, a similar interface for a drum machine will be constructed in TuneScope. TuneScope has several types of drums, including a snare drum and a bass drum.
To simplify construction, a drum track with only eight rather than sixteen slots is created in the example that follows. However, once the foundational principles are understood, a drum machine of any type or design can be constructed. In the block shown below, each of the eight slots represents an eighth note.

![Drum Pattern Block](image)

The drum machine can be created by using a **Drum Pattern** block provided as a built-in block in TuneScope. The **Drum Pattern** block represents patterns in the form of a list in which an “X” represents a drum hit and an empty box represents a rest. A beat duration of a quarter, eighth, or sixteenth note must also be selected. A **Track Drum Loop** block, also provided in TuneScope, can be used to play parallel drum tracks.

![Track Drum Loop Block](image)

The **Track Drum Loop** block reporter block must be used in combination with the **Play Tracks** block. In the example below, the **Snare Drum** pattern repeats for as long as the chords in the piano track play.
A complete drum track can be built by adding more tracks on top of one another and using various percussion instruments, such as a snare drum, bass drum, hi-hat, etc. This capability can be expanded to play multiple instrumental tracks and multiple drum tracks in parallel.

**Exploration 9.3 Creating Parallel Drum Tracks**

Create a custom **Play Tracks** custom sequence that combines at least two drum sequences with a melody track and a chord progression track.

**Topic 9.4 Controlling the Tempo**

The beats per minute can be adjusted using the **Set Tempo** code block.

![Set Tempo Code Block]

If a variable named *Beats per Minute* is created, a slider can be created to control the tempo. Right-clicking a variable watcher displayed on the stage accesses a menu that can be used to convert the variable watcher to a slider.
Inserting the code block **Set Tempo to Beats Per Minute BPM** before the **Play Tracks** block enables the slider to control the tempo.

**Exploration 9.4 Controlling the Tempo**

Add a control to the user interface that enables the user to control the tempo with a slider.

**Topic 9.5 Exploring More Drum Machine Patterns**

The pattern of Bass Drum – Snare – Bass Drum – Snare is one of the most basic drum patterns beginners learn, and it forms the basis of many more advanced patterns, like those explored in previous topics with the musical circle. Try doubling a drum using eighth notes instead of quarter notes. Try shifting a beat slightly or leaving it out completely. How does this change the feel of the rhythm? In addition, how does the experience of using the drum machine, which reads rhythms linearly, differ from the musical circle, which reads rhythms circularly?
Once you have created a basic pattern with two drum tracks, try adding a cymbal or a hi-hat as a third track. Cymbals and hi-hats are often used to provide additional ornamentation to a drum pattern. Explore placement at different places in the pattern to see what sounds best to you.

Rhythms from different genres of music can continue to be explored using the drum machine. Below are the same rhythm examples from Topic 9.2 translated into the **Drum Pattern** block format.

This code corresponds to a common rhythm found in Greek hasapiko music.

![Drum Pattern Block for Greek Hasapiko](image)

The code below corresponds to the Cuban cinquillo rhythm.

![Drum Pattern Block for Cuban Cinquillo](image)
This code corresponds to a common rhythm found in Persian music.

**Exploration 9.5 Exploring More Drum Machine Patterns**

Implement the examples shown above. Then, try searching for more rhythms from around the world. Implement those first using the code blocks described above, then, try translating the rhythms you found to the musical circle.
Appendix A  
TuneScope Blocks Introduced

Drum Pattern

The **Drum Pattern** block reports a series of beats that tell the drum when to play. Beats can be added or subtracted from the track using the arrows on the right of the block.

Loop Drum

The **Loop Drum** block reports a drum pattern, assigns a drum to it, and sets the duration value of each item in the pattern.

Tempo

The **Tempo** block sets the speed at which music in TuneScope plays. The more beats are in a minute, the faster the tempo.
10. Lyrics and Vocal Tracks

Glen Bull, Jo Watts, and Rachel Gibson

Topic 10.1 Lyrics

The process of creating a melody, accompanying chords, and a drum track in TuneScope has been described in previous modules. Lyrics can be combined with music to create a song. For example, Judy Garland’s classic theme song in the Wizard of Oz (“Somewhere Over the Rainbow”) begins with the following lyrics.

Somewhere over the rainbow
Way up high
There’s a land that I heard of
Once in a lullaby

The following table illustrates the way in which lyrics are combined with the melody, chords, and drum track.

<table>
<thead>
<tr>
<th></th>
<th>Measure 1</th>
<th>Measure 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lyrics</td>
<td>Some where</td>
<td>O-ver the Rain-bow</td>
</tr>
<tr>
<td>Melody</td>
<td>F3 F4 E4 C4 D4 E4 F4</td>
<td></td>
</tr>
<tr>
<td>Drum</td>
<td>x x x</td>
<td>x</td>
</tr>
<tr>
<td>Chords</td>
<td>F Major (F A C)</td>
<td>A Minor (A C E)</td>
</tr>
<tr>
<td></td>
<td>Whole Note</td>
<td>Whole Note</td>
</tr>
</tbody>
</table>

Exploration 10.1

Enter the first two measures of a tune created in previous modules in a table similar to the one above. Then add a line of lyrics to accompany the tune.

Topic 10.2 Computer Poetry

Creation of original lyrics to accompany a tune can be daunting for the novice. Creation of a poem that can read with the accompaniment of music offers a useful entry point. Computer-generated provides one method for stimulating the imagination. One strategy involves random selection of parts of speech such as nouns and adjectives to complete phrases.
The **Adjective** reporter block, for example, generates a randomly selected adjective as output.

The **Say** code block causes the currently selected sprite (in this example, Abbey) to say the words provided as inputs.

This method can generate metaphor poems such as:

Halloween is scary.
Halloween is a ghost
in the moonlight.

**Exploration 10.2 Computer Poetry**

Create a metaphor poem similar to the one above. Ask friends to provide items for each category in the metaphor poem. Then run the program several times and pick the poem that you like the best.

**Topic 10.3 Rhyming Dictionaries**

A metaphor poem does not require words to rhyme. A rhyming dictionary can provide a useful way to identify words that rhyme. For example, the first line of a winter poem might be:

I love the winter snow
The **Words Rhyming with** reporter block can generate a list of words that rhyme with “snow”:

![Words Rhyming with snow](image)

This might lead to a second line that rhymes with the first line:

I love the winter snow,  
Hearing the wind blow

There are a number of online dictionaries that can be accessed to generate lists of rhyming words. An application interface is needed to access rhyming words from within Snap! or TuneScope. One of these services can be accessed through: “datamuse.com/api/”. The first part of the web address for the api consists of the following element:

https://api.datamuse.com

This element is combined with

/words?rel_rhy=

Followed by a word such as “snow”. The final construct would consist of the following:

https://api.datamuse.com/words?rel_rhy=snow

The **Join** block can join these elements together in Snap!:

The web address constructed in this manner is then placed in the Snap! **URL** block:
The output is returned in a format that looks like this:

```json
[{
  "word": "go",
  "score": 7263,
  "numSyllables": 1,
}, {
  "word": "brow",
  "score": 4758,
  "numSyllables": 1,
}, {
  "word": "how",
  "score": 4706,
  "numSyllables": 1,
}, {
  "word": "know",
  "score": 3449,
  "numSyllables": 1,
}, {
  "word": "forego",
  "score": 3335,
  "numSyllables": 2,
}, {
  "word": "so",
  "score": 3158,
  "numSyllables": 1,
}, {
  "word": "throw",
  "score": 3103,
  "numSyllables": 1,
}, {
  "word": "flow",
  "score": 3075,
  "numSyllables": 1,
}, {
  "word": "though",
  "score": 2846,
  "numSyllables": 1,
}, {
  "word": "ratio",
  "score": 2727,
  "numSyllables": 3,
}, {
  "word": "pro",
  "score": 2653,
  "numSyllables": 1,
}, {
  "word": "hello",
  "score": 2628,
  "numSyllables": 2,
}, {
  "word": "quo",
  "score": 2527,
  "numSyllables": 1,
}, {
  "word": "grow",
  "score": 2500,
  "numSyllables": 1,
}, {
  "word": "Paw",
  "score": 2467,
  "numSyllables": 1,
}, {
  "word": "sew",
  "score": 2379,
  "numSyllables": 1,
}, {
  "word": "apropos",
  "score": 2279,
  "numSyllables": 3,
}, {
  "word": "mow",
  "score": 2123,
  "numSyllables": 1,
}, {
  "word": "bio",
  "score": 2118,
  "numSyllables": 2,
}, {
  "word": "low",
  "score": 2100,
  "numSyllables": 1,
}, {
  "word": "forgo",
  "score": 2026,
  "numSyllables": 2,
},]
```

This output includes the number of syllables in each rhyming word and a score used to rank the rhyming words. The data that accompanies the rhyming words is in a format known as JavaScript Object Notation (JSON). The **Split** block can be used to split JSON data into lists of data.
The **Map** block can be used to apply an operation (i.e., map the operation) to all of the items in a list simultaneously. This method can be more efficient that using a **For** loop to apply an operation to each item in a list one at a time. In this example, the **Map** block has been used to keep Item 1 of the columns that contain the item “word”:

Once this result has been obtained, Item 2 of each remaining list (i.e., the rhyming word) can be selected:

If only the ten most relevant results are wanted, the **Item** block can be combined with the **Numbers 1 to 10** block to select the first ten items in the list:
By this means, the Data Muse api can be to create a **Words Rhyming with ___** code block.

The **Item Random** block can then be combined with the **Words Rhyming with _** block to generate lines of poetry in the same manner as the Metaphor Poems described in the preceding section.

The rhyming dictionary can be used to generate poems that rhyme. For example, the following is an example of an A-B A-B rhyming pattern.

Winter is snow.
You can hear it crunch.
Winter is _____.
You can feel it _____.

This rhyme could be implemented in the following way:

These results can be edited to create a poem that is the most satisfactory by the user.

Winter is snow.
You can hear it crunch.
Winter is whoa!
You can feel it scrunch.

In this manner, the computer can be used to stimulate creation of poems.
Exploration 10.3 Rhyming Dictionaries

Use the rhyming dictionary to generate a rhyming poem. After running the procedure several times, pick the version of the poem that you like the best.

Topic 10.4 Other Options for Generation of Words

The Data Muse API can be used to access several other options such as words that follow another word or words that are associated with another word. The table below lists options that can be accessed by appending them to the URL for the Data Muse API: “api.datamuse.com”

<table>
<thead>
<tr>
<th>Examples of Data Muse Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option</td>
</tr>
<tr>
<td>Words that rhyme with</td>
</tr>
<tr>
<td>Words with a meaning similar to</td>
</tr>
<tr>
<td>Words that follow</td>
</tr>
<tr>
<td>Words associated with</td>
</tr>
</tbody>
</table>

There are many other options that can be accessed in a similar way to generate poetry.

Exploration 10.4 Other Options for Generation of Words

Incorporate some of the other options such as “Words that Follow” or “Words Associated With” to generate poetry.

Topic 10.5 Creating Lyrics for an Existing Melody

In the United States, the copyright for a musical composer expires 70 years after the death of the composer. Pachelbel died in 1706, so this musical composition has been in the public domain since 1776.

This means that anyone has been free to re-purpose this work and incorporate it into their own tunes and compositions. Consequently, elements of Pachelbel’s Canon have been incorporated into the following songs and many others.

Hook, Blues Traveler
Go West, Pet Shop Boys
Cryin’, Aerosmith
Tunnel of Love, Dire Straits
No Woman No Cry, Bob Marley and the Wailers
Streets of London, Ralph McTell
Memories, Maroon Five
Let It Be, The Beatles

The form also lends itself to parody. The following are lyrics created at the beginning of allergy season to accompany Pachelbel’s Canon:

Pachelbel’s Allergy

Flowers blooming
Pollen looming

157
Spring in C’ville
Sneezin’ season
Buy your Kleenex
In profusion
Claritin is part of mealtime

The following lyrics in a more serious vein were created by Mary Peng, a University of Virginia student.

*Faith, Hope, Life*

- You are my faith, my hope, my life
- You gave me the chance to breathe
- The fresh air of the world
- And walk around
- With a heart of wonder
- Eyes for splendor
- Time goes by so fast
- My love for you
- Grows deeper
- Every passing day

**Exploration 10.5 Lyrics**

Create lyrics to accompany Pachelbel’s Canon or a similar class work.

**Topic 10.6 Recording Vocals**

Once lyrics have been created, the next step is to record a vocal track to accompany the other music tracks. A sound recording program such as Audacity (https://www.audacityteam.org/) can be used to record vocals. The vocals can then be exported as an “.mp3” or “.wav” file and imported into TuneScope.

Although the audio editing capabilities of TuneScope are not as advanced as the editing features of a dedicated audio recording program such as Audacity, it is possible to edit audio waveforms directly in TuneScope. The process of recording a sound was described in a previous module (Module 5).

This process converts the back-and-forth movements of a computer’s microphone into an electrical signal. The amplitude of the varying electrical signal is repeatedly measured and converted into a series of digital numbers. This process is known as analog-to-digital (A/D) conversion. The process indirectly measures minute movements of the microphone and converts that movement into a series of digital numbers.
Exploration 10.6

Use the record feature to record and play back a short utterance.

Topic 10.7 Graphing Sound Waveforms

Even a short segment of digitized sound may consist of thousands of sound samples. If the digitization rate is 48,000 samples per second (for example), a single second of speech will result in a list of 48,000 numbers. These numbers represent the voltage levels generated by the back-and-forth movement of the diaphragm of the microphone used to capture the digitized sound.

This list of numbers representing a second of digitized speech, however, is more comprehensible if it is presented as a graphical representation rather than in number form. The coordinate system for the default Snap! stage spans the range from -240 to +240 units on the horizontal (X) axis, and from -180 to +180 units on the vertical (Y) axis.
These coordinates provide the framework for graphing any type of data in *Snap!*, including a segment of sound.

Prior to graphing the waveform, the turtle is positioned on the left side of the screen. The screen is cleared, and the turtle’s pen is lowered.

Rather than entering an explicit number such as “-240”, the **Left of Stage** block (accessed through the **Costume #** block in the *Sensing* palette) can also be used.

Depending on the computer, the digitized sound samples may consist of numbers such as 0.27 and 0.42. To be clearly seen, the positive peaks of the waveform should ideally fall between 20 and 100 on the vertical axis of the stage. The sound samples can be multiplied by an appropriate number to increase the scale if necessary. In the example below, the samples of the sound utterance “One Two Three” have been multiplied by 100 and assigned to a variable named **Sound**.

Once these setup and scaling activities are completed, a loop can be used to plot the sound samples. The utterance is slightly more than 2 seconds long. At a sample rate of 48,000 samples per second, more than 96,000 samples were collected.

Dividing 48 into 48,000 yields the result of 1000. Consequently, if every 48th point is plotted, each point will represent a one millisecond time increment (since there are 1000 milliseconds in a second).

The default width of the stage is 480 steps. If each step in the graph represents a one millisecond time increment, it is still only possible to graph approximately a half-second of the utterance (0.48 seconds).

This time scale can be adjusted by calibrating the number of sound samples that are graphed. If every 96th sound sample is graphed, each point will represent two milliseconds, and it becomes possible to graph nearly a full second of sound on the stage. If every 192nd sound sample is graphed, each point will represent four milliseconds, and it becomes possible to graph nearly two seconds of sound on the stage.
Since the recording of the words “one, two, three” is approximately two seconds long, graphing every 192nd point results in a time scale that makes it possible to display the entire utterance on the stage. This can be accomplished with a For loop that graphs every 192nd sound sample in the utterance. The default increment for a For loop is 1; in this instance, the increment has been increased to every 192nd point. After each point is plotted, the turtle moves over one step. By this means, the amplitude of each sound sample is graphed.

The graph shown in the illustration below results. The words “one”, “two”, and “three” can be clearly seen. The up and down variations in amplitude correspond to minute variations in the back-and-forth movements of the diaphragm of the microphone that captured the sound.
The graph shown in the illustration below results. The words “one”, “two”, and “three” can be clearly seen. The up and down variations in amplitude correspond to minute variations in the back-and-forth movements of the diaphragm of the microphone that captured the sound.

The Plot Sound Samples code block accepts the samples of a recorded sound as an input and plots the results.

Displaying the sound samples graphically makes it possible to see each of the recorded words in the graphed waveform.

**Exploration 10.7**
Graph the utterance recorded in the previous section

**Topic 10.8 Editing Sound Segments**
Previously the Numbers block was used to select the ten most relevant rhyming words.

The Numbers reporter block can also be used to select a specific range of sound samples.
The starting point and end point can be expressed in seconds if these numbers are multiplied by the sample rate of the sound.

This sound sample selection procedure can be encapsulated as a **Sound Segment** block.

The **Sound Segment** block can, for example, be used to select the sound samples that fall between a half-second and one and one-half seconds.

The sound samples selected in this manner can be graphed.
They also can be used to play specific segments of the recorded sound.

In this manner, specific segments of a sound recorded in TuneScope can be identified and played. Sound segments, in turn, can be combined to form a vocal track to accompany other music tracks created in TuneScope.

**Exploration 10.8 Editing Sound Segments**

Create and record lyrics to accompany one verse of a tune. Then use the sound editing capabilities described above to trim the beginning of the recording so that it starts at the correct time when played synchronously with the accompanying music.
Appendix A

TuneScope Blocks Introduced

Say

The **Say** block causes the sprite on the stage to say the input text for the set duration.

![Say Block Example](image)

Join

The **Join** block appends multiple strings into a single string.

![Join Block Example](image)

Split

The **Split** block splits a string according to the option selected from the drop down menu.

![Split Block Example](image)

URL

The **URL** block reports web page text.

![URL Block Example](image)

Map

The **Map** block applies an operator to each item of a list.

![Map Block Example](image)

Number from _ to _

The **Numbers from _ to _** block reports a list of numbers beginning with the first number and ending with the last one.

![Number from _ to _ Block Example](image)
**List Contains**

The **List Contains** block checks to see whether or not a list contains a given value. If the value exists in the list, the block reports true. If the value doesn’t exist in the list, the block reports false.

**Keep Items from List**

The **Keep Items from List** block reports only the selected items from a list.

**Duration of Sound**

The **Duration of Sound** block reports different characteristics of a sound. Different characteristics can be selected from a drop down menu.

**Length of List**

The **Length of List** block reports a different characteristics of a list. Different characteristics can be selected from a drop down menu.

**Plot Sound**

The **Plot Sound** block draws the waveform of a sound on the stage.
Appendix B
New APCS Principles

Topic 10.2 Computer Poetry

CS Principle AAP-2.D Evaluate Expressions that Manipulate Strings

String concatenation joins together two or more strings end-to-end to make a new string. A substring is part of an existing string.

Application

In computer science terms, words and phrases can be viewed as strings. The Join block concatenates these substrings into longer strings to form sentences.
Appendix A

TuneScope User Guide
TuneScope User Guide

TuneScope combines tools for music composition with an educational computing language, Snap!, developed at the University of California, Berkeley. TuneScope was developed through a collaboration among the School of Education and Human Services, the Department of Music, and the Department of Computer Science at the University of Virginia. We would like to express our appreciation for the assistance and guidance provided by the creators of Snap! as we have developed TuneScope.

Note: In order to use TuneScope, Javascript extensions must be turned on under the Settings menu.

I. Musical Notes

Set Instrument

The Set Instrument code block is used to select a musical instrument.

Play Note

The Play Note code block is used to play a musical note. Notes consist of a note name (e.g. C, C#, etc.) combined with an octave number (e.g. 3, 4, etc.). The term scientific pitch notation is used to describe this method of defining musical notes. Note: Octaves in the Western musical tradition begin with the note C.

Several Play Note blocks can be combined to create a musical chord.
Rest

The **Rest** code block can be used to insert a delay in a sequence of notes.

Play Note and Wait

The **Play Note and Wait** code block waits until one note is completed before beginning the next note. Several **Play Note and Wait** code blocks can be combined to play a series of notes.

Set Volume

The **Set Volume** code block is used to set the overall sound level.

Set Volume of Instrument

The **Set Volume of Instrument** code block is used to adjust the sound level of a specific instrument.
II. Musical Tracks

Musical tracks can be used to combine sequences of notes to create a song. The **Note** reporter block is the basic unit. Notes can be combined into measures. Measures, in turn, are combined to form musical tracks.

🎵 (Note)

The 🎵 (Note) reporter block consists of a note and a note duration.

![Note Example](image)

Measure

The **Measure** block is used to group sequences of notes into measures. *Note: A musical measure consists of the notes within one bar of music.*

![Measure Example](image)

Validate Measure

The **Validate Measure** block can be used to determine if the combined durations of notes within a measure are correct. (The combined duration of notes within a measure must equal the number of beats specified by the time signature. For example, in the most common time signature, 4/4 time, there are four quarter notes within a measure. Since four quarter notes equal one whole note, the combined durations of all of the notes within a measure must equal a whole note.)

![Validate Measure Example](image)

If the durations are not correct, the **Validate Measure** will report the discrepancy.

![Validate Measure Example](image)

Track

Completed measures are assembled into tracks.

![Track Example](image)
Play Tracks

The Play Tracks block is used to play music tracks. In most cases several tracks with different musical instruments are played in parallel. For example, a piano track and a guitar track are played in parallel in the example below.

Track Loop ♪

The Track Loop ♪ code block assigns an instrument to play a set of notes. These notes will repeat for the total duration of the song. If only the Track Loop ♪ code block is being used in a song, it is necessary to add a Track ♪ code block filled with rests to dictate the length of the song.

Section

When a group of measures will be repeated several times (for example, in the chorus of a song), they can be grouped into sections using the Section block.
III. Musical Scales

**Chromatic Scale**

The **Chromatic Scale** block reports the list of notes in the chromatic scale.

![Chromatic Scale](image)

**Major Scale**

The **Major Scale** block reports the notes in the specified major scale.

![Major Scale](image)
**Minor Scale**

The **Minor Scale** block reports the notes in the specified minor scale.

![Minor Scale](image)

**Major Scale Interval**

The **Major Scale Interval** block reports the distance (in steps) between one note in a major scale and a second note.

![Interval](image)

**Minor Scale Interval**

The **Minor Scale Interval** block reports the distance (in steps) between one note in a minor scale and a second note.

![Interval](image)

**IV. Musical Chords**

**Major Chord**

The **Major Chord** block reports the notes in a major triad chord. Both the note and the octave may be selected from a drop-down menu or directly entered.

![Major Chord](image)

**Minor Chord**

The **Minor Chord** block reports the notes in a minor triad chord. Both the note and the octave may be selected from a drop-down menu or directly entered.

![Minor Chord](image)
**Major Chord & Octave**

The **Major Chord & Octave** block reports the notes of a chord in the selected position within the major scale. Roman numerals are used to differentiate chord positions from octave numbers. The roman numeral “I” reports the notes of the first chord in the sequence. Upper case roman numerals represent major chords within the scale. Lower case roman numerals represent minor chords within the scale.

![Diagram of Major Chord & Octave]

**Minor Chord & Octave**

The **Minor Octave Chord** code block reports the notes of a chord in the selected position within the minor scale. Roman numerals are used to differentiate chord positions from octave numbers. The roman numeral “I” reports the notes of the first chord in the sequence. Upper case roman numerals represent major chords within the scale. Lower case roman numerals represent minor chords within the scale.

![Diagram of Minor Chord & Octave]

**Add Note to Chord**

The **Add Note to Chord** code block reports a chord with an additional note appended.

![Diagram of Add Note to Chord]

**♫ Chord**

The **♫ Chord** code block reports a chord paired with a note duration.

![Diagram of ♫ Chord]
V. Drum Tracks

Drum Pattern

The Drum Pattern block reports a series of beats that tell the drum when to play. Beats can be added or subtracted from the track using the arrows on the right of the block.

Loop Drum

The Drum Loop block reports a drum pattern, assigns a drum to it, and sets the duration value of each item in the pattern. Different drums can be selected from the drop down menu. Beat durations can be selected from the drop down menu or entered manually.

VI. Acoustic Tools

Scope

The Scope code block toggles the TuneScope oscilloscope on and off.

When the oscilloscope is displayed, sounds captured by the computer’s microphone can be displayed. In addition, tones generated using TuneScope Tone commands can also be displayed.
Tone

The **Tone** block generates a tone with a specified frequency and amplitude. Each tone is also assigned a number.

<table>
<thead>
<tr>
<th>Tone Number</th>
<th>Frequency</th>
<th>Amplitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>.5</td>
</tr>
</tbody>
</table>

**Tone Number**

Once a tone has been created, the **Tone Number** block can be used to turn that tone on or off.

| Tone Number 1 | On/Off |

**Tone Off**

The **Tone Off** code block turns all tones.

| Tone Off |

**Sound Segment**

The **Sound Segment** reporter block reports a specified segment of a larger sound sample.

| Sound Segment of samples of sound One Two Three Sound from 5 Start (seconds) to 1.5 End (Seconds) |

This block can be used to play a selected segment of a larger sound sample.

**Plot Sound**

The **Plot Sound** block plots a graphical representation of a sound wave. The time scale displayed can be selected (one-tenth second, one second, etc.) from a drop-down window.
VII. MIDI Capture Tools

**MIDI** is an acronym for *Musical Instrument Digital Interface*. The MIDI capture tools can be used to capture notes from a MIDI keyboard.

**Record MIDI**

The **Record MIDI** block is used to capture notes from a MIDI keyboard. The model number of the MIDI keyboard must be specified in the *Controller Name* input. This is a new, experimental TuneScope feature. Currently the tempo must be set to 60 beats per minute or less; notes may be lost if the tempo is faster.

When the **Record MIDI** instrument block is clicked, a click track is played to provide the musician with the tempo. Notes played on the MIDI keyboard are captured by TuneScope.

**Stop MIDI**

When the **Stop MIDI** block is clicked, the list of notes recorded are saved to a variable name specified in the “Variable Name” input. The durations of notes are also recorded, but currently this feature is not reliable.

**Play MIDI**

When the **Play MIDI** block is clicked, the notes played on the MIDI keyboard are played through the computer speakers but are not recorded. This feature can be useful when practicing prior to recording a note sequence.
Appendix B

Alignment with Advanced Placement
Computer Science Principles
## Alignment with APCS Principles

<table>
<thead>
<tr>
<th>Module</th>
<th>Topic</th>
<th>APCS Principles Introduced</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1</td>
<td>AAP-2.A Developing Algorithms</td>
<td>Express an algorithm that uses sequencing without using a programming language.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-2.J Iterations</td>
<td>Express an algorithm that uses iteration without using a programming language</td>
</tr>
<tr>
<td>1.2</td>
<td></td>
<td>AAP-3.A Procedures</td>
<td>Write statements to call procedures.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-3.B Procedural Abstraction</td>
<td>Explain how the use of procedural abstraction manages complexity in a program.</td>
</tr>
<tr>
<td>1.3</td>
<td></td>
<td>AAP-3.A Parameters for Procedures</td>
<td>Add parameters to a procedure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-3.C Developing Procedures</td>
<td>Develop procedural abstractions to manage complexity in a program by writing procedures.</td>
</tr>
<tr>
<td>1.4</td>
<td></td>
<td>AAP-3.D Selecting Appropriate Procedures</td>
<td>Select appropriate libraries or existing code segments to use in creating new programs.</td>
</tr>
<tr>
<td>1.5</td>
<td></td>
<td>AAP-1.A Variables</td>
<td>Represent a value with a variable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-2.C Mathematical Expressions</td>
<td>Evaluate expressions that use arithmetic operators.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-2.J Iteration</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-2.K Iterative Statements</td>
<td>Write iteration statements.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CRD-2.B Programs</td>
<td>Explain how a program or code segment functions.</td>
</tr>
<tr>
<td>2</td>
<td>2.1</td>
<td>AAP-2.C Mathematical Expressions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.2</td>
<td>CRD-2.D Outputs</td>
<td>Identify output(s) produced by a program.</td>
</tr>
<tr>
<td></td>
<td>2.3</td>
<td>CRD-2.C Inputs</td>
<td>Identify input(s) to a program.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-1.A Variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.4</td>
<td>CRD-2.C Inputs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-1.A Variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.5</td>
<td>CRD-2.C Inputs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-1.A Variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.6</td>
<td>AAP-2.B Programming Algorithms</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-3.A Procedures</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-3.C Developing Procedures</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-1.A Variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CRD-2.C Inputs</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3.1</td>
<td>No New Principles Introduced</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Topic</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td><strong>AAP-3.E Random Values</strong></td>
<td>For generating random values, write expressions to generate possible values.</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td><strong>AAP-3.C Developing Procedures</strong>&lt;br&gt;<strong>AAP-3.D Selecting Appropriate Procedures</strong>&lt;br&gt;<strong>AAP-3.E Random Values</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td><strong>AAP-3.F Simulations</strong>&lt;br&gt;<strong>AAP-3.A Procedures</strong>&lt;br&gt;<strong>AAP-2.J Iteration</strong></td>
<td>Explain how computers can be used to represent real-world phenomena or outcomes.</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td><strong>AAP-3.F Simulations</strong>&lt;br&gt;<strong>AAP-3.A Procedures</strong>&lt;br&gt;<strong>AAP-2.J Iteration</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td><strong>AAP-1.A Variables</strong></td>
<td>Lists as Variables.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>AAP-1.D Data Abstraction</strong></td>
<td>Develop data abstraction using lists to store multiple elements. Explain how the use of data abstraction manages complexity in program code.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>DAT-1.A Data Representation</strong></td>
<td>Explain how data can be represented using bits.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.1</td>
<td><strong>No New Principles Introduced</strong></td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td><strong>DAT-2.A Data Processing</strong></td>
<td>Describe what information can be extracted from data.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>DAT-2.D Data Extraction</strong></td>
<td>Extract information from data using a program.</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td><strong>CRD-2.A The Purpose of Innovation</strong></td>
<td>Describe the purpose of a computing innovation.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>AAP-4.A Algorithmic Efficiency</strong></td>
<td>For determining the efficiency of an algorithm, explain the difference between algorithms that run in reasonable time and those that do not.</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td><strong>CRD-2.E Program Testing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td><strong>CRD-2.J Identify Expected Outputs</strong>&lt;br&gt;<strong>AAP-1.A Variables</strong>&lt;br&gt;<strong>DAT-1.A Data Representation</strong></td>
<td>Identify inputs and corresponding expected outputs or behaviors that</td>
<td></td>
</tr>
<tr>
<td>Chapter</td>
<td>Section</td>
<td>Topics</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>5.3</td>
<td>CRD-2.F</td>
<td>User Interface</td>
<td>can be used to check the correctness of an algorithm or program.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-3.A Procedures</td>
<td>Design a program and its user interface.</td>
</tr>
<tr>
<td>5.4</td>
<td>AAP-1.A Variables</td>
<td>AAP-2.K Iterative Statements</td>
<td>AAP-3.A Procedures</td>
</tr>
<tr>
<td>5.5</td>
<td>AAP-1.A Variables</td>
<td>AAP-3.A Procedures</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>CSN-2.A Parallel Computing</td>
<td>AAP-3.A Procedures</td>
<td></td>
</tr>
<tr>
<td>5.7</td>
<td>AAP-2.H Conditional Statements</td>
<td>AAP-2.E Relational Operators</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AAP-3.E Random Values</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6.1</td>
<td>No New Principles Introduced</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.2</td>
<td>AAP-3.A Procedures</td>
<td>Calling Procedures.</td>
</tr>
<tr>
<td></td>
<td>6.3</td>
<td>AAP-3.D Selecting Appropriate Procedures</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.4</td>
<td>AAP-2.O Iterative Statements that Traverse a List</td>
<td>AAP-1.A Variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-3.C Developing Procedures</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-2.O Iterative Statements that Traverse a List</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.5</td>
<td>CSN-2.A Parallel Computing</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-2.E Relational Operators</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7.1</td>
<td>AAP-1.A Variables</td>
</tr>
<tr>
<td></td>
<td>7.2</td>
<td>AAP-1.C Strings as Variables</td>
<td>Represent a list or string using a variable.</td>
</tr>
<tr>
<td></td>
<td>7.3</td>
<td>AAP-1.C Strings as Variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-2.D Manipulating Strings</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.4</td>
<td>AAP-3.E Random Values</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AAP-2.O Iterative Statements that Traverse a List</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.5</td>
<td>AAP-2.B Programming Algorithms</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-3.A Procedures</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-3.C Developing Procedures</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-1.A Variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-2.C Mathematical Expressions</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CRD-2.C Inputs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CRD-2.D Output</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-2.O Iterative Statements that Traverse a List</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AAP-2.H Conditional Statements</td>
<td></td>
</tr>
<tr>
<td>Chapter</td>
<td>Section</td>
<td>Topic</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>7.6</td>
<td>AAP-2.N</td>
<td>List Indexing</td>
<td>Write and evaluate expressions that use list indexing and list procedures.</td>
</tr>
<tr>
<td>7.7</td>
<td>AAP-2.N</td>
<td>List Indexing</td>
<td>CRD-2.C Inputs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CRD-2.D Output</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AAP-1.A Variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AAP-3.C Developing Procedures</td>
</tr>
<tr>
<td>7.8</td>
<td>AAP-1.C</td>
<td>Strings as Variables</td>
<td>CRD-2.C Inputs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CRD-2.D Output</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AAP-3.C Developing Procedures</td>
</tr>
<tr>
<td>7.9</td>
<td>AAP-1.A</td>
<td>Variables</td>
<td>CRD-2.C Inputs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CRD-2.D Output</td>
</tr>
<tr>
<td>7.10</td>
<td>AAP-3.A</td>
<td>Procedures</td>
<td>CRD-2.C Inputs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CRD-2.D Output</td>
</tr>
<tr>
<td></td>
<td>AAP-3.C</td>
<td>Developing Procedures</td>
<td></td>
</tr>
<tr>
<td>7.11</td>
<td></td>
<td></td>
<td>AAP-3.C Developing Procedures</td>
</tr>
<tr>
<td>7.12</td>
<td>AAP-1.A</td>
<td>Variables</td>
<td>CRD-2.C Inputs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CRD-2.D Output</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CSN-2.A Parallel Computing</td>
</tr>
<tr>
<td>8.2</td>
<td>AAP-1.A</td>
<td>Variables</td>
<td>AAP-2.N List Indexing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AAP-3.C Developing Procedures</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AAP-2.O Iterative Statements that Traverse a List</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AAP-2.B Programming Algorithms</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CRD-2.C Inputs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CRD-2.D Output</td>
</tr>
<tr>
<td>8.3</td>
<td>AAP-1.A</td>
<td>Variables</td>
<td>CRD-2.C Inputs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CRD-2.D Output AAP-2.D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AAP-2.N List Indexing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AAP-3.C Developing Procedures</td>
</tr>
<tr>
<td>8.4</td>
<td>AAP-1.A</td>
<td>Variables</td>
<td>CRD-2.C Inputs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CRD-2.D Output AAP-2.D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AAP-2.N List Indexing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AAP-3.C Developing Procedures</td>
</tr>
<tr>
<td>8.5</td>
<td>AAP-1.A</td>
<td>Variables</td>
<td>AAP-2.N List Indexing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CRD-2.C Inputs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CRD-2.D Output</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AAP-3.C Developing Procedures</td>
</tr>
</tbody>
</table>
| 8.6 | AAP-1.A Variables  
     | AAP-2.N List Indexing  
     | CRD-2.C Inputs  
     | CRD-2.D Output  
     | AAP-2.O Iterative Statements that Traverse a List  
     | AAP-3.A Procedures |
| 8.7 | AAP-1.A Variables  
     | AAP-2.N List Indexing  
     | AAP-2.D Manipulating Strings  
     | CRD-2.C Inputs  
     | CRD-2.D Output  
     | AAP-2.O Iterative Statements that Traverse a List  
     | AAP-3.A Procedures |
| 8.8 | AAP-1.A Variables  
     | AAP-1.D Data Abstraction  
     | AAP-2.N List Indexing  
     | CRD-2.C Inputs  
     | CRD-2.D Output  
     | AAP-2.O Iterative Statements that Traverse a List  
     | AAP-3.A Procedures |
| 8.9 | AAP-1.A Variables  
     | AAP-1.D Data Abstraction  
     | AAP-2.N List Indexing  
     | AAP-2.D Manipulating Strings  
     | CRD-2.C Inputs  
     | CRD-2.D Output  
     | AAP-2.O Iterative Statements that Traverse a List  
     | AAP-3.A Procedures  
     | CSN-2.A Parallel Computing |
| 8.10 | AAP-1.A Variables  
     | AAP-2.K Iterative Statements  
     | CRD-2.C Inputs |
| 9   | 9.1 | AAP-1.A Variables  
     | AAP-3.A Procedures  
     | AAP-3.C Developing Procedures  
     | CRD-2.C Inputs |
|     | 9.2 | AAP-2.A Developing Algorithms  
     | AAP-2.J Iterations |
|     | 9.3 | AAP-1.A Variables  
     | AAP-2.D Manipulating Strings  
     | AAP-2.N List Indexing  
     | CRD-2.C Inputs  
     | CRD-2.D Output |
| 10 | 10.1 | No New Principles Introduced |
| 10.2 | AAP-2.D Manipulating Strings |
| 10.3 | AAP-2.D Manipulating Strings |
| 10.4 | AAP-2.D Manipulating Strings |
| 10.5 | No New Principles Introduced |
| 10.6 | No New Principles Introduced |
| 10.7 | DAT-1.A Data Representation |
| 10.8 | AAP-2.B Programming Algorithms |

| 9.4 AAP-1.A Variables |
| 9.5 AAP-1.A Variables |
| CRD-2.C Inputs |
| CRD-2.D Output |

| 10.1 No New Principles Introduced |
| 10.2 AAP-2.D Manipulating Strings |
| CRD-2.C Inputs |
| CRD-2.D Output |

| 10.3 AAP-2.D Manipulating Strings |
| CRD-2.C Inputs |
| CRD-2.D Output |

| 10.4 AAP-2.D Manipulating Strings |
| CRD-2.C Inputs |
| CRD-2.D Output |

| 10.5 No New Principles Introduced |
| 10.6 No New Principles Introduced |
| DAT-2.A Data Processing |
| AAP-2.K Iterative Statements |
| AAP-2.O Iterative Statements that Traverse a List |

| 10.7 DAT-1.A Data Representation |
| 10.8 AAP-2.B Programming Algorithms |
| AAP-2.C Mathematical Expressions |
| AAP-3.A Parameters for Procedures |
| AAP-3.C Developing Procedures |
| AAP-3.F Simulations |
| CRD-2.B Programs |
| CRD-2.C Inputs |
| CRD-2.D Output |