CodeMonkey

Write code. Catch bananas. Save the world.

# BANANA TALES

## PART ONE

Lesson Plans
1-12

# Table of Contents

# Getting Started

Thank you for choosing Banana Tales to teach your students how to code in Python. With gamified challenges, a real programming interface and an engaging user experience, Banana Tales is a great way to teach your students Python coding. The following document includes the lesson plans for Part One of Banana Tales. These lesson plans consist of two complementary documents. This file includes the lesson plans that are intended for use by the teacher in organizing their lessons and supporting students as they work through the challenges and learn the syntax of Python. The Classroom Slides (located under Banana Tale's course resources) are a separate slide deck that contains visuals and code samples referenced in the lesson plans. These Slides are formatted to present to students on a TV or projector screen.

Although previous coding experience will help in teaching this course, these lesson plans, in conjunction with the correlating Classroom Slides, will help teachers without previous coding experience to successfully teach Banana Tales. Each lesson is made up of 3 parts, an introduction, playtime, and debriefing, and is designed to be 45 minutes long. Each section is further divided into the amount of time it takes to complete.

For information regarding setting up a class, please read A Beginner's Guide to CodeMonkey. The guide can be found here or in the Teacher's Resources Menu on your homepage. Please feel free to email us at info@codemonkey.com for any questions you may have along the way.

Have fun!

The CodeMonkey Team

# Lesson 1 - Introduction

## CSTA Standards

| Elementary (3 - 5) | Middle (6 - 8) | High (9 - 12) |
|---|---|---|
| 1B-AP-10<br>1B-AP-15 | 2-AP-12<br>2-AP-17 | 3A-AP-15<br>3B-AP-21 |

## Objectives

- Define object and property
- Add objects to the play area when necessary
- Modify properties of game objects
- Complete challenges 1 - 10

## Components

**PYTHON**

- Accessing and modifying object properties

**PLATFORM**

- Feeding the banana to the monkey
- UI elements, including the Run and Rewind buttons and the drag and drop area
- Giraffe and Snake objects

## Introduction - 15 min

**DISCUSSION – 3 MIN**

Ask students about their prior experiences with coding. If no students have experience coding give them an opportunity to share other types of projects that they have created with computers: pictures, presentations, web pages, etc. For students who have done some coding, ask them what language or languages they did their coding in. Explain that in this course we will be using a language called Python.

**ACTIVITY – 10 MIN**

For this activity you will need to have room for a student to walk in a straight line for 15 to 20 feet (5 to 6 meters). You will also need a couple of easily movable

obstacles that can be used to block this path. In many classrooms it may be possible to use student desks as long as they are not too heavy or unwieldy. Otherwise you can use cardboard boxes or something similar. Finally, you will need some sort of little knick-knack or token for one student to deliver to another. A banana (real or fake) is ideal, but it can really be anything.

For the first version of the activity, you will need three students to participate. One will be the Deliverer, one will be the Receiver, and one will be the Programmer. The Deliverer and Receiver should stand at opposite ends of the cleared path. When the Programmer tells the Deliverer to go, the Deliverer should walk in a straight line towards the Receiver. When the Deliverer reaches the Receiver, he or she should give them the banana (or whatever you used).

Now change things up. Let the students position two or three of the obstacles in the path. Position the Deliverer and Receiver as before, and have the Programmer say go once more. When the Deliverer encounters an obstacle, they should simply stop rather than trying to steam-roller over it and possibly cause an accident.

Repeat the exercise one last time. Assign a student as a Mover for each obstacle. This time the Programmer gets to give the Movers instructions to clear the path before they tell the Deliverer to start. For each instruction, they should address the Mover by name and tell them what to do. For example, “Chris, please move your box back three feet (one meter).” This exercise is not intended as any great puzzle for the Programmer to solve, so with any reasonable set of instructions he or she should be able to clear the path and allow the Deliverer to reach the Receiver once more.

### EXPLANATION – 2 MIN

Explain to students that the activity they just completed is a model for how the challenges in Banana Tales work. In every challenge the goal is to get the banana to the baby monkey, but the little car carrying the banana needs a clear path to do so. As programmers, the students will write code not to control the car, but to control the environment to make sure that the path between the car and the baby monkey is clear and safe.

## Playtime - 25 min

### LOG-IN INFORMATION - 2 MIN

Go to <www.codemonkey.com>.

Instruct your class on how to log in to their CodeMonkey accounts. If your students use usernames and passwords to login, make sure they store their usernames and passwords where they can easily access them in the future. Optionally, you may provide students with log-in cards and instruct them to store the cards in a safe place.

If a student forgets their password, you can reset it by visiting the classroom dashboard, locating the student’s username, and clicking on the edit button which will appear if you hover over the username.

## PLAYTIME (1) - 3 MIN

Once students have navigated to Banana Tales Part 1, Challenge 1 and watched the intro video, ask them to take a moment to familiarize themselves with the interface. The game world is represented on the right and the area for writing code is on the left. The console and level_setup.py tabs will be introduced later; for the moment students only need to concern themselves with the solution.py tab where they can write their code and the Run! button that will cause it to execute. For Challenge 1 they don't even need to do that. All they need to do is click the car and it will start driving towards the monkey. Since the path is clear, that results in a banana for the monkey and a three-star solution for the player.

Things get a little more interesting in Challenge 2. Have students try the same thing that worked in the last challenge - just clicking on the car to send it on its way - and see what happens. Since that doesn't solve the challenge, they will need to click on the Rewind button to try again. This time it is actually necessary to use some code to solve the challenge. The correct code has been provided, but students will need to click the Run! button to make it execute. Aha! Now the car has a safe path to the monkey, but it is still necessary to click on the car to send it on its way and win the challenge. Make sure students understand that for every challenge from here on out they will always have to run their code first and then click the car to test their solution and end the challenge.

## EXPLANATION - 5 MIN

So, what is the code in Challenge 2 doing? The seemingly very simple statement `giraffe.height = 6` actually involves a couple of pretty heavy ideas in computer science, so you want to break this explanation down very carefully for your students.

In this statement, `giraffe` represents an *object*. Object has a special meaning in computer science. The following definition is a bit oversimplified, but it will do for right now. An object is a collection of code and data intended to represent something. In this case, the `giraffe` object represents, well, a giraffe, or at least a version of a giraffe that works in the Banana Tales game world. The `giraffe` object includes code that does things like telling the car not to fall when it drives over the giraffe's head and data like the image files used to draw the giraffe on the screen.

Most of the code and data that makes up the `giraffe` object is hidden away inside the CodeMonkey platform where we can't get to it, but we do have access to part of it. That's where the `.height` comes in. `height` is a **property** of the `giraffe` object. A property of an object is a piece of its data that we, the programmers, have access to. In this case, the `height` property of the `giraffe` object represents the height of the giraffe we see on screen. The statement `giraffe.height = 6` says to change the height of the giraffe to 6 units, and when it executes, we can see the change happen.

The dot between `giraffe` and `height` is important. The name of the object goes before the dot, the name of the property goes after the dot. The analogy is not perfect, but the dot in Python works a lot like the 's for possessives in English. `giraffe.height` essentially means "giraffe's height".

## PLAYTIME (2) - 15 MIN

At this point students should continue working through the challenges on their own. For this lesson, they need to complete Challenges 1 - 10 and get three stars on each.

- For Challenges 5, 8, and 9, students will need to drag extra giraffes and/or snakes onto the screen to solve the challenge. It is easy to miss that these are available if you are not looking for them. When there are draggable objects available, the bank for them will appear in the top middle of the screen.
- The snake object, introduced in Challenge 6, has a `length` property instead of `height` and stretches horizontally instead of vertically but otherwise works very much like the giraffe.
- For Challenge 9, it is possible to solve by using the snake or the giraffes, or a combination of both. Encourage students to try and find all three solutions.
- In challenges like challenge 9 where there are multiple objects of the same type, they will have names like `giraffe_1`, `giraffe_2`, etc. Students can see these names by hovering their mouse over the object on screen, and if they click on the object its name will be automatically be copied into the code window.

# Debrief

**EXPLORATION - 5 MIN**

Have the students replay Challenge 9. Ask them to experiment and try to answer the following questions.

- What is the maximum and minimum height of a giraffe?
- What is the maximum and minimum length of a snake?
- What happens when a giraffe is too tall for the space it is in?
- Do giraffes have a `length`?
- Do snakes have a `height`?

Note that investigating these questions will generally result in code that produces an error message and/or that fails the challenge. Make sure that students understand that that is okay; Banana Tales is a tool to help us learn about coding in Python, and sometimes you can learn more from an error than a success.

# Lesson 2 - Sequencing

## CSTA Standards

| Elementary (3 - 5) | Middle (6 - 8) | High (9 - 12) |
|---|---|---|
| 1B-AP-08<br>1B-AP-15 | 2-AP-17 | 3B-AP-11<br>3B-AP-21 |

## Objectives

- Define debugging
- Explain why the order of execution of the steps in a program matters
- Review concepts from previous lesson
- Complete challenges 11 - 18

## Components

**PYTHON**

- Sequential execution

## Introduction - 10 min

**ACTIVITY - 7 MIN**

For this activity you will need a closeable bottle filled with some kind of beverage. The following instructions will refer to water, but you can use whatever works. You also need several cups.

Present the students with the bottle and a cup and tell them that we want to come up with a step by step procedure for drinking the water from the cup. Take a moment and brainstorm the steps with the class. Don't let this take too long; for this activity the steps do not need to be overly detailed. You should end up with something like this:

- Open the bottle
- Pour the water into the cup
- Drink the water from the cup

Have a student follow the instructions to demonstrate that they work. Unless you have extra bottles or the means to refill, make sure they don't pour out all of the water.

Now write (or have a student write) the steps on a sheet of paper and cut them apart. Scramble the steps so they are not in the original order. Let another student attempt to follow the reordered instructions (For the sake of hygiene, use a new cup). Most likely they will not succeed in drinking the water. Let a couple more students try with different permutations of the steps.

### DISCUSSION - 3 MIN

Ask the students if the order of the steps in the procedure matters. They will say of course. Then ask them how they knew what order to do the steps in when they were originally writing the procedure. This is a fairly deep question. They may have insights on their own, but the main idea you want to them to get is that ordering the steps contains an element of working backwards: To drink the water from the cup, the water has to get there first, and to get the water into the cup, the bottle has to be opened first.

## PLAYTIME - 30 MIN

### LOG-IN - 2 MIN

Students should log-in to their accounts as usual.

### PLAYTIME (1) - 6 MIN

When students begin challenge 12, have them read the code that is already there and make a prediction about what is going to happen. Have them write their predictions down rather than saying them out loud. Students should then run the code to test their predictions.

Ask one or more students to explain why the given code failed to solve the challenge (The giraffe was in the way when the snake was stretching). Then ask someone what can be done to fix the problem (Switch the order of the two steps).

Let students implement this solution and get their three stars. Then explain that the process of looking at code that does not do what you want it to and figuring out what can be done to fix it is called **debugging**. As this challenge illustrates, debugging often involves making sure that each step is in the proper order and that nothing is left out.

Now is a good time to point out to students the button with the counterclockwise arrow located to the left of the Run! button. If they ever get ahead of themselves or get lost and need to restore the default code for a challenge, they can click this button to do so.

### PLAYTIME (2) - 22 MIN

Students should work on Challenges 11 - 18 and try to complete each with three stars.

- In Challenges 15, 16, and 18 there are additional snake and giraffe objects that can be added to the game area by drag and drop. In some cases, using the additional animals is necessary, in some cases not.
- Remind students that they can always find the name of an object by hovering their mouse over it and that clicking on an object will copy the name to the code area.
- The three-star solution to Challenge 15 is four lines of code and involves using only the two snakes and two giraffes that are already on the screen. Nonetheless, the availability of the additional animals in the drag and drop area means there are a number of other possible solutions. Challenge students to find as many different solutions as possible. Alternative solutions that use more animals and more lines of code will not receive three stars but trying to find them is still a worthwhile exercise.
- There are also multiple solutions possible for Challenge 18, though they are less interesting than the ones for Challenge 15.

## Debrief - 5 min

Have students share and compare the alternate solutions they found to Challenges 15 and 18.

# Lesson 3 - Lists

## CSTA Standards

| Elementary (3 - 5) | Middle (6 - 8) | High (9 - 12) |
|---|---|---|
| 1B-AP-15 | 2-AP-17 | 3A-AP-14<br>3B-AP-12<br>3B-AP-21 |

## Objectives

- Define list and index
- Use index syntax to refer to list elements
- Complete Challenges 19 - 25

## Components

**PYTHON**

- Function call syntax
- Lists

**PLATFORM**

- Whale objects

## Introduction - 5 min

**ACTIVITY - 5 MIN**

For this activity you will need to go outside or some space in your classroom. It does not have to be a large space, but there needs to be enough room for at least three students to move around. Use colored paper to create several “targets” in the play area.

Choose several students to be the Walkers and one to be the Programmer. You should use as many students as you have target papers; at least three but not more than five. Start the Walkers at random positions in the play area. The

Programmer needs to give the Walkers instructions to navigate to the targets. They should use directions like “turn left”, “turn right”, “take three steps forward”, etc. In giving the instructions, the Programmer must address each Walker by name, e.g. “Alex, turn right. Shannon, take four steps forward.”

Play this game until the all of the Walkers have reached a target. Then (if your numbers allow), choose a new set of Walkers and a new Programmer. This time give each walker a sign with a number, starting with 0. So if you have three Walkers, you will use signs with the numbers 0, 1, and 2. This second round works the same as the first but instead of addressing students by name the Programmer will refer to them by number, i.e. “Student 0”, “Student 1”, etc.

### DISCUSSION - 1 MIN

Explain to students that today’s lesson is about lists. Lists are a way of grouping similar objects together, so we don’t have to give every single one its own name. Just as we did during the activity, when we write code, we refer to items on a list using the name of the list (like “Students”) and a number.

## Playtime – 35 min

### LOG-IN - 2 MIN

Students should log-in to their accounts as usual.

### PLAYTIME (1) - 3 MIN

Before students run the starter code for Challenge 19, ask them to click on the car to see what happens without any code. They should notice that while the whale by itself does have an impact (the car can drive on top of it), an inactive whale isn’t of much use in getting the banana to the monkey.

Now let students run the code and launch the car again. This time the whale sends up a waterspout that lifts the car.

### EXPLANATION (1) - 5 MIN

Explain to the students that the whale is a new type of object. Recall that we said that in computer science an object is a collection of code and data used to represent something. When we worked with the giraffe and snake objects in the previous sections, we used their `height` and `length` properties to change how they were represented on the screen. The property “remembers” the height or length and when we change it, it affects the object.

The whale object doesn’t have height or length properties. Instead, it has a `blow()` **method**. A method is a specific piece of code that is part of an object that makes it do something. In our case, the `blow()` method instructs the whale object to spray a waterspout above its head.

The `blow()` method does not “remember” the height of the water, it just performs the action on a whale.

Notice the syntax for a method. Like a property, it is connected to the object by a dot. Unlike a property, the method name is followed by parentheses. Inside the

parentheses is a value called the **argument** of the method. Argument probably seems like kind of a strange word here, but in this context, it just means an input that tells the method how to do its job. The argument to the `blow()` method is a number that tells the whale how tall to make the water spout.

### PLAYTIME (2) - 3 MIN

Have students move on to Challenge 20. Let students solve this challenge and then move into the explanation.

### EXPLANATION (2) - 5 MIN

Notice how the two whales are referred to here. In the past when we had multiple objects of the same type, each one had its own name, like `giraffe_2` and `giraffe_3`.

Here the plural word `whales` refers to a new kind of thing, a **list**. A list is pretty much just what it sounds like: an ordered collection of things. In this case, the `whales` list contains two separate whale objects. We can refer to them individually by using an **index** in square brackets following the name `whales`. The index counts from the beginning of the list, except the counting starts at 0. So the first whale in the `whales` list is `whales[0]` and the second whale is `whales[1]`.

Make sure students understand this distinction: In a name like `whale_2`, the underscore and the number have no special meaning to Python. `whale_2` is just a name that happens to have an underscore and a number. But when they see something like `whales[2]`, the square brackets and number do have a special meaning. They mean that we are referring to the third item (count 0, 1, 2) in a list called `whales`.

(To be clear, it is not the fact that `whales` is plural that makes it a list. But using a plural name is a good way to remind ourselves that `whales` is a list that contains multiple objects.)

### PLAYTIME (3) - 17 MIN

Let students complete Challenges 21 - 25 on their own. They will continue to use lists to refer to the whales and the `blow()` method to create water spouts of the appropriate height.

- In Challenges 21 and 22 there are columns of obstacles that block the water spout if it goes too high, so students will have to choose the arguments for their `blow()` methods carefully. If they have trouble, remind them that a grid appears when the mouse is over the play area and they can just count the squares to find the right height.
- In Challenge 23, students will need to add whales from the drag and drop area.
- Challenges 24 and 25 use giraffes and snakes instead of whales. The syntax for referring for an individual animal that is part of the list is the same, but you may need to remind students that giraffes and snakes have `height` and `length` properties instead of a `blow()` method. So the command to change the height of the leftmost giraffe in Challenge 24 would be `giraffes[0].height = 7`, for example.

# Debrief - 5 min

Get students to pull up their solutions to Challenge 23. There should be four whales on the screen. Ask students how to refer to each one in Python code (`whales[0]`, `whales[1]`, `whales[2]`, and `whales[3]`, though which is which will depend on the order in which the student dragged them onto the play area).

Have students add the line `whales[4].blow(5)` to their Challenge 23 solutions. What happens? Ask a student to explain what the error message means (`whales[4]` refers to the fifth whale in the list but there are only four there).

Ask students how their code would be different if Challenge 23 were much wider and it took twenty whales to reach across the screen. Would it be inconvenient to call the `blow()` method individually for that many whales? End with a promise that the next lesson will show them how to make jobs like that much easier.

# Lesson 4 - For Loops

## CSTA Standards

| Elementary (3 - 5) | Middle (6 - 8) | High (9 - 12) |
|---|---|---|
| 1B-AP-08<br>1B-AP-09<br>1B-AP-10<br>1B-AP-15<br>1B-AP-17 | 2-AP-11<br>2-AP-12<br>2-AP-17<br>2-AP-19 | 3A-AP-14<br>3A-AP-15<br>3A-AP-23<br>3B-AP-11<br>3B-AP-12<br>3B-AP-21 |

## Objectives

- Use `for` loops to iterate through lists
- Review concepts from previous lessons
- Complete Challenges 26 - 30

## Components

### PYTHON

- `for` loops
- Indentation
- Comments

## Introduction - 5 min

### ACTIVITY - 5 MIN

This activity is similar to the activity in the previous lesson. You will need to clear a play area with room for several students to move and colored paper to use as targets. This time, arrange the colored papers in a row at one end of the play area.

For this version of the activity it is better to have five or six Walkers and Targets, but as few as three can still work.

This time you will need Walkers, a Programmer, and a Recorder. The Recorder's job is to record the instructions that the Programmer gives. Give the Walkers signs (starting with 0) and line them up exactly opposite the target.

Let the Programmer direct the Walkers while the recorder writes down their instructions. Remind the Programmer that the numbers on the signs represent the list index of each student and that they still must address each student individually using the appropriate index. They should come up with something like:

```
Student 0 take six steps forward
Student 1 take six steps forward
Student 2 take six steps forward
Student 3 take six steps forward
Student 4 take six steps forward
```

Now ask the class to look at this list of instructions and think if there is any way to simplify it. The students may have a number of suggestions, but someone will probably say "Why can't we just say, 'Every student take six steps forward'?" Explain that in real life we easily could, but in programming it's a little trickier because Python can only do one thing at once. Nonetheless, there is a structure in Python called a `for` loop that makes it easy to repeat the same instructions for every item in a list, and that's what today's lesson is about.

# Playtime – 30 min

### LOG-IN - 2 MIN

Students should log-in to their accounts as usual.

### PLAYTIME (1) - 6 MIN

Let students run the starter code for Challenge 26 and test it by launching the car. They can make the straightforward fix to the code in a moment, but first help them understand the new elements of the Python code presented here.

This is an example of a `for` loop. In computer science, a loop is a control structure that tells the computer to repeat certain code over and over. There are different types of loop structures, and they determine how many times to repeat the loop in different ways. In Python, a `for` loop is used to repeat the code one time for each element in the list.

Look at the first line of the code: `for whale in whales:` The keywords `for` and `in` and the colon and the end of the line are part of Python syntax and will be the same for every `for` loop. The `whales` after the word `in` refers to list we intend to loop through, and `whale` is the name of a **variable** that changes each time the loop repeats. In other words, each time through the loop, `whale` will represent a different object that is part of the `whales` list.

The second line is the code that will be repeated by the loop. Notice how it is indented several spaces relative to the `for`. In Python, indentation is used to group code together. If we wanted to have several lines of code repeat within the loop, we would have to indent each one the same amount.

**PLAYTIME (2) - 22 MIN**

Students should work to complete Challenges 26 - 30 with three stars.

- Challenge 29 is an assessment challenge that asks students to write all of the code from scratch. If they get stuck, have them look back at their solutions to the previous challenges to remind themselves of the syntax for a `for` loop.
- The lines that begin with #s in Challenge 30 are **comments**. Comments are written in computer code for humans to read; the computer ignores them when running the code. They are useful when programmers are working together to communicate with each other about what a particular piece of code does, or even for a programmer working alone as notes to her or himself. In Python, any line beginning with # is a comment. Comments don't count against students' line totals when it comes to earning stars.

# Debrief - 10 min

Have students go back to Challenge 23 and try to solve it with a `for` loop. If they were successful with the other challenges in this section, then it should be fairly easy. If they need help, here is a working loop solution to Challenge 23:

```
for whale in whales:
    whale.blow(6)
```

Now have them go to Challenge 22 and try to write a `for` loop-based solution to it. They will probably struggle with it unless they have previous experience with Python outside of this course. Give them a few minutes to experiment then call time out. Ask them what it is about Challenge 22 that makes it so much harder to use a loop than in Challenge 23. They may have a variety of answers, but the main factor is that in Challenge 22 each whale has to `blow()` to a different height but in a loop the code is executed the same way each time. Later in the course we will learn several ways to remove that limitation and make loops much more versatile.

# Lesson 5 - Range

## CSTA Standards

| Elementary (3 - 5) | Middle (6 - 8) | High (9 - 12) |
|---|---|---|
| 1B-AP-09<br>1B-AP-10<br>1B-AP-15<br>1B-AP-17 | 2-AP-11<br>2-AP-12<br>2-AP-17<br>2-AP-19 | 3A-AP-14<br>3A-AP-15<br>3A-AP-23<br>3B-AP-12<br>3B-AP-21 |

## Objectives

- Use the `range()` function for iteration
- Use the `print()` function to display the values of variables
- Complete Challenges 31 - 40

## Components

### PYTHON

- `print()` and `range()` functions
- Function call syntax with multiple arguments

### PLATFORM

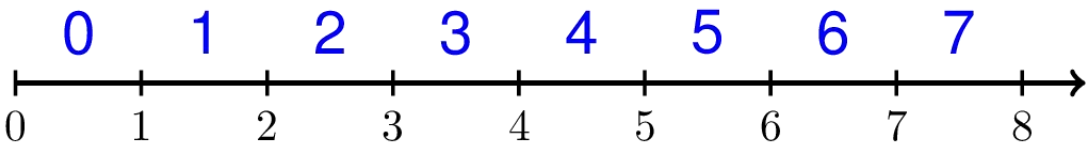
- The console tab

## Introduction – 5 min

### EXPLANATION - 5 MIN

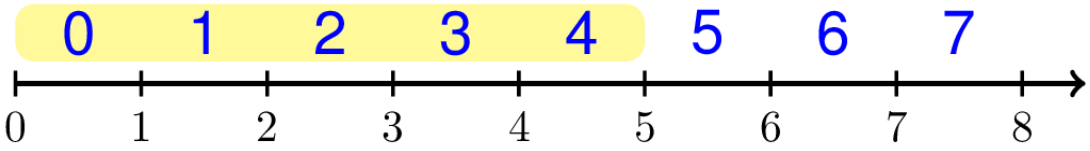
This lesson introduces several functions. Explain to students that a function, like a method, is a piece of code that does a specific job, except it is not tied to a particular object. The main focus of this lesson is a using the class `range()`.

Start by displaying a labeled number line like the one below:




Explain that the `range()` is used to generate a sequence of numbers. The number line model will be used to illustrate how `range()` works. In the number line model, inputs to the `range()` correspond with the numbers under the hash marks and outputs correspond to the numbers above the line between the marks.

The simplest way to use `range()` is with a single argument. In that case `range()` returns a sequence of integers starting at 0 and going up through one less than the argument. So `range(5)` would return the numbers: `0, 1, 2, 3, 4`. The fact that 5 is not included in the list will probably bother students, but the logic of it can be explained with the model.




Remind students that inputs are on the bottom and outputs are on top. `range(5)` means start shading from the left (at hash mark 0) and stop at hash mark 5. This means that the numbers that get shaded are 0, 1, 2, 3, and 4. 5 is not included in the output because it is to the right of the 5 hash mark.

This model will probably help some but not all students, but it should give them a foundation to build on as they approach the first few challenges.

# Playtime – 35 min

### LOG-IN - 2 MIN

Students should log-in to their accounts as usual.

### PLAYTIME (1) - 13 MIN

A lot of new things are introduced in Challenge 31. To avoid overwhelming students, they only need to change the giraffes' heights. As usual, have students run the code as given first, then review the following elements with them before letting them complete the challenge.

Lines 2 - 4 use the `print()` function and `range()`. The `print()` function displays data as text in the **console** window below the code area. Make sure students note that when they want to display literal text as on lines 2 and 4 they have to put it in quotation marks.

As discussed in the Introduction, if `range()` is given a single argument, the sequence consists of all of the integers from 0 to one less than the argument. So `range(6)` creates the sequence `0, 1, 2, 3, 4, 5`.

`range()` is often used with `for` loops. The `for` loop that begins on line 5 will execute six times, with `index` taking the value 0, then 1, then 2, then 3, then 4 and then 5. Inside the loop the `index` variable is used to refer to the different elements of the `giraffes` lists.

Also make sure students notice that this time the code inside the `for` loop contains multiple statements and that they are all indented the same amount.

Once you have completed the code walk through, let the students complete Challenge 31 and then complete 32 - 35 on their own. These challenges follow the same model as 31 but each requires students to write more code on their own.

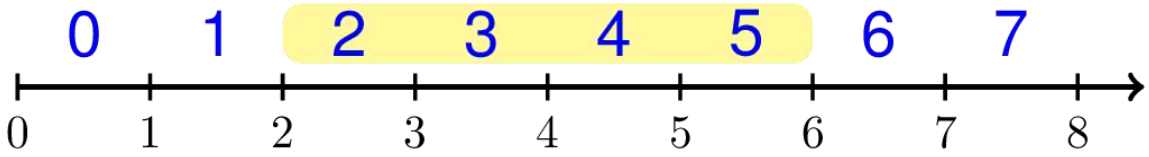
**PLAYTIME (2) - 20 MIN**

Before students get to work on Challenge 36, ask them if all of the `giraffes` have room to change their `height`. They should see that the answer is no. Then ask them which ones do have room. Make them give their answer in Python terms. In other words, they should say `giraffes[3]`, `giraffes[4]`, `giraffes[5]`, and `giraffes[6]`.

This challenge illustrates another form of `range()` that can be used to create a sequence of these indices. When `range()` is given two arguments inside the parentheses separated by a comma (e.g. `range(3, 7)`), the first argument is the integer to start at and the second is one less than the final number. So `range(3, 7)` creates the sequence `3, 4, 5, 6`, exactly the indices we need for the giraffes in this challenge.

In fact, `range(7)` is the same as `range(0, 7)`, but since the default is to start from 0, we can use range with just one argument.

The number line model extends quite naturally to two-argument `range()`. The shading goes from the left of the hash mark corresponding to the first argument to the right of the hash mark corresponding to the second.




Once students understand the new `range()` syntax, let them complete the challenge.

Challenge 39 is an assessment challenge for the two-argument `range()` syntax. If students read the instruction text it shouldn't be too hard.

# Debrief - 5 min

Have students write down the answers to these questions to check their understanding of `range()`:

- What is produced by the statement `range(5)`?
- What is produced by the statement `range(7)`?
- What is produced by the statement `range(2, 7)`?

- What is produced by the statement `range(4, 9)`?

The answers are, respectively:

- `0, 1, 2, 3, 4`
- `0, 1, 2, 3, 4, 5, 6`
- `2, 3, 4, 5, 6`
- `4, 5, 6, 7, 8`

Discuss and review using the number line model as necessary.

# Lesson 6 - Variables

## CSTA Standards

| Elementary (3 - 5) | Middle (6 - 8) | High (9 - 12) |
|---|---|---|
| 1B-AP-08<br>1B-AP-09<br>1B-AP-10<br>1B-AP-15<br>1B-AP-17 | 2-AP-11<br>2-AP-12<br>2-AP-17<br>2-AP-19 | 3A-AP-14<br>3A-AP-15<br>3A-AP-23<br>3B-AP-11<br>3B-AP-12<br>3B-AP-21 |

## Objectives

- Define variable
- Use a variable to store a numerical value
- Assign to a variable using an arithmetic expression
- Complete Challenges 41 - 47
-

## Components

**PYTHON**

- `len()` function
-

## Introduction - 10 min

**EXPLANATION - 10 MIN**

Ask students what they know about the word *variable*. Depending on their backgrounds you might hear a lot of different ideas. Students who have had algebra or pre-algebra may talk about variables in math and solving equations.

Some may refer to the variables that are used in `for` loops that were introduced in the last lesson. Many may have had enough exposure to coding so they'll say something about a variable being a place to store numbers or other kinds of data.

Build on students' existing knowledge to make sure they understand the following ideas about variables. A variable is a place in the computer's memory for storing some kind of data. The data could be a number, a text string, a list, a code and data object like a giraffe, or any of a number of other types that Python understands.

Variables have names that we can use in our code. A pretty good way to think of a variable is as a box with a name stamped on the front of it. The box provides the storage space and the name gives us a way to refer to it.

Variables get created and named in a number of different ways. In most of the challenges so far, there have been different kinds of animal objects that were automatically created and named by the CodeMonkey system. If a challenge had a single giraffe on the screen, the system gave us a pre-made variable called `giraffe` so we could refer to it.

We've also seen variables created by `for` loops. In code like:

```
for snake in snakes:
    snake.length = 2
```

we are telling Python to create a new variable called `snake` and make sure it contains the next item from the `snakes` list each time through the loop.

We can also create our own variables. The statement `goal = 3` creates a storage space, gives it the name `goal`, and stores the number 3 in that space. Then later in our code we can use `goal` anywhere a number would go.

# Playtime - 30 min

**LOG-IN - 2 MIN**

Students should log-in to their accounts as usual.

**PLAYTIME - 28 MIN**

Look at challenge 41 together with your students. Point out that it is quite similar to challenges from the last lesson; as before we are using a for loop to set multiple snake to the right length to create a path for the car. The difference is that in this case a variable is used to represent that length. Point out how the variable goal_length is created at set on line 2 and used to set the lengths of the other snakes on line 7.

Let students test the code as written and then debug it. The only fix needed is the index on line 6 needs to be changed to 3. Then have students complete Challenge 42, which is very similar.

Ask them what the advantage of using the goal variable like this is. There may be a number of possible answers but one is that since `giraffes[0]` is already the

"right" height, then using the goal variable saves us the trouble of counting out the distances manually. Later challenges in this lesson will illustrate this and other ways variables can make code more versatile.

Challenge 43 is a one liner; students just need to change the argument of the `blow()` method. But make sure that they notice use of the `len()` function on lines 2 and 4. Explain that `len()` takes a list as an input and returns the number of elements in the list as an output. `len()` is often used as the input to `range()` as on line 4 because that means the list of numbers returned will be exactly what is needed for indexing the list.

Challenge 44 is a new format. There are two play areas with two baby monkeys on the right. The goal is to write one program that solves both challenges. Students should test their code in each play area without making any changes in between. This challenge is intended to illustrate how using a `goal_height` variable to set the heights of all of the giraffes allows the same code to be used for both parts of the challenge.

Challenges 45 through 47 ask students to use a variable called `height` to set the heights of the giraffes inside of the `for` loop, but add the additional wrinkle of that variable changing each time through the loop. This is another way that variables allow us to write more flexible code. In

# Debrief - 5 min

Review with your students what they learned in this lesson.

Show your students the following sets of code and ask them to name the variables and their values:

```
height = 4
for giraffe in giraffes:
    giraffe.height = height
    height = height + 1
```

- variables are:
    - height – its value is 4; on each iteration its value increases by 1
    - giraffe – on each iteration its value changes and gets the next giraffe in the giraffes list

```
for index in range(5):
    giraffes[index].height = 4
```

- variable is:
    - index – its initial value is 0; on each iteration its value increases by 1 until it reaches 4

# Lesson 7 - If

## CSTA Standards

| Elementary (3 - 5) | Middle (6 - 8) | High (9 - 12) |
|---|---|---|
| 1B-AP-09<br>1B-AP-10<br>1B-AP-15<br>1B-AP-17 | 2-AP-11<br>2-AP-12<br>2-AP-17<br>2-AP-19 | 3A-AP-14<br>3A-AP-15<br>3A-AP-23<br>3B-AP-12<br>3B-AP-21 |

## Objectives

- Define conditional
- Use `if` statements to choose which objects to perform an action with during iteration
- Complete Challenges 48 - 53

## Components

**PYTHON**

- `if` statements

**PLATFORM**

- Dragon and obstacle objects
- `fire_at()`, `smash()`, and `is_ice()` methods

## Introduction - 7 min

**ACTIVITY - 7 MIN**

Arrange a few of your students (at least six, if possible) in a line. Choose one additional student who is not in the line. Display the following pseudo-Python code and ask them to do what it says:

```
for student in the line:
    give student a high five
```

The chosen student should of course high five every student in the line. Now display this code and have another student carry it out:

```
for student in line:
    if student is wearing a blue shirt:
        give student a high five
```

Replace “is wearing a blue shirt” with any description that is fairly obvious and that applies to approximately half of the students in the line. It is best not to use students’ gender or physical appearance for this. If you can provide some of the students with silly props to wear (novelty sunglasses or hats, etc.) and base the condition on these that would be ideal.

If you have multiple different props you may want to repeat this exercise a couple of times with different conditions for the “if” part, but that’s not absolutely necessary. Conclude the activity be explaining that the lesson today is about learning how to use the `if` statement in Python to choose whether or not to do actions on items in a list based on their properties.

# Playtime - 31 min

### LOG-IN - 2 MIN

Students should log-in to their accounts as usual.

### PLAYTIME (1) - 15 MIN

Let students work on Challenges 48 - 50 on their own. These challenges introduce the dragon character and the types of obstacles he can remove. Students need to notice that the dragon has two main tricks - he can breathe fire and he can smash things - each with a corresponding method. These methods each take one argument, the object that the dragon is attacking. Make sure that students test what happens when they use the “wrong” method on an object; they should notice that if they `fire_at()` a box or `smash()` an ice block it doesn’t remove it as an obstacle.

In Challenge 48 the obstacles all have individual names so students will have to write one line of code for each obstacle. In Challenges 49 and 50, the students will have to write a `for` loop to earn three stars. Do not let students move on until they have received three stars on each of these challenges.

### PLAYTIME (2) 14 MIN

Give students a few minutes to work on Challenge 51. This challenge introduces a new concept (the `if` statement) but it may also be just a little tricky to figure out what the dragon needs to do to solve the puzzle. Eliminating every obstacle won’t work because it will leave the banana trapped in the “pit” in the middle of the screen. The same is true of eliminating just the boxes. What does work is eliminating the ice blocks which allows the boxes to drop down and create a path for the banana.

After students have figured out the solution to the puzzle, either by themselves or with some hints, take a few moments to look at and discuss how the `if` statement works. The basic syntax is like this:

```
if CONDITION:
    # One or more statements
    STATEMENT
    STATEMENT
    # etc.
```

`CONDITION` is a special kind of expression that represents a yes or no answer, though in Python we usually express those concepts as `True` or `False`. Some other computer languages spell these keywords `true` and `false` but in Python they have to begin with a capital letter. In Challenge 51, we use the `is_ice()` method of each obstacle which returns `True` if and only if the obstacle is an ice block. The `if` statement in Python works in a way that more or less corresponds with how the word "if" works in English: the statements are executed only if the `CONDITION` is `True`. Otherwise, the statements are skipped over entirely. Students should also notice that just as with the `for` statement, indentation is used to mark which statements are controlled by the `if`.

Draw the students' attention to the fact that the methods is`_ice()`, `is_box()`, `is_fence()` are used with empty parenthesis. When a method does not have any arguments, we use the empty parenthesis.

Once you are confident that students have a handle on how `if` works, you can let them move on to Challenges 52 and 53. Challenge 52 is a variation on the theme of Challenge 51. Make sure students notice that there are draggable items in this challenge; they will need to use boxes to fill in the mini-pits between the banana and the monkey.

Challenge 53 is an assessment challenge. It uses the same basic idea as in the previous challenges - the dragon must only eliminate the ice blocks - but students have to write the code for the loop with the `if` statement from scratch. They can refer to their solutions from the previous challenges for help. This challenge also contains the additional wrinkle that it is necessary to eliminate one of the middle two boxes. Some students may need a reminder that not all of their code needs to be part of the loop. In this case a single `dragon.smash(obstacles[7])` either before or after the loop will get the job done (`dragon.smash(obstacles[5])` works as well).

## Debrief - 7 min

Display the following code:

```
for obstacle in obstacles:
    if obstacle.is_ice():
        dragon.fire_at(obstacle)
```

Have students look back at Challenges 48, 49, and 50. Ask students which of these challenges the given code would successfully solve (not necessarily with

three stars). Once they all agree that Challenge 50 is the only one that can be solved with this code, ask them to explain why the code does not work for the other two.

- For Challenge 48, the obstacles are not part of a list so the `for` loop won't work.
- For Challenge 49, none of the obstacles are ice so the code in the `if` statement will not be executed. And even if it was, `fire_at()` is the wrong method to use for boxes anyway.

Finally, ask students to reconsider Challenge 48. What if the obstacles in this challenge were part of a list? Would the given code work in that case? Why not?

Guide the conversation here. The main idea you want to conclude with here is that the `if` statement is somewhat limited. What we really want is a way to do one action if the condition is true and a different action if the condition is false. That is the focus of the next lesson.

# Lesson 8 - If/Else

## CSTA Standards

| Elementary (3 - 5) | Middle (6 - 8) | High (9 - 12) |
|---|---|---|
| 1B-AP-09<br>1B-AP-10<br>1B-AP-15<br>1B-AP-17 | 2-AP-11<br>2-AP-12<br>2-AP-17<br>2-AP-19 | 3A-AP-14<br>3A-AP-15<br>3A-AP-23<br>3B-AP-12<br>3B-AP-21 |

## Objectives

- Use `if`/`else` statements to choose what action to perform with an object during iteration
- Complete Challenges 54 - 59

## Components

**PYTHON**

- `if`/`else` statements

**PLATFORM**

- `is_fence()` and `is_box()` methods

## Introduction - 7 min

**ACTIVITY - 7 MIN**

This activity is similar to the one from the previous lesson. Arrange a few students (at least six) in a line and choose one additional student to carry out the following procedure:

```
for student in line:
    if student is wearing a green shirt:
```

```
        give student a high five
    otherwise:
        shake the student's hand
```

Ask students to compare this version of the activity to the one from the last lesson. In addition to possibly using a different condition, they should note that this time the procedure includes instructions on what do if a student does not meet the condition.

Do the activity one more time with this pseudo-code. You may change the student who is executing the procedure but keep the students in line the same.

```
for student in line:
    if student is wearing white shoes:
        give student a high five
    otherwise:
        shake the student's hand
```

As before, you can change the conditions of the two procedures to fit your students. It is important that you have some students in the line that meet the first condition, some that meet the second, and some that meet neither. Giving students simple props or even something like colored stickers might make this easier.

After the second run through, point out that while each time everyone in line got either a high five or a handshake, which students got which changed depending on the condition. This is the idea behind today's challenges, which involve choosing the right conditions to have the dragon doing different actions with different obstacles.

# Playtime - 33 min

### LOG-IN - 2 MIN

Students should log-in to their accounts as usual.

### PLAYTIME (1) - 8 MIN

Challenge 54 is mostly a review, but you need to make sure that students earn three stars before they move on. Students will need to use a new method, `is_box()`, in order to have the dragon destroy only the box obstacles. The starter code for this challenge is a little misleading because it suggests writing two lines of code for each obstacle. In order to three star this challenge students will need to use a loop instead.

Once a student has achieved three stars on this level direct them back to Challenge 48. Based on what they see here ask them to predict what other method obstacle objects might have besides `is_ice()` and `is_box()`. Once they have made their prediction, let them move on to Challenge 55.

When students reach Challenge 55 point out the `is_fence()` method and have them check their predictions. Then let them complete the challenge. If anyone has trouble point out that the two `if` statements inside the loop are meant to

choose the appropriate action depending on the type of obstacle. Everything in the given code is correct; to finish all they need to do is fill in the appropriate statements on lines 3 and 5. You may also need to remind students about the draggable objects that are available on this challenge.

**ANALYSIS - 11 MIN**

Once everyone has completed challenges 54 and 55 with three stars, direct them to Challenge 56 and ask them to temporarily ignore the code outline they are given. Instead, they should consider each of the following possible solutions and discuss whether they solve the challenge and why or why not.

```
for obstacle in obstacles:
    dragon.fire_at(obstacle)
```

Here students should recognize that this will not work because `fire_at()` only removes ice blocks.

```
for obstacle in obstacles:
    if obstacle.is_ice():
        dragon.fire_at(obstacle)
```

This is a slight improvement over the previous code. The dragon will not waste its breath firing at obstacles that are not ice. But this code still contains no instructions for removing the boxes.

```
for obstacle in obstacles:
    if obstacle.is_ice():
        dragon.fire_at(obstacle)
    if obstacle.is_box():
        dragon.smash(obstacle)
```

This works. Once all the students recognize this, ask them to consider what it means when `obstacle.is_ice()` is `False`. They need to recognize that since all the obstacles on the screen are either ice blocks or boxes, once you have determined that a particular obstacle is *not* an ice block, you already know it's a box. So in a sense the second `if` is redundant. It would be nice if we had a way to write code that says "if this condition is true, do one thing, otherwise do something else". Not surprisingly, Python provides a way for us to do that.

**PLAYTIME (2) - 12 MIN**

Now let the students return to the starter code for Challenge 56. If they made changes to the code during the discussion, remind them that they can use the button to the left of Run! to reset everything.

This code illustrates the `if`/`else` construct for Python. `if`/`else` does exactly what was described above; it executes one block of code if the condition is true, and a different block if not. The general syntax looks like this:

```
if CONDITION:
    # One or more statements to execute if CONDITION is True
```

```
    STATEMENT
    STATEMENT
    # etc.
else:
    # One or more statements to execute if CONDITION is False
    STATEMENT
    STATEMENT
    # etc.
```

It is important that students understand that either the first (condition `True`) or second (condition `False`) block will always be executed, not both and not neither.

After the explanation, let students tackle Challenge 56. The `if`/`else` syntax is already set up correctly; they just need to set up the loop in line 1 and write an appropriate statement to destroy the ice blocks in line 3. Provide hints as needed, and as soon as students get three stars in this challenge let them move on to the next.

Challenges 57 and 58 provide students with for practice using `if`/`else`. Challenge 57 is similar to 56 except that the smashable objects are fences not boxes. Challenge 58 introduces a small complication by having both types of smashables. Students may be tempted to use `is_box()` or `is_fence()` in their condition but they should heed the instructions and base their solution on `is_ice()` instead. The general principle is that the `if` condition should be the more specific one, as the else must apply to everything else.

Challenge 59 is an assessment challenge, and a two level one at that. Students will have to write code that solves both challenges from scratch, but the ideas are the same as for the previous challenges.

# Debrief - 5 min

Tell students to go to Challenge 59 and then show them the following potential solution:

```
for obstacle in obstacles:
    if obstacle.is_box():
        dragon.smash(obstacle)
    else:
        dragon.fire_at(obstacle)
```

Ask students if this will work and why or why not. Ideally, they will be able to answer without running the code, but if they have trouble let them type it in and try it. They should see that the problem here is that the dragon does the wrong action with fences - it burns instead of smashing them.

The takeaway is that since `if`/`else` only allows the code to go in one of two directions the condition must be carefully chosen so that it is `True` for *all* of the situations that require one code path and `False` for *all* that require the other.

Later lessons will introduce ways to write more elaborate conditions that can make these choices easier.

# Lesson 9 - While Loops

## CSTA Standards

| Elementary (3 - 5) | Middle (6 - 8) | High (9 - 12) |
|---|---|---|
| 1B-AP-09<br>1B-AP-10<br>1B-AP-15<br>1B-AP-17 | 2-AP-11<br>2-AP-12<br>2-AP-17<br>2-AP-19 | 3A-AP-14<br>3A-AP-15<br>3A-AP-23<br>3B-AP-12<br>3B-AP-21 |

## Objectives

- Express a conditional with arithmetic comparison operators
- Use a `while` loop to repeat an action as long as a certain condition is true
- Complete Challenges 60 - 66

## Components

**PYTHON**

- `while` statements

**PLATFORM**

- Elephant and Well objects
- `water_level` and `max_water_level` properties
- `spray_at()` method

## Introduction - 10 min

**ACTIVITY - 9 MIN**

For the activity you will need at least one container and a supply of counters. A glass, cup, or jar is fine for the container (transparent is better) and the counters can be chips, marbles, coins, or something similar. Ideally you can provide a

container and some counters to every student, but the activity can work with just one. You will also need some way to play music that is audible to the entire class.

Provide cups and counters to as many students as possible then display the following pseudo-code:

```
repeat these steps while the music is playing:
    add 1 counter to the cup
    wait 2 seconds
```

If any students ask how they will know how long two seconds is, tell them just to estimate. You may wish to demonstrate counting off "one-one-thousand, two-one-thousand" but don't let concerns about exact timing bog down the activity.

Tell the students to follow the code and then start the music. They should begin adding counters to their cups at approximately two second intervals. After 20 or so seconds stop the music.

Briefly discuss the activity. If you noticed any student who only added one counter to the cup or who kept adding counters after the music had stopped, point out that the first statement tells them to repeat the actions below, but only while the music is playing. If any students were adding counters continually without any pause remind them that because both instructions were indented both should be repeated - add counter, wait; add counter, wait; add counter, wait; etc.

Now have students empty their cups and reset. If you don't have enough supplies for everyone, switch off now and let a new group of students take a turn. This time the students will be following this pseudo-code:

```
repeat these steps while the number of counters in the cup is less than 6:
    add 1 counter to the cup
    wait 2 seconds
```

Tell the students to start. It should only take them 20 seconds or so to complete the procedure. Ask students to count the number of counters in their cup. If they followed the procedure correctly, they should each have six, but it is fairly likely that some of them will have only five and some may possibly have seven.

Ask one or more students who carried out the procedure correctly to explain their thinking. The key idea is that you have to check the number of counters in the cup before you repeat each time. If the number of counters in the cup is five, yes, that is less than six, so you do the steps again and add another counter. When there are six counters in the cup, six is *not* less than six so you do not repeat anymore and the procedure is over.

## EXPLANATION - 1 MIN

Ask students what you call a structure in computer programming that repeats the same steps over and over. The answer is loop, though if a student says `for` loop you can point out that there are other types of loops that use different rules to determine how many times to repeat the steps. Today's lesson is about a new kind of loop called a `while` loop.

# Playtime – 30 min

### LOG-IN - 2 MIN

Students should log-in to their accounts as usual.

### Playtime (1) - 4 min

Let students work through Challenges 60 and 61 on their own. These challenges introduce the elephant and well objects and the `spray_at()` method that causes an elephant to add water to a well. It shouldn't take students very long to complete these challenges, but before you move on make sure they understand the following ideas.

- `spray_at()` does not automatically fill the well; it only adds enough water to raise the level one unit. That is why it is necessary to `spray_at()` multiple times.
- Trying to `spray_at()` a well that is already full causes an error that forces you to Rewind without completing the challenge.

### Explanation - 9 min

Before students move on from Challenge 61, discuss the properties of wells that are introduced there. `max_water_level` never changes. It represents the amount of water the well can hold without causing an error. `water_level` is the current water level. It increases by one each time an elephant sprays water at the well.

Now look at Challenge 62 as a class. This is a two-level challenge. Remind the students that for this type of challenge they have to write code that works for both levels without any changes and ask them why that makes this challenge harder. They should be able to recognize that because the well on the second level is deeper than the well on the first, it will require more `spray_at()` statements to fill. Completing this challenge depends on finding a way to write code that will spray the right number of times for each well.

Ask a student to read the comment and supply the missing statement. To solve the challenge all that is needed is `elephant.spray_at(well)` under the `while` statement. But have students first try this lengthier solution, The extra output is helpful for explaining how the `while` statement works.

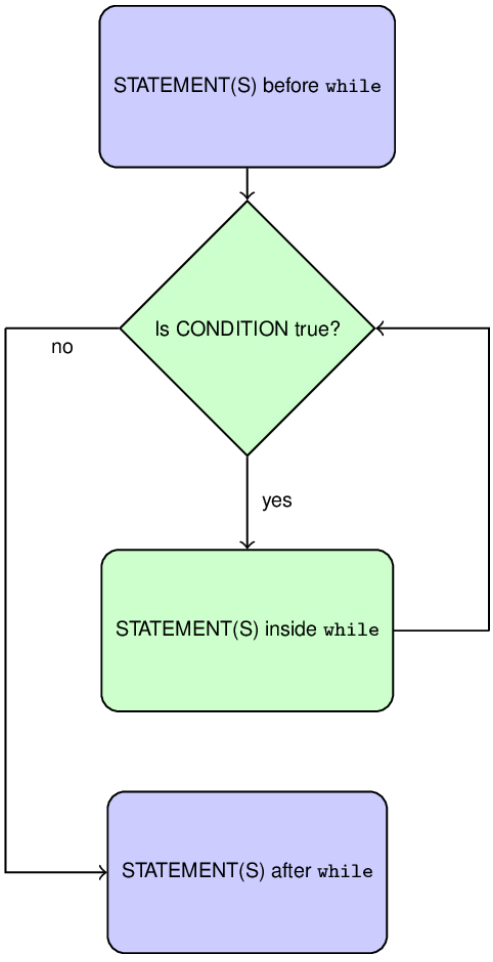
```
print("Max water level is:")

print(well.max_water_level)

print("Water level is:")

print(well.water_level)

while well.water_level < well.max_water_level:

    # The elephant should spray at the well.
```

```
elephant.spray_at(well)
print("Water level is:")
print(well.water_level)
```

The general idea of a `while` statement is simple; it is a new kind of loop that repeats a set of steps over and over as long as a certain condition is true. In this case the condition is a mathematical one: The loop will repeat as long as the water level in the well is less than the maximum possible water level for the well. That means that when water level is *equal to* the maximum water level then the loop will *not* repeat again. That is exactly what we want, because if the well is already full, we don't want to add any more water.

It is important to understand that the condition of the `while` statement is checked before every time through the loop (including the first time), but not in the middle of the loop. In the code above, `well.water_level` changes as soon as the `spray_at()` method is executed, but the next two `print()` statements happen no matter what since the condition is only checked again once all of the statements have been executed.

The flowchart below illustrates how `while` loops work in general:




**PLAYTIME (2) - 15 MIN**

Challenges 63 and 64 are a bit more difficult than usual, at least as far as earning three stars goes. But these are good challenges for students to grapple with on their own. They will need to use `while` loops as in the previous challenge to fill each well the right amount, but they will also need to use a `for` loop to write the code to fill multiple wells in the fewest number of lines.

Challenges 63 - 66 gradually reduce the scaffolding and increase the difficulty as students practice setting up and using `while` loops.

- Challenge 63 can be solved by simply duplicating the given code and changing the list indices to 0, but in order to earn three stars students will have to use a `for` loop. If they get hung up on this tell them to move on then come back later to pick up the third star.
- Challenge 64 provides a template for using the elephant `while` loop inside of a `for`. Point out the use of the `len()` function inside of `range()`. This makes the code more flexible; it will work for lists of any length.
- Challenge 65 is similar to 66 except students have to write all the code themselves. They should refer back as necessary.
- On Challenge 66, the order in which the draggable elephants are added matters. The first elephant should be added to the central ledge, the second to the ledge on the right. This is important so the indices of elephants and wells match.

# Debrief - 5 min

Display each of the following code snippets:

*IF*

```
if CONDITION:
    # One or more statements
    STATEMENT
    STATEMENT
    # etc.
```

*FOR*

```
for VARIABLE in LIST:
    # One or more statements
    STATEMENT
    STATEMENT
    # etc.
```

*WHILE*

```
while CONDITION:
    # One or more statements
    STATEMENT
    STATEMENT
    # etc.
```

Ask students to compare and contrast these types of statements. Similarities they might point out are that they all use a colon, they all use indentation, and they all involve one or more statements. Going deeper, they might observe that each of them is a way to control if and how many times the statement(s) are executed. `if` executes the statements or not depending on the condition. `for` executes the statements for each element in the list. And `while` executes the statements over and over again as long as the condition is true.

Explain that all of these are examples of what are called **control structures**. Every programming language has some kinds of control structures. While there are a few more control structures in Python, at this point students have met all of the most important ones.

# Lesson 10 - Boolean Operators 1

## CSTA Standards

| Elementary (3 - 5) | Middle (6 - 8) | High (9 - 12) |
|---|---|---|
| 1B-AP-09<br>1B-AP-10<br>1B-AP-15<br>1B-AP-17 | 2-AP-11<br>2-AP-12<br>2-AP-17<br>2-AP-19 | 3A-AP-14<br>3A-AP-15<br>3A-AP-23<br>3B-AP-12<br>3B-AP-21 |

## Objectives

- Use `and` and `or` operators to write compound conditional statements
- Solve complex problems using compound conditionals and multiple `if` statements
- Complete Challenges 67 - 74

## Components

**PYTHON**

- `and` and `or` operators

**PLATFORM**

- `is_on_ground()` method

## Introduction - 10 min

**ACTIVITY - 5 MIN**

For this activity you need a note card for each of your students. Each card needs to show one of three shapes (triangle, square, circle) in one of two colors (red and blue). So, there should be a total of six different kinds of cards, with approximately equal numbers of each. You can either prepare these in advance or if you have a

little extra time have your students create them. Either way, each student should end up with one card, again with all six types represented more or less equally.

Explain that you are going to give instructions for certain students to stand based on what is on their card. When they stand, they should hold up their card so the whole class can see. After each round of the activity everyone should sit back down again to prepare for the next instruction.

Give the following instructions in this order. If you notice any students standing when they should not be or vice versa gently correct their error.

- If your picture is a triangle, stand up
- If your picture is blue, stand up
- If your picture is red and is a circle, stand up
- If your picture is a square and is blue, stand up
- If your picture is a triangle or is a square, stand up
- If your picture is red or is blue, stand up
- If your picture is red or is a square, stand up

After these seven rounds, sit the students down for the explanation.

**EXPLANATION - 5 MIN**

Point out that each of the instructions in the activity was phrased as an "if" statement. Just as with an `if` statement in Python, each of these was based on a condition, a fact that could either be true or false. In the first couple of instructions, the condition was simple, e.g. "your picture is a triangle". But the later instructions used the words "and" and "or" to combine simple conditions into more complex ones.

Ask students what it means if you combine two conditions with the word "and". The answer should be relatively clear from the everyday meaning of the word. Both individual conditions have to be true for the whole statement to be true. When the instruction was "If your picture is a square and is blue, stand up", only the people with blue squares stood up.

Now ask about "or". Again, this is pretty easy though there is one little subtlety that can throw people off sometimes. In English the word "or" is typically used to refer to things that are mutually exclusive. For example, if the menu says the value meal comes with French fries or onion rings, it means you get one or the other, not both. And asking an "or" question when both conditions are true usually sounds strange: Do humans need oxygen or food?

Computers interpret the word `or` differently. A condition made with the keyword `or` is `True` if either part is `True`. If you are talking about conditions that can't overlap, like "is a triangle or is a square", then that's easy. But both conditions can be true at the same time, like "is red or is a square", that's kind of different from how we usually use the word "or" and can be confusing. But in computer logic, an `or` statement is considered true if either of its parts are true or if both parts are true.

# Playtime - 29 min

**LOG-IN - 2 MIN**

Students should log-in to their accounts as usual.

**PLAYTIME - 27 MIN**

Students should be able to play through Challenges 67 - 74 on their own. Take a couple minutes to clarify the following two ideas as they begin the first challenge then circulate providing hints as needed.

- `and` and `or` are operators used to combine conditions in Python. They work just as was discussed during the introduction.
- The `is_on_ground()` method only returns `True` if the obstacle is actually on the ground; sitting on top of another obstacle doesn't count.

Let the students puzzle out the challenges on their own as much as possible but offer hints as necessary.

- Challenge 68 is very similar to 67 except that the dragon must also destroy the fences. This can be done inside the same `while` loop.
- The point about how `is_on_ground()` works is important in Challenge 69. Using the condition `obstacle.is_box() and obstacle.is_on_ground()` only selects the bottom box in each stack, which is what we want.
- Challenge 71 needs a `for` loop. Students may not realize this since it doesn't start with a blank line.
- Challenge 72 should be based on `if`/`else`. While it is possible (and arguably simpler) to solve without using an `or` condition, earning three stars requires it.
- The hint in the comment for Challenge 74 says to use two `if` statements. One should be used to deal with the ice on the ground, the other for the boxes and fences.

# Debrief - 6 min

Display the table below with the entries in the final column blank. Explain that this is called a truth table for the `and` operator. The last column is intended to show whether the whole condition is `True` or `False` based on the two conditions on either side of the `and`. Go through the table line by line and fill it in with your class.

| • CONDITION_A | • CONDITION_B | • CONDITION_A and CONDITION_B |
|---|---|---|
| • False | • False | • False |
| • True | • False | • False |
| • False | • True | • False |
| • True | • True | • True |

Once you have finished `and`, do the same thing with the truth table for the `or` operator.

| CONDITION_A | CONDITION_B | CONDITION_A or CONDITION_B |
|---|---|---|
| False | False | False |
| True | False | True |
| False | True | True |
| True | True | True |

# Lesson 11 - Boolean Operators 2

## CSTA Standards

| Elementary (3 - 5) | Middle (6 - 8) | High (9 - 12) |
|---|---|---|
| 1B-AP-08<br>1B-AP-09<br>1B-AP-10<br>1B-AP-15<br>1B-AP-17 | 2-AP-11<br>2-AP-12<br>2-AP-17<br>2-AP-19 | 3A-AP-14<br>3A-AP-15<br>3A-AP-23<br>3B-AP-11<br>3B-AP-12<br>3B-AP-21 |

## Objectives

- Write conditionals using the `not` operator
- Solve even more complex problems using compound conditionals including the `not` operator
- Review concepts from lesson 9
- Complete Challenges 75 - 78

## Components

**PYTHON**

- `not` operator

**PLATFORM**

- Crocodile objects
- `mouth_closed` property
- `toggle()` method

## Introduction - 10 min

## ACTIVITY - 6 MIN

Discuss the word “toggle”. Ask students if they have heard the word before and what they know about it. It is important that they understand that toggle means reversing the state of something. A good example to use if students aren’t familiar with the word is a checkbox in a graphical interface. Clicking on a checkbox checks it if it is unchecked or clears it if it is checked.

In the following activity students will toggle between standing at sitting. The instruction toggle (in this context) will mean stand if you are sitting and sit if you are standing.

Have your students count off (starting at 0) and remember their number. Explain that for the activity the whole class is going to be treated as a list and that this number is their index in the list.

Choose approximately half the students at random and have them stand. Then ask the class to carry out the procedure represented by the following pseudo-code:

```
for student in class:
    student.toggle()
    wait 1 second
```

Every student should change positions, acting in the order they counted off earlier. If multiple students move at once, remind them that `for` loops visit each item in the list in order and start the exercise over.

Once the activity has been completed correctly, have students stay in the same positions and show them this pseudo-code:

```
for student in class:
    if student.is_sitting()
        student.toggle()
```

Ask students to make a prediction about what will happen and then have them carry out the procedure. If they followed the instructions correctly everyone should end up standing.

Have about half of the students (at random) sit down then display one last procedure:

```
for student in class:
    if not student.is_sitting()
        student.toggle()
```

Explain that `not` is actually a Python operator, and while the grammar may look strange here, this is how `not` works in Python. It goes in front of a condition and reverses the sense of it. `not student.is_sitting()` would be `False` when `student.is_sitting()` is `True` and vice versa.

Let students do the procedure. If they do it correctly it should finish (conveniently enough) with everyone seated.

## EXPLANATION - 4 MIN

In Python `not` is a logical operator like `and` and `or`. But `not` only operates on one condition, not two. Reiterate the point that was made during the activity that `not` goes *in front of* the condition that it is operating on. Then show students this truth table with the second column blank and discuss with the class how to complete it. While the table is pretty trivial, this is a good way to underline both the syntax and semantics of `not`.

| • CONDITION | • not CONDITION |
|---|---|
| • True | • False |
| • False | • True |

# Playtime - 30 min

### LOG-IN - 2 MIN

Students should log-in to their accounts as usual.

### PLAYTIME (1) - 10 MIN

Look at Challenge 75 together as a class. This challenge adds a complication to the crocodiles in wells introduced back in the lesson on `while` loops. Now sometimes the crocodile's mouth will be open and must be closed before it is safe to drive the banana over.

The first two lines are already correct for this challenge. Take a couple of minutes to walk students through them so they understand how everything fits together.

```
if not well.crocodile.mouth_closed:
    well.crocodile.toggle()
```

Notice that `crocodile` is a property of the `well` object. But `crocodile` is itself an object with properties of its own. The `mouth_closed` property is `True` if and only if the crocodile's mouth is closed. The `crocodile` object has no `mouth_open` property but remember that the `not` operator reverses a conditional. `not well.crocodile.mouth_closed` essentially means "Is the crocodile in the well's mouth open?" So the whole `if` statement says "If the crocodile in the well's mouth is open, toggle it (which will close it)."

Even though the code for the crocodile part is correct for this challenge, students will still have to fill the well. They can refer back to Challenge 62 if they need a reminder on how to do that.

### PLAYTIME (2) - 18 MIN

Before allowing students to work independently on the remaining challenges in this section, have them start Challenge 76 by testing the following code:

```
for well in wells:
    well.crocodile.toggle()
```

Ask students to explain why this code does not solve the challenge. They should be able to tell you that the problem is calling `toggle()` for the crocodiles whose

mouths are already closed. An `if` statement is needed to only `toggle()` the crocodiles whose mouths are open.

After this quick discussion let students work on Challenges 76 - 78 on their own. If they have trouble getting started on 76 tell them to look back at the `if` statement from Challenge 75.

Challenge 77 might be a little tough. Challenge 64 contains a good model for how to write a loop that lets you refer to each well and its corresponding elephant.

The headline for Challenge 78 is an important clue for students. They should first consider which obstacles need to be removed and then try to write a condition that applies to those obstacles. If they get stuck first remind them about the `not` operator and then get them to list all of the `is_SOMETHING()` methods that apply to obstacles (`is_ice()`, `is_box()`, etc.)

## Debrief - 5 min

Have students go back to Challenge 58 and ask them to write a solution that uses the `not` operator. This is a good way to check their understanding of the concept. The following is the simplest possible solution.

```
for obstacle in obstacles:
    if not obstacle.is_ice():
        dragon.smash(obstacle)
    else:
        dragon.fire_at(obstacle)
```

# Lesson 12 - Functions

## CSTA Standards

| Elementary (3 - 5) | Middle (6 - 8) | High (9 - 12) |
|---|---|---|
| 1B-AP-09<br>1B-AP-10<br>1B-AP-11<br>1B-AP-12<br>1B-AP-15<br>1B-AP-17 | 2-AP-11<br>2-AP-12<br>2-AP-13<br>2-AP-14<br>2-AP-16<br>2-AP-17<br>2-AP-19 | 3A-AP-14<br>3A-AP-15<br>3A-AP-17<br>3A-AP-23<br>3B-AP-12<br>3B-AP-16<br>3B-AP-21 |

## Objectives

- Write functions to solve multiple similar problems without duplicating code
- Complete Challenges 79 - 87

## Components

**PYTHON**

- `def` keyword and function definition

## Introduction – 5 min

**DISCUSSION - 5 MIN**

Have every student think about a skill or activity that is easy for them now but that took them a while to learn. Ask them to write down what the activity is and underneath write down step by step instructions for doing that activity. They don't need to be terribly detailed; 4 - 8 steps is sufficient.

Let as many students as would like to share their activity and steps. After several have shared, ask them if they actually have to think about those individual steps each time they do the activity. Answers may depend on what it is, but most will probably say no, they "just know" how to do it now.

Explain that computers can also "learn" new skills in such a way that we don't have to think about the individual steps each time we use that skill. Of course, computers don't really learn by practice the way humans do; if we want a computer to "learn" a new skill we have to give it step by step instructions. But there is a way to group those instructions together and give them a name so we don't have to repeat each individual instruction each time. That is what today's lesson is about.

# Playtime – 35 min

### LOG-IN - 2 MIN

Students should log-in to their accounts as usual.

### PLAYTIME (1) - 5 MIN

Give students a few minutes to work on Challenge 79. As they get started point out the note about not using a `for` loop. The elephants and wells here are not part of a list so a loop won't work.

Let students work the challenge on their own and then move on when everyone is ready.

### EXPLANATION - 6 MIN

Have students look at the code they wrote for Challenge 79. Ask if they think there is anything wrong with it. If they don't suggest this on their own, ask what they think about having to write three separate `while` loops that all work essentially the same way. What if there were ten wells that needed to be filled, or twenty? Writing a separate loop for everyone would be a pain. And while loops can often be used as a way to avoid duplicating code, sometimes, as in this challenge, they aren't an option.

Explain to students that there is another way to avoid duplicating code: Writing functions. Some functions like `print()` and `range()` are already built into Python to do things like display information on the console and create lists, but we can make our own functions to do anything we want and then use those functions over and over.

There are three main things you need to figure out when creating your own function:

- A name for the function. This can be almost anything, but it should be fairly descriptive of what the function does.
- The parameters (inputs) to the function. Some functions don't take parameters, but most do so they can be used to solve multiple problems.
  - When we use parameters, we allow a function to solve the same problem but with different values.

- The steps the function needs to do. We still have to write code to solve the problem, but putting it in a function lets us write that code just once.

Ask students to keep these elements in mind as they tackle the next challenge.

### PLAYTIME (2) - 10 MIN

Look at the starter code for Challenge 80 with your class. There is something new here - the keyword `def`. `def` is the Python instruction that creates a new function.

The word following `def` is the name of the function. In this case it is `fill_well`, which seems like a sensible name for a function that fills a well.

It is good to work with a standard convention for naming functions. In Python we use the convention that function names are all lower case and if there are multiple words they are separated by underscores.

In parentheses following the function name are the parameters. These are variable names that can be used in the steps below to refer to the inputs to the function. When there is more than one, the parameters are separated by commas. The `fill_well()` function needs to know what well is being filled and what elephant is doing the filling, so those are the parameters here. When the function is actually called, each parameter will be given the value of the corresponding argument.

The `def` line ends with a colon. Below that are the lines that actually make up the function body, the Python instructions that actually solve the problem the function is intended for. As usual, all of the lines of code that are part of the function definition are indented.

Once students have finished studying the function definition have them run their code. The first two elephants should fill their wells, which is great but doesn't quite solve the challenge. Point out that the `fill_well()` function is only called twice (on lines 10 and 11). The students need to add two more calls to `fill_well()` to complete the challenge. Remind them they can mouse over objects in the play area to see their names; this will help match up the right elephant with the right well.

### Playtime (3) - 12 min

Now students need to work to complete Challenges 81 - 87 on their own. Most of these challenges are based on writing and using a `fill_well()` function.

- Challenge 81 Is a lot like 80 except the given function definition for `fill_well()` is not complete. Students will need to complete it and then make sure it Is called with the right arguments for each elephant and well.
- Challenge 82 students to write the `fill_well()` function from scratch; the rest of the code is already correct.
- In Challenge 83 students have to write the definition of `fill_well()` and the function calls to use it. The correct arguments will depend on which elephant they drag to which ledge.
- Challenge 84 introduces crocodiles with open mouths. If students need to review how to close a crocodile's mouth have them look back at Challenge

75. This challenge also puts the elephants and wells into lists so students will need to use list notation to refer to them.

- Challenge 85 is an assessment challenge that uses all of the skills from this lesson.
- Challenges 86 and 87 ask students to apply the concept of functions in a new context. For these challenges student will need to write a function that takes an obstacle as an argument uses `if`/`else` statements to make the dragon destroy it the right way. For both of these challenges all the students have to do is complete the function definition.

# Debrief - 5 min

Ask students why functions are a useful programming tool. Let students' ideas guide the discussion as much as possible, but try to hit each of the following points:

- Functions make it easier to reuse code.
- Functions make it easier to break a problem down into sub-problems.
- Functions make it easier for multiple programmers to collaborate. One programmer can work on a function that does a certain job while another writes code that uses that function.

Regarding the last point, here is a good place to emphasize the importance of paying attention to the readability of programs. Some students may be tempted to choose very short names for their functions and parameters to save a little typing. This is a bad idea because it makes it much harder to read and understand what the program is doing. Coding is often done in a team environment, and it is essential that programs be written to communicate with other people, not just the computer.

# Great Job!

You have completed Banana Tales Part One.

Continue the course with Part Two! You can find the rest of the lesson plans and classroom slides in the Teacher Resources menu on your homepage.