The picture can't be displayed.

CodeMonkey

Write code. Catch bananas. Save the world.

BANANA TALES

PART TWO

Lesson Plans
13-22

Table of Contents

CLICK HERE TO ACCESS THE CORRELATING CLASSROOM SLIDES

Lesson 13 - Classes

CSTA Standards

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

Objectives

- Define Python classes to create on screen objects with specific behavior
- Instantiate custom classes and use them to solve problems
- Complete Challenges 88 - 93

Components

- PYTHON
- *class* keyword
- *__init__* method
- *self* argument
- PLATFORM
- Drill objects

Introduction - 8 min

- DISCUSSION - 6 MIN

- Ask students to imagine that they are responsible for designing a robot that helps out around the house. Lead a brainstorming session for the design of such a robot. At first keep the discussion free form and record (or have a student record) all of the students' ideas.
- After about three minutes, display the column headings below. Explain that "Appearance" has to do with the physical aspects of the robot, "Need to Do" refers to the tasks we want the robot to help with, and "Need to Know" is the information the robot would need to do those tasks. Take a few additional minutes to classify the ideas already generated into these categories and to add any more that the students can think of.

• Appearance	• Need to Do	• Need to Know

- EXPLANATION - 2 MIN
- Tell students that while today's lesson is not actually about creating robots, it is about creating Python objects, and the design process for a virtual object is actually similar to the design of a physical device. To design a Python object, we have to think about what we want it to do. These tasks will become the object's methods. The object will often need to store and use some kind of data to do its job; that data is represented by the object's properties. And while many objects used in computer programming are completely abstract, the objects in Banana Tales are represented by animated characters on the screen, and so that too is part of the design.

Playtime – 32 min

- LOG-IN - 2 MIN
- Students should log-in to their accounts as usual.
- PLAYTIME (1) - 3 MIN
- Have students go to Challenge 88 and take a quick look at the code there. They should note that while the code mentions something called a Drill there doesn't appear to be anything drill-like on screen with the monkey.
- This challenge is just a demo; there is nothing to modify. Let students click Run! and see what happens.
- EXPLANATION - 12 MIN
- There is a lot of new material in this challenge. While students will probably be eager to move on to the next challenges, it is important to spend time on this explanation.

```
class Drill:
    def __init__(self):
        self.x = 9
        self.y = 13
```

```
    def drill_right(self):
        self.x = self.x + 2

drill_1 = Drill()
drill_1.drill_right()
```

- The best place to start understanding this code is at the bottom. `drill_1.drill_right()` on line 10 calls the `drill_right()` method on the `drill_1` object, which moves the drill two units to the right, drilling through any brick walls that are in the way. This should be totally familiar. We've seen object methods used to solve challenges before (e.g. `whale.blow()`). The difference here is that we are getting a peak behind the scenes to see how objects actually work.
- Line 9 is where the drill object is actually created and assigned to the `drill_1` variable. The `Drill()` function is what is doing the creating, but how does Python actually know what a Drill is and how it works?
- That's where the **class definition** on lines 2 - 7 comes in. A class definition is a kind of blueprint for creating objects. These lines don't create a Drill object, they create the blueprint for one.
- A class definition starts with the keyword `class`, followed by the name we want to give to the class, followed by a colon. As usual for Python, the lines that make up the class definition are all indented.
- The class definition is composed of several function definitions. These use the same syntax for creating functions that we saw in lesson 12: `def` keyword, name of function, arguments in parentheses, colon. But these functions will be attached to the object. What are functions attached to an object called? Methods.
- All methods have at least one argument. The first argument to an object method, which is always called `self`, refers to the object itself. This parameter is not listed between the parentheses when the method is called. Instead, Python adds it automatically.
- The `__init__()` method is special. It's a **constructor** function and it is called whenever a new object is created. Here the constructor for the Drill object creates two properties `x` and `y` and assigns them the values 9 and 13. These are the coordinates on the screen where the Drill object will appear. Notice the syntax using the `self` variable: `self.x` means the `x` property of this object, the one being created.
- We don't type out `__init__()` to call the constructor. We call it using the name of the object's class, as we see on line 9. But Remember, the `self` parameter gets added automatically, which is why `Drill()` is called with no parameters.
- `drill_right()` is fairly easy to understand. It moves the drill two spaces to the right by adding 2 to the x coordinate. Again, notice that `self.x` is used to refer to the `x` property of the Drill object that the `drill_right()` method is called on.

- Putting it all together: Lines 1 - 7 are the class definition for a new type of object called `Drill`. The constructor for `Drill` uses the `x` and `y` properties to set the drill's initial position to the coordinates (9, 13). `Drill` has a method called `drill_right()` that moves the drill two spaces to the right. Line 9 actually creates a drill at which point it appears on the screen, and in line 10 the `drill_right()` method moves the drill and in the process destroys the brick wall.
- PLAYTIME (2) - 15 MIN
- The explanation of class definitions will take a little while, but fortunately the rest of the challenges are fairly easy once that explanation is understood. Students should be able to proceed through Challenges 89 and 90 based on the information they've already been given.
- Regroup on Challenge 91 to look at a small example of new syntax. In this challenge the constructor (the `__init__()` function) takes two arguments, `x` and `y`. That makes the class definition for `Drill` more versatile because it can now represent a drill anywhere on the screen. The parameters are passed in through the `Drill()` functions as we see on lines 9 - 11. If necessary, remind students they can hover their mouse pointer over the play area to see the grid and the coordinates of individual squares.
- Similarly, Challenge 92 adds an argument to the `drill_right()` method so the drill can move any number of spaces to the right, not just two. It's good to reemphasize both with `__init__()` and `drill_right()` that the number of arguments you see in the method definition is always one more than the number of parameters that the method is called with since `self` is always first in the argument list.
- Challenge 93 asks students to create four drills using individual variables and then put them in a list and use a loop to move each of them right. The syntax for creating a list is shown on line 15. As long as students name their new drill variables `drill_3` and `drill_4,` they should not need to change that line.

Debrief - 5 min

- It is important to end this lesson with a "confession" that this lesson actually glosses over quite a lot. The class definition of `Drill` used in this lesson is actually incomplete. A real implementation of a drill object would need to address things like what images, animations, and sounds are used to represent the drill as well as making sure that the drill can only move through dirt or bricks, not stone or steel. These details are built into the Banana Tales platform. The software knows to use them for a class called `Drill` even though they are not included in the Python code we type in. The lesson is intended to give students a taste of how creating objects works in Python without going into the full details of how to implement a complex game object.

Lesson 14 - Input

CSTA Standards

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

Objectives

- Use the *input()* function to accept user input as a string
- Use the *int()* function to convert a string into an integer
- Write programs that take different code paths based on user input
- Complete Challenges 94 - 101

Components

- PYTHON

- *input()* function
- *int()* function
- *if*/*elif*/*else* statements

- PLATFORM

- `Pigeon` objects
- Randomized behavior in challenges

Introduction – 8 min

- ACTIVITY - 8 MIN
- For this activity you will need several sheets of colored paper to mark target areas in the classroom. You will also need some small object (toy, book, banana, etc.) to use as a target. Clear enough room for students to move around in the classroom and place the target papers in different locations as far apart as is reasonable. You need at least two target areas but not more than four.
- Each run through of the activity requires three students to fill three different roles. As in a few earlier lessons, one student will be the Programmer and one will be the Walker. The third student is the Randomizer. The Walker turns their back and the Randomizer chooses one of the target papers and places the target object on it. It doesn't really matter whether the object is hidden or hard to see, but the programmer must know where it is.
- Once the object is placed, have the Walker turn back around. Now it is the Programmer's job to direct the Walker to the object using the same kinds of instructions that have been used in the past: "take 5 steps forward", "turn left", etc.
- Repeat the exercise several times with different students in the roles. Then ask students to think about how they might cope with randomness in a Banana Tales challenge. In this exercise the Programmer had the chance to choose their instructions after the target had been placed. What if they had to write the code before they knew the banana's location?
- Listen to students' ideas then share that today we are going to see one possible solution to this problem: Accepting input from the user while the program is running to adapt to different situations.

Playtime – 32 min

- LOG-IN - 2 MIN
- Students should log-in to their accounts as usual.
- PLAYTIME (1) - 15 MIN
- It may not be immediately clear to students what they need to do for challenge 94. Give them a couple of minutes to puzzle over it then go ahead and tell them what to do if they are stuck. They need to change the first line to `snake.Length = 1`. In all of the challenges in this section there is a pigeon trapped by a snake, so the first instruction for every challenge will be `snake.Length = 1`.
- Let students click Run! and see what happens. The pigeon will drop the banana somewhere on the screen. Whether it is to the left or the right of the monkey the car will turn to face the right direction so it can deliver its precious cargo when clicked. That's all that is needed to complete the challenge with three stars, but don't let students move on just yet.

- Have them replay the challenge several times (without changing the code). Ask them if they notice anything interesting. They should realize that the pigeon does not drop the banana in the same place each time. Tell students to click over to level_setup.py. While there is a little bit of unfamiliar syntax here, it should be clear that one of the arguments to the `Pigeon()` constructor is a list of (x, y) pairs. These are the possible locations where the pigeon may drop the banana. When the pigeon is freed, it chooses one of these locations randomly. So, the big challenge in these lessons is writing code that works wherever the banana is dropped.

- Make sure that students understand that where the pigeon drops the banana is really random; there is not some secret pattern they are expected to figure out. It's as if each time it Is freed the pigeon rolls a die (a 12-sided die) and based on the result that's how it chooses the coordinates where it drops the banana.

- Move on to Challenge 95. Explain that this challenge introduces the `input()` function. The `input()` function accepts one argument, a prompt to display. When the `input()` function executes, the prompt is displayed on the console and the program pauses, waiting for the user to type something. When the user presses Enter, the `input()` returns whatever the user typed.

- In this challenge, the prompt asks for a choice between up or down. Then an `if` statement is used to decide what to do based on the input. If the user types "up", then the giraffes' heights are set to 11, filling in the gaps at the top. If they type "down", the heights are set to 4, filling in the lower gap. Which one is right depends on where the pigeon drops the banana.

- Let students fix line 12 and then replay the challenge several times to confirm that the code works wherever the banana lands. Make sure that they understand to type "up" or "down" without any quotation marks when responding to the prompt.

- There is one more important aspect of the code that you need to point out to students. The condition for the `if` statement on line 6 is `answer == 'up'`. Not surprisingly, this tests whether the value of the variable `answer` is equal to `'up'`, but notice the double equals sign. A single equals sign would be wrong here and would not work. The detailed hows and whys of single versus double equals signs aren't important at this point, but students need to understand that when writing conditions that test if two things are equal, they must use the double equals sign.

- Students can work on Challenges 96 and 97 on their own. Challenge 97 does introduce some new syntax, but at this point students should be able to make reasonable educated guesses about how `elif` works based on the example (they don't need to modify that part of the code anyway). For the record though, `elif` lets us test multiple conditions with an `if` statement and do different things based on which one is true. The general format is:

```
if CONDITION1:
    # Do these statement(s) if CONDITION1 is true
    STATEMENT
    STATEMENT
```

```
    # etc.
elif CONDITION2:
    # Do these statement(s) if CONDITION1 is false and CONDITION2 is true
    STATEMENT
    STATEMENT
    # etc.
else:
    # Do these statement(s) if none of the CONDITIONs are true
    STATEMENT
    STATEMENT
    # etc.
```

- PLAYTIME (2) - 15 MIN
- Have students stop when they get to Challenge 98. Tell them to replace the given code with the following:

```
snake.length = 1
giraffe.height = input("Enter giraffe's height: ")
```

- Let them run the code and enter an appropriate height when prompted. This will cause an error. Tell students to read the error message carefully and try to figure out what the problem is.
- The issue is that the `input()` function returns whatever the user types and returns it as text, even if it "looks like" a number. That may seem dumb, but if you think about it there are a lot of times when we use the digits zero through nine but don't really treat them as numbers. For example, if someone is telling you their phone number is 5551234, they say "five five five one two three four", not "five million, five hundred fifty-one thousand, two hundred and thirty-four".
- So even if you type a number, `input()` doesn't know you mean a number. Fortunately, Python offers another function that can change text that looks like a number into an actual number.
- Have students click on the button to the left of Run! to restore the default code. Now the final line says `giraffe.height = int(input("Enter giraffes's height: "))`. The output of the `input()` function is being used as input to the `int()` function, and the output of `int()` is assigned to `giraffe.height`. The function `int()` does just what we want; it takes the text representation of a number we get from `input()` and turns it into an actual number.
- Once the original code is restored the only thing needed to make Challenge 98 work is to set the giraffe's height using the variable `height` on line 7.
- After the discussion of the `int()` function, students can proceed to work on Challenges 99 - 101 on their own.

- Challenge 99 is a straightforward extension of Challenge 98. Students just need to write a loop to set the heights of multiple giraffes.
- In Challenge 100 they need to get two inputs. One is the index of the snake to modify and one is its length.

- Challenge 101 requires two inputs as well. One is the index of the giraffe to modify and one is its height.
    - Make sure that students are replaying each of the challenges several times to ensure their code works wherever the pesky pigeon lands.

Debrief - 5 min

- Ask students if they found any ways to "break" their solutions in this lesson. If nobody has any examples, have them go back to Challenge 95 and bring up their solution. What happens if when the computer asks for input they type something other than "up" or "down"? Students should realize that based on the way the `if` statement is written, any response other than "up" will be treated as down.
- Have them test Challenge 98 as well. What happens when you input something that is not a number? Students need to recognize that `int()` only works on text that looks like a number. `int("pizza")` causes an error because `"pizza"` just isn't a number.
- The moral of the story is handling user input is hard. Users can cause lots of errors because they can type anything they want, whether it makes sense or not. Professional programs that accept user input have to include lots of code to check for and correct those errors.

Lesson 15 - Strings

CSTA Standards

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

Objectives

- Concatenate strings using the + operator
- Write an algorithm involving iteration over a string with a *for* loop
- Complete Challenges 102 - 110

Components

- PYTHON

- *+* as string concatenation
- Iteration over characters in a string using a for loop

- PLATFORM

- Gate objects
- *code* property
- *open()* method

Introduction - 14 min

- EXPLANATION - 10 MIN
- Start by showing students the following line of Python code:

```
word = "banana"
```

- This assigns a kind of value to a variable. Ask students what kind of value is *"banana"*? Some may already know that it is called a **string**. A string is just a sequence of characters. We often think of strings as representing text, but they can contain numbers and symbols as well. Show students the following examples:

```
# Example 1
example_1 = "i am a string"

# Example 2
example_2 = "8675309"

# Example 3
example_3 = "s78!hGD#"

# Example 4
example_4 = "answer()"
```

- In Python, strings must be enclosed in double quotes " or single quotes '. In Banana Tales, most of the time we will use double quotes. The quotes are very important; they tell Python to interpret the string just as a sequence of characters, not as code to be executed. In the examples above, without the quotes Python would try to execute a function called *answer()* and store its output to the variable *example_4*. With the quotes *example_4* is just the eight-character sequence a n s w e r ( ).
- Point out to students that we have been calling strings sequences of characters. What other type of data in Python is considered a sequence? Students should realize that lists are also sequences. Explain that because of that similarity, many of the things you can do with lists you can also do with strings. During Playtime we will see how to use a *for* loop with a string.
- We can use bracket notation to refer to individual characters of strings. If *word = "banana"* then *word[0]* is *"b"*, *word[1]* is *"a"*, *word[2]* is *"n"*, etc. Though only a single character long, each of these is a string itself. Because they are part of a bigger string, we refer to them as **substrings** of the string *"banana"*.
- PRACTICE - 4 MIN
- Show the students this Python code, and then have them try to evaluate each of the expressions below.

```
mystring = "baby monkey"
```

1. *mystring[0]* (*"b"*)
2. *mystring[3]* (*"y"*)
3. *mystring[4]* (*" "* - space character)

Playtime – 24 min

- LOG-IN - 2 MIN
- Students should log-in to their accounts as usual.
- PLAYTIME (1) - 10 MIN
- Challenge 102 is a “Click Run! to solve” challenge. Let students see what happens and then discuss the new game elements demonstrated here.
- Now we have gates that need to be opened using the right password. The password is a string that must be used as the parameter to the *`open()`* method of the gate.
- Gates also have a *`code`* property. *`code`* is a string. For this challenge the *`code`* is the same as the password, but in general it is more like a clue to the password. Students will have to use the already opened gates to figure out how *`code`* relates to the password. In the play area, the code for each gate is displayed on top of the screen above it. Once the gate is open, the password is displayed at the bottom of the screen.
- Move on to Challenge 103. In this challenge the password is not the same as *`code`*. Instead, the password is the code with the additional string “abc” appended to it. Explain to students that sticking strings together like this is called concatenation. In Python, we use the plus sign to concatenate strings. The given code demonstrates how this works, but concatenates “aaa” instead of “abc”. Let students fix this and then move on to the next challenge.
- Students should try Challenge 104 on their own. This one involves two concatenations, one in front of the *`code`* value and one behind it. To three star this challenge, students will have to write a loop, but they may be thrown off by the fact that some of the gates are already open. You can let them know that calling the *`open()`* method on a gate that is open already won’t cause an error or lower their star rating.
- PLAYTIME (2) - 12 MIN
- Have students stop at Challenge 105 so you can look at it as a class. First ask students to look at the codes and passwords for the already open gates and describe what the pattern is. In this case, each letter of the code is doubled to make the password. For example, “boat” becomes “bbooaatt”.
- To solve this challenge, we can’t just concatenate strings to the beginning or end of *`code`*. We have to build the password from scratch. The given code demonstrates how to do this.
- Line 1, *`password = ""`*, stores an **empty string** to the variable *`password`*. An empty string is exactly what it sounds like, a string with no characters. That may sound useless, but it is actually the perfect starting point for building a string piece by piece.

- The `for` loop on lines 3 - 5 will execute once for each character in the string `code`, with `c` equal to that character. Inside the loop, `c` is concatenated to the password-in-progress twice, which is exactly what we want. If it isn't clear how this works (or even if it is), let students run the code. The `print()` statement will show the steps of assembling the password.
- To complete the challenge, students will need to repeat the process for `gates[4]` as well. To earn three stars, they will need to use a loop. It's good to let them know that they can (and should) leave the `print()` statement, it will not prevent them from earning three stars.
- At this point students have all the information they need to complete Challenges 105 - 110 on their own.

- For Challenge 106 which requires reversing the code string, students should pay close attention to the order of the concatenation in the sample code they are given.
- Challenge 107 requires building the password from the second letter. Students should understand the code given as a sample to open one gate.
- are with different lengths.
- Challenge 109 requires building the password with the code's last letter as the password's first letter. Students should understand the code given as a sample to open one gate.
- Challenge 110 is an assessment challenge. It is similar to Challenge 106, except with multiple gates in a loop.

Debrief - 7 min

- Finish by having students go to any challenge from this lesson. Tell them to just delete whatever code is already there. We're not going to try and solve the Challenge, we're just testing some Python code.
- Have them type the following and predict what will happen when it runs:

```
word = "house"
print(word[0] + "ome")
```

- Ask a student or students to describe what the expression `word[0] + "ome"` means. They should be able to explain that `word[0]` picks out the first character in `"house"` and the `+` concatenates it with `"ome"` to produce the string `"home"`.
- Now get students to try this code. Again, before they run it, they should predict what it might do:

```
word = "house"
word[0] = "m"
print(word)
```

- Students probably very reasonably predicted that this would print the string `"mouse"` to the console, but it causes an error instead. Why? Strings in Python are **immutable** objects. That means they can't be changed once they are created. It may have seemed like strings were being changed

during the challenges, but an expression like `password = c + password` actually creates a brand-new string and makes the variable `password` point to it. We will encounter another type of immutable object a little bit later in the course.

Lesson 16 - Dictionaries

CSTA Standards

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

Objectives

- Define dictionary
- Find the value in a dictionary associated with a given key
- Add a given key/value pair to a dictionary
- Check if a given key is in a dictionary
- Complete Challenges 111 - 117

Components

- PYTHON
  - Dictionaries
  - *in* and *not in* operators
- PLATFORM
  - Rat object
  - *get_passwords()* method

Introduction - 10 min

- ACTIVITY - 5 MIN
- Give a student a dictionary and ask them to look up the definition of the word “ramify” as quickly as they can. Repeat the exercise several times with different words. “squalid”, “demur”, and “ignominious” are all good possibilities. Insist that students look up the words and report the dictionary definition, even if they already know the meaning themselves. If you have enough dictionaries and your class handles competition well, you can ask students to race to find the definitions.
- Now ask a student (or students) to use the dictionary to find a word that means “to make known by open declaration”. The word in question is “promulgate”, but finding this in the dictionary will likely be a difficult task. Again, make sure that that students use the dictionary to find the word rather than simply guessing (or knowing, if they have exceptional vocabularies). Allow enough time for the student(s) to make a genuine effort to find the word but call time after about two minutes and move on to the discussion questions.
- This activity will be best with a real physical dictionary rather than an online version. If you don’t have physical dictionary, the first part of the activity will still work fairly well. For the second part, have students work with a dedicated dictionary website (like dictionary.com) rather than using a general-purpose search engine. That way they will not be able to search for the definition directly but will be forced to browse or use some form of guess and check strategy.
- There is nothing special about the words and definitions given. You can choose any words that students are likely not to already know the definitions of. For the second task, it would be best to give students the exact definition of “promulgate” (or whatever word you use) found in your dictionary rather than to use our wording.
- DISCUSSION - 2 MIN

1. How difficult was it to find the definition when you were given the word?
2. How does the dictionary make it easier to find the definition of words quickly?
3. If the dictionary had twice as many words in it, how would that affect how long it takes to look up a word?
4. Was finding the word if you know the definition easier or harder than the other way around? Why?

- EXPLANATION - 3 MIN
- The physical dictionary used in the activity is a model for a Python data structure called a dictionary. Some other programming languages call a dictionary an associative array. A dictionary is a collection of pairs of data called keys and values. The key is a “simple” Python type – usually a string, sometimes an integer, very occasionally one of the other data types we will encounter later in our studies. The value can be any Python data type: a string, an integer, a list, an object, or anything else. In these challenges we keep things simple and the values are strings.

- The main idea with a dictionary is that if you know the key, it is easy and fast to find the corresponding value. That corresponds with the procedure of looking up the definition of a known word in the dictionary. On the other hand, if you know a value for a dictionary, it is not particularly easy to find the corresponding key; generally the best you can do is look at every pair in the dictionary until you find the one with the value that was given. This corresponds with trying to figure out the word that goes with a given definition. Dictionaries are very much a one-way data structure. They make it easy to find a value if you know the key, but not the other way around.
- In fact, while keys in dictionaries must be unique, the same is not true for values. Different keys can be associated with the same value, so it wouldn't even necessarily make sense to ask what key a certain value goes with. The answer could be multiple keys.
- There is a subtle point that is worth making about the difference between Python dictionaries and real-life dictionaries. In a real-life dictionary, even with the help of alphabetical order it takes time to search for a word, and the longer the dictionary the longer the search takes. On the other hand, a Python dictionary with a million key/value pairs will take essentially no longer to search for a given key than one with only ten pairs. For the small dictionaries used in these challenges that isn't very important, but in the real world the speed of accessing values even in large dictionaries is one of the things that makes them most useful.

Playtime - 30 min

- LOG-IN - 2 MIN
- Students should log-in to their accounts as usual.
- PLAYTIME (1) - 5 MIN
- All students should complete Challenges 111 and 112 with at least two stars.
- For Challenge 111, make sure that students notice the syntax for referring to a value in a dictionary as demonstrated on line 3. In general, the syntax is:

```
# This represents the value that corresponds with key
dictionary_variable[key]
```

- The use of square brackets is similar to the syntax for referencing elements of a list, but students should recognize that here gate_1.code is a string being used as a key instead of an integer index.
- For Challenge 112, ask students to look at what line 2 prints out as the representation of the *codes_and_passwords* dictionary. The main elements of the syntax they should notice are the curly braces, the keys and values separated by a colon, and the comma between key : value pairs.
- PLAYTIME (2) - 23 MIN

- All students should continue and complete Challenges 113 through 117 with at least two stars.
- Starting in Challenge 114, students will need to write code that recognizes when the *codes_and_passwords* dictionary returned by *get_passwords()* doesn't contain a certain key. That's what the *in* operator is for. The expressions

```
key in dictionary_variable
```

- equals *True* if *dictionary_variable* contains a pair with the given key, *False* otherwise.
- We can also do:

```
key not in dictionary_variable
```

- which is the exact opposite; it equals *True* if *dictionary_variable* does not contain a pair with the key and *False* otherwise.
- Make sure that students understand that *not in* is special syntax; you can't just apply *not* to any old operator in the middle of an expression. For example:

```
3 not < 1
```

- is wrong and will cause an error.

Debrief - 5 min

- Display the following line of Python code:

```
favorite_foods = {"monkey" : "banana", "giraffe" : "leaves", "crocodile" : "monkey"}
```

- Then ask students to write Python expressions and statements using the *favorite_foods* dictionary.

1. Write an expression that has the value "banana".
2. Write an expression that has the value "monkey".
3. Write a statement that changes the value that goes with key "giraffe" to "fruit".
4. Write a statement that adds the value "pizza" to the dictionary with a key of "programmer".

- Answers

1. `favorite_foods["monkey"]`
2. `favorite_foods["crocodile"]`
3. `favorite_foods["giraffe"] = "fruit"`
4. `favorite_foods["programmer"] = "pizza"`

- Students may struggle briefly with using strings directly as keys. If they have difficulty, give them the answer to number 1 as a model.

Lesson 17 - Sets

CSTA Standards

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

Objectives

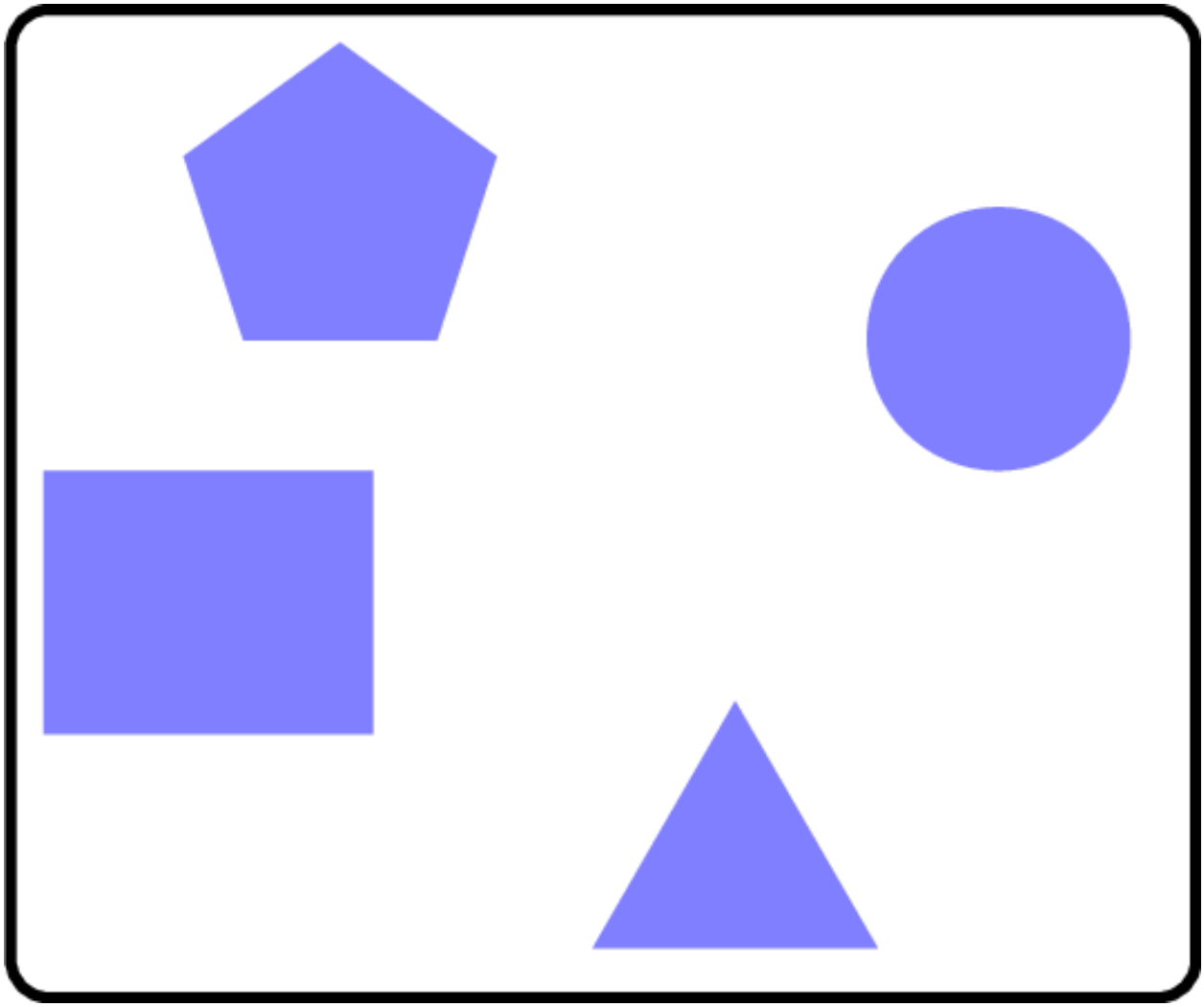
- Understand that sets cannot contain duplicate members
- Manipulate sets using the `add()`, `pop()`, and `remove()` methods
- Complete Challenges 118 - 123

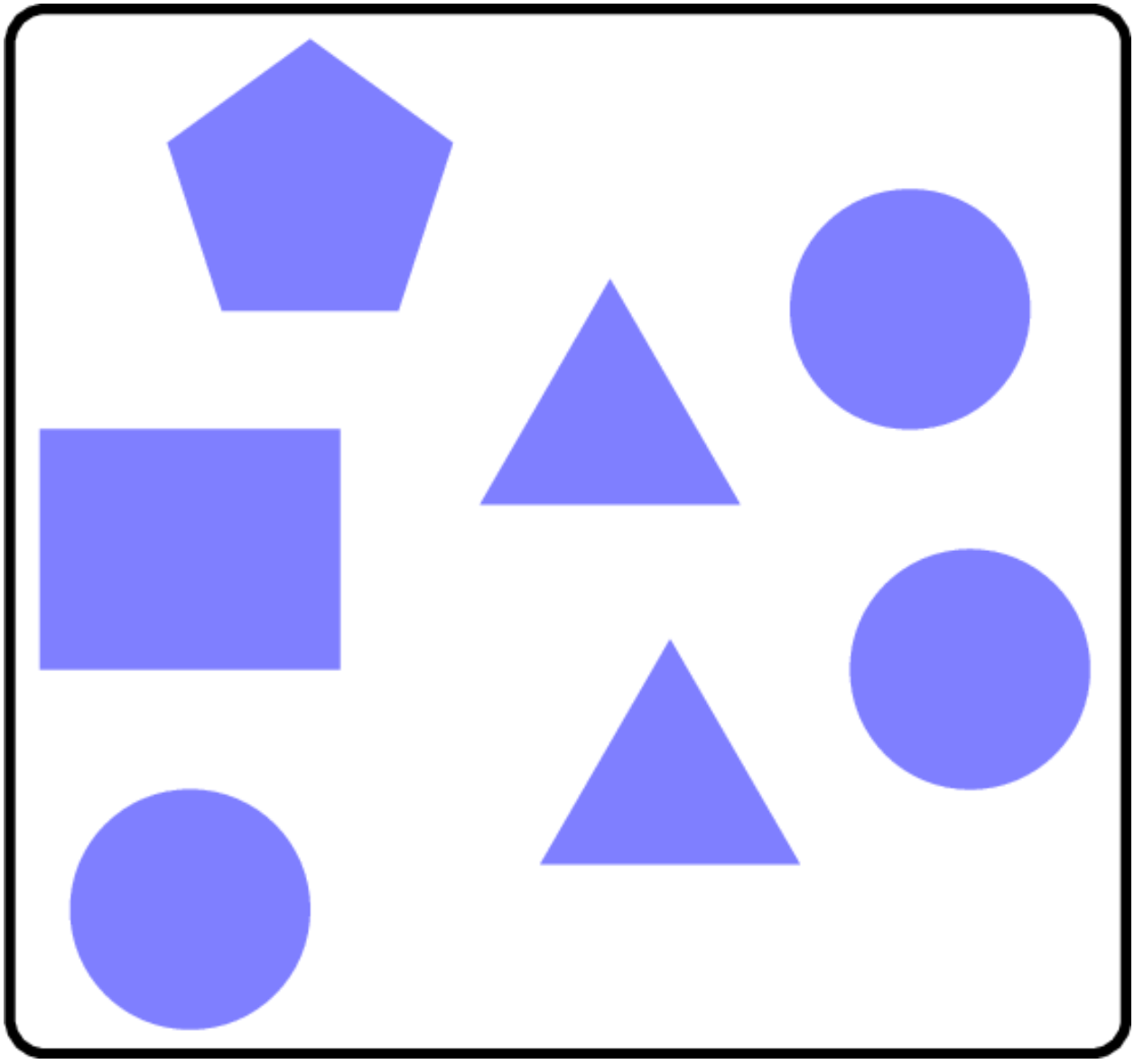
Components

- PYTHON
- Sets
- PLATFORM
- Basket objects
- `balloon_colors` property

Introduction - 7 min

- Show the following image and ask a student to list what kinds of shapes they see inside the black boundary.

- 
- Once a student has listed the four shapes, ask if there is another way to say the answer. The question is a little vague, so students may try to use synonyms or creative rephrasings, but the idea you are trying to get at is that the order does not matter; it is equally correct to say “pentagon, circle, rectangle, triangle” as it is to say “rectangle, triangle, circle, pentagon”.
- Now show this image and again ask a student to list what kinds of shapes they see inside the boundary.

- 
- Again, ask for alternate answers. If some students repeat the names of shapes, ask if that is actually necessary. The question was to list the *kinds* of shapes in the picture, not to list all of the shapes, and there are still only four kinds: triangle, rectangle, circle, pentagon.
- Explain that these two quick examples are meant to introduce a concept called **sets**. A set is a collection of items in which the order doesn't matter and where duplicates are not allowed. The metaphor here is a little tricky: The second shape image *is not* a set because there are duplicates, but our description of what *kinds* of shapes are there *is* a set.
- In this lesson we are going to study Python's version of sets, which is based around those two main principles: no duplicates and order does not matter.

Playtime – 32 min

- LOG-IN - 2 MIN
- Students should log-in to their accounts as usual.
- PLAYTIME - 30 MIN
- Let students work through Challenge 118 on their own. The comments explain what to do so it should only take a minute or two. When they are finished, ask them how the repeated uses of `basket.balloon_colors.add("blue")` on lines 1, 4, and 5 illustrate that the `balloon_colors` property of basket is a set. If they have trouble, have them run the code again and see how many blue balloons are added to the basket. Because `balloon_colors` is a set, it doesn't allow duplicates. The `add()` method of a Python set object is forgiving; if you try to add something that is already in the set, it just does nothing rather than cause an error.
- Before students leave Challenge 118, have them try as many different color names as they can think of on line 10. They should discover that Banana Tales knows a lot of the most common color names, but if they get too fancy and try a "cerulean" or "vermilion" they will get an error.
- Students should be able to handle Challenge 119 on their own. Have them pause on Challenge 120. This challenge introduces the `pop()` method. It is important for students to understand that `pop()` has nothing to do with balloons. `pop()` is a method that every set object has that removes a *random* item from the set and returns it as output. Remember, sets are unordered, so there is no such thing as the first item or last item in a set.
- Once students understand how `pop()` works, they can continue to work on Challenge 120 up through Challenge 123. Using the `add()` and `pop()` methods to manipulate sets is straightforward, but for some of the challenges it can be a little tricky to see exactly what balloons need to be added or removed.

- Make sure students notice the draggable baskets for Challenge 120. Remind them also that they can mouse over the baskets to determine which one is which. In this case `baskets[0]` is actually the rightmost of the baskets.
- Challenges 121 and 122 involve doing the same thing to each basket on the screen, either adding balloons (Challenge 121) or removing balloons. In Challenge 121 students need to make sure to add two colors that aren't already in any of the baskets.
- In Challenge 123, the solution is to "move" exactly four balloons from the top basket to the bottom. Remind students that the `pop()` method returns the item that it removes. The output of `pop()` can be used as the input of `add()`.

Debrief - 6 min

- Review with students the two key properties of sets:

- Sets are unordered collections.
- Sets do not allow duplicate members.

- One important operation on sets that is not covered in these challenges is checking if an item is in a set. Have students go to Challenge 121 and replace the given code with the following:

```
for basket in baskets:
    if "red" in basket.balloon_colors:
        print("Red balloon")
    else:
        print("No red balloon")
```

- This demonstrates that the *in* operator can be used to check if an item is a member of a set the same way it can be used to check if a particular key is in a dictionary.

Lesson 18 - Tuples

CSTA Standards

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

Objectives

- Compare and contrast tuples and lists
- Solve complex challenges involving making sets contain specific numbers of items
- Complete Challenges 124 - 128

Components

- PYTHON
- Tuples
- Iteration over tuples using a *for* loop

Introduction - 5 min

- REVIEW - 5 MIN
- Choose three or four students and have them stand. Ask the other students to think about Python lists. How would they write a Python list of the standing students' first names in alphabetical order? Let the students

actually write down their answers and check each other. The correct syntax is something like this:

```
["Avery", "Charlie", "Spencer", "Taylor"]
```

- The names will vary of course, and it is not important (from a Python perspective) whether they are capitalized or not. The important elements of the syntax are the square brackets, the commas between elements, and the names in quotes. If some students omitted the quotes, remind them that for *Avery* to make sense (without quotes) there would have to be a variable named *Avery* defined in the program.

Playtime – 32 min

- LOG-IN - 2 MIN
- Students should log-in to their accounts as usual.
- PLAYTIME (1) - 10 MIN
- Give students a few minutes to solve Challenge 124. Make sure they see the baskets available in the drag and drop area. After they have solved it, call their attention to line 1. The value assigned there looks *almost* like a list, and in line 3 it is used as the basis for a *for* loop just like a list is. But it is not quite the same syntax as a list. How is it different?
- Students should observe that the elements of the value created on line 1 are placed inside parentheses, whereas when creating a list, the elements are inside square brackets. Explain that this is an example of a Python data structure called a **tuple**. Aside from using parentheses instead of square brackets, tuples are very much like lists, with one other important difference.
- Have the students delete all of the code for Challenge 124 except for line 1. Once again, they are going to use the Banana Tales platform to explore a topic in more depth for a few moments without trying to solve the challenge. First, have the students type in this code and predict what it will do before they run it:

```
new_colors = ("blue", "yellow", "brown", "red", "white", "green", "gray")
print(new_colors[3])
```

- Of course this prints the string “red” on the console. Now have them try this:

```
new_colors = ("blue", "yellow", "brown", "red", "white", "green", "gray")
new_colors[3] = "pink"
print(new_colors)
```

- This one may be a bit surprising. Line 2 of this code produces an error. Ask students if they can remember having encountered a similar error before. Back in lesson 16 we learned that in Python strings cannot be changed once they are created. Students should recall that the term for this

property is immutable. This example illustrates that tuples are another immutable data type in Python. Tuples are immutable and lists are not. That is the most important difference between the two data types.

- PLAYTIME (2) - 20 MIN
- Because tuples work so much like lists, students are in a good position to use their existing knowledge to tackle Challenges 124 - 128 at their own pace. Students will need to keep in mind that while each of these challenges uses a tuple, the `balloon_colors` property of the basket object is still a set. Challenge 127 introduces the `remove()` method of set objects. This is a good place to remind students that `pop()` removes a random item from a set. The `remove()` method lets us remove a specific element but causes an error if that element is not in the set.
- Encourage students to study and think carefully about the algorithm in the code given for Challenge 127. A `for` loop is used to visit each color in the `balloon_colors` property of a basket. If that color is not in the `basic_colors` tuple, that balloon is removed from the basket. To solve this challenge students will need to write similar code for the other basket.
- Clever students may notice a seeming contradiction here. In the last lesson the point was made repeatedly that sets are unordered but using the `balloon_colors` set here for a `for` loop implies that the elements are in some order. The fact is that internally Python does have a rule for putting them in order, but it is not something that programmers have any control over. So, a `for` loop can be used with a set, but the order in which the elements will be visited is not predictable.
- Challenge 128 is harder. The key here is that the two baskets (`basket_1`, `basket_2`) need to come down while the other two baskets (`basket_3`, `basket_4`) need to go up. The algorithm for raising baskets based on the `colors` tuple, and the algorithm for lowering them is given in the sample code.

Debrief - 8 MIN

- Lists, dictionaries, sets, and tuples are all examples of **collection objects** in Python. Take a few moments and review the major properties of each with your students.

Lists

- Ordered collection of elements
- Represented with square brackets, e.g. `["red", "blue", "green"]`

Dictionaries

- Stores data as key / value pairs
- Finding the value given the key is easy and fast; the reverse is much slower
- Duplicate keys are not allowed
- Represented using curly braces and colons to separate keys and values, e.g. `{"apple" : "red", "blueberry" : "blue", "pear" : "green"}`

Sets

- Unordered collection of elements
- Duplicate elements are not allowed
- Represented with curly braces, e.g. `{"red", "blue", "green"}`

Tuples

- Ordered collection of elements
- Immutable
- Represented with parentheses, e.g. `("red", "blue", "green")`

    - Students will probably be curious as to why tuples are useful, given that they work pretty much like lists. The answer is that because they are immutable, tuples are more efficient for the computer to use. For very short programs like the ones we write in Banana Tales it doesn't really matter, but for bigger programs using the faster data structure can be a big win.

Lesson 19 - 2D Lists 1

CSTA Standards

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

Objectives

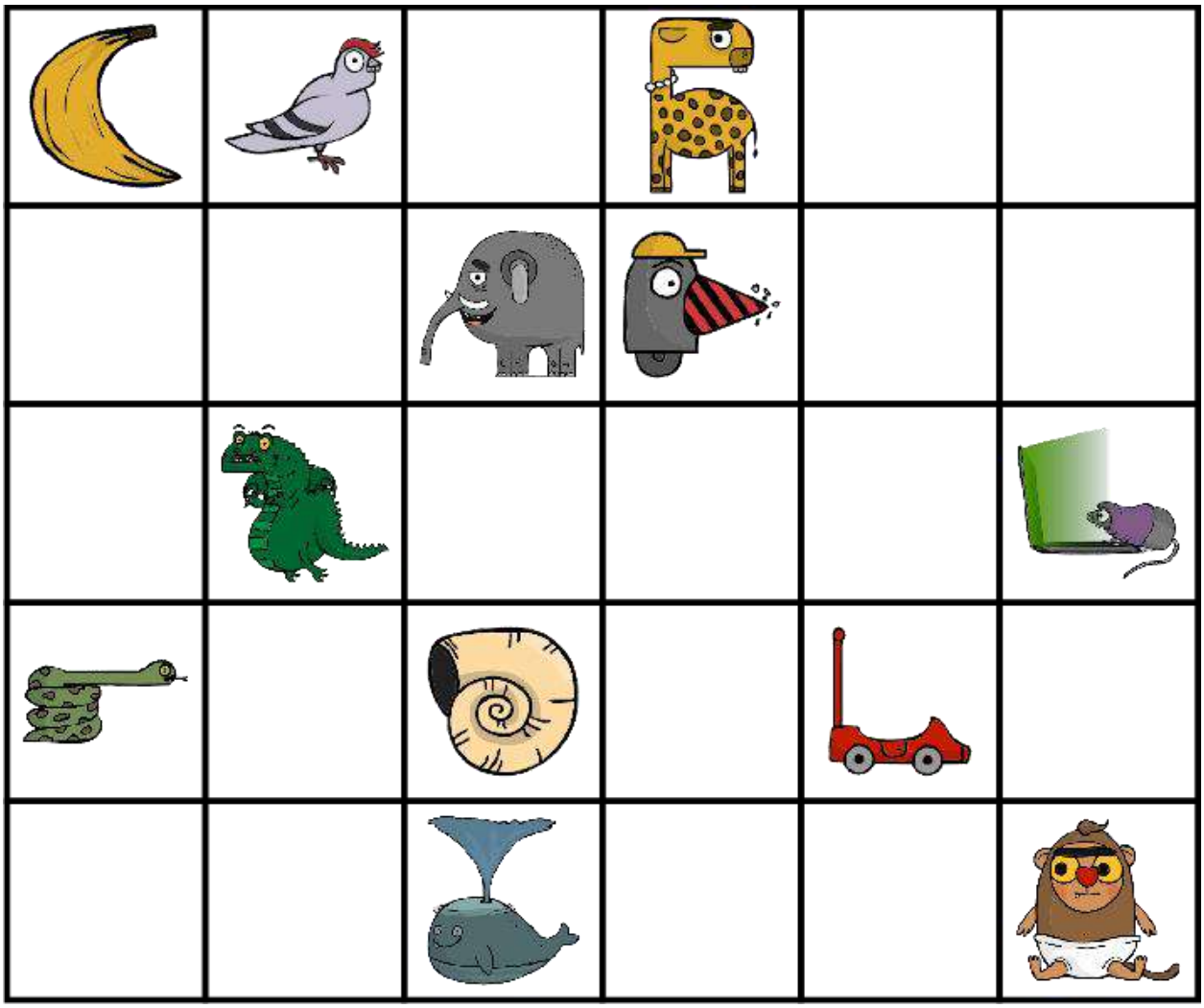
- Refer to and modify specific elements in a 2D list by using their coordinates
- Assign the same value to multiple variables in a single statement
- Iterate over a single row of a 2D list
- Complete Challenges 129 - 135

Components

- PYTHON
  - Notation for 2D lists
  - *None* value
  - Chained assignment
- PLATFORM
  - Cloud objects

Introduction - 6 min

- DISCUSSION - 6 MIN
- Show the image below:



- Explain that this is a model of a **2D list**. A 2D list has a number of elements grouped into horizontal rows and vertical columns. By convention, when we describe a location within the list, we say the row first. We will also follow the computer science convention of starting the numbering with 0, going top to bottom and left to right. For example, in this list, the banana is located at row 0 column 0 and the drill is located at row 1 column 3.
- Ask students the following questions so they can practice the conventions for 2d lists. Make sure that in each case where a location is asked for, they say the row number first.

1. Where is the pigeon located? (row 0 column 1)
2. Where is the snake located? (row 3 column 0)
3. What is at row 3 column 2? (shell)
4. What is at row 2 column 5? (rat)
5. Where is the monkey located? (row 4 column 5)
6. Where is the dragon located? (row 2 column 1)
7. What is at row 1 column 2? (elephant)

Playtime – 27 min

- LOG-IN - 2 MIN
- Students should log-in to their accounts as usual.
- Playtime - 25 min
- Challenge 129 is ready to go as given. Let the students run it, then pause for a brief discussion.
- Point out that in this challenge (and all the challenges in this lesson), `clouds` is a 2D list that follows the same conventions that were discussed in the introduction. The first index in brackets following the variable name is the row number, the second index is the column number. In Banana Tales, the `clouds` lists represent a rectangular grid of clouds. If the list has a `Cloud` object at a particular position, then there is a cloud block in the corresponding position in the grid.
- Let students continue and do Challenge 130 which asks them to fill in several locations in the cloud bank.
- Pause at Challenge 131. This challenge introduces the `None` constant. `None` is a special keyword in Python that is used to represent a missing value. Make sure students understand that it is not the same as 0. Also make sure they note that it is spelled with a capital letter: `None`, not `none`.
- In Banana Tales, we use `None` to represent a spot in the `clouds` grid that is empty. We can remove a cloud from the grid by assigning `None` to the appropriate row and column of `clouds`.
- After discussing `None` students can move on to complete the challenges through 135 on their own.

- Challenge 133 introduces a little new syntax but it is easy and fairly self explanatory. `clouds[1][2] = clouds[1][4] = clouds[1][5] = None` assigns the same value (`None`) to each of the locations `clouds[1][2]`, `clouds[1][4]`, and `clouds[1][5]`. This is called **chained assignment**. It can be a way to save lines of code (and is needed to three star this challenge), but it should be used sparingly.
- Students won't be able to three-star Challenge 135 if they write separate loops for adding and deleting clouds. They will need to use a single loop that has two statements, one to add the cloud on one row and one to remove the cloud on the other row.

Debrief - 10 min

- Explain to students that while 2D lists were introduced in this lesson as a new type of Python value, they really aren't. 2D lists are just lists of lists. That is, they are lists in which each element is another list.

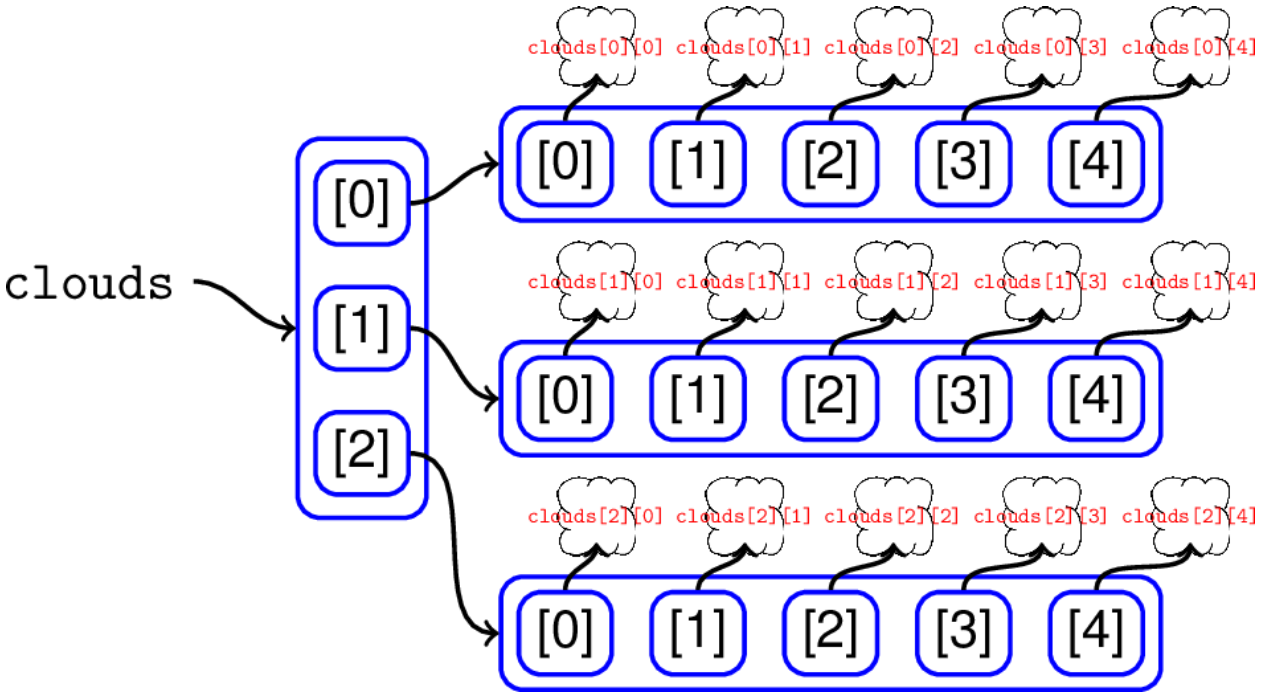
- Have students go back to Challenge 133 and look at the level_setup.py tab. There they will see the assignment statement that creates the *clouds* variable. Here it is with better formatting:

```
clouds = [
    [Cloud(),Cloud(),Cloud(),Cloud(),Cloud(),Cloud()],
    [None,None,Cloud(),None,Cloud(),Cloud()],
    [Cloud(),None,None,Cloud(),None,Cloud()],
    [Cloud(),Cloud(),Cloud(),Cloud(),Cloud(),Cloud()]
    ]
```

- The outermost set of square brackets creates a list. Inside these, there are four other lists that serve as the elements of the outer list. It is the elements of these inner lists that actually make up the cloud grid.
- Remind students that variables are really just pointers to spaces in the computer's memory that hold different types of values.
- Let's look at another example:

```
clouds = [
    [Cloud(),Cloud(),Cloud(),Cloud(),Cloud()],
    [Cloud(),Cloud(),Cloud(),Cloud(),Cloud()],
    [Cloud(),Cloud(),Cloud(),Cloud(),Cloud()],
    ]
```

- The diagram below shows what's going on behind the scenes with a 2D list.



- *clouds* points to a list somewhere in the memory. This list has three elements, each of which points to another list somewhere else. Each of these lists has five elements that point to some other kind of value, *Cloud* objects in this example.

- The way we follow the pointers in the diagram corresponds with the way indexing works for a 2D list. The first index specifies which row, the second specifies which item in that row.

Lesson 20 - 2D Lists 2

CSTA Standards

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

Objectives

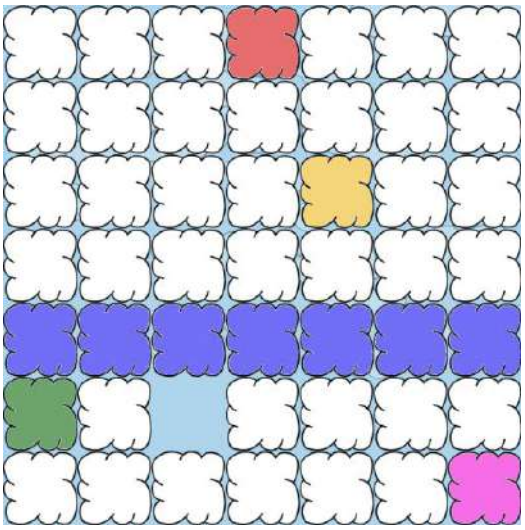
- Iterate over a single column in a 2D list
- Develop algorithms to iterate over 2D lists in more complex ways involving multiple rows and columns
- Complete Challenges 136 - 141

Components

- No new components

Introduction - 8 min

- REVIEW - 8 MIN

- Display the image below for students:
- 
- Do a quick review of the concepts from the previous lesson by asking students the questions below. They should assume that as in the challenges `clouds` refers to the 2D list represented by this cloud grid.

1. What Python expression would refer to the red cloud? (`clouds[0][3]`)
2. What Python expression would refer to the yellow cloud? (`clouds[2][4]`)
3. What Python expression would refer to the pink cloud? (`clouds[6][6]`)
4. What Python statement could be used to remove the green cloud? (`clouds[5][0] = None`)
5. What Python statement could be used to place a cloud in the empty space? (`clouds[5][2] = Cloud()`)
6. Write a loop to remove every cloud in the row of blue clouds.

```
# range(len(clouds[0])) or even range(7) would work below as well
for i in range(len(clouds[4])):
    clouds[4][i] = None
```

Playtime – 32 min

- LOG-IN - 2 MIN
- Students should log-in to their accounts as usual.
- PLAYTIME – 30 MIN
- This lesson is essentially a continuation of the last so students should spend the first part of Playtime working through them independently.

- Challenge 137 involves looping over a column instead of a row. That means that the index variable should be used for the first index of `clouds`, not the second.
- Challenges 137 and 138 require the students to remove a row *and* a column. Because the cloud grids are square this can be done with a single loop.

- In Challenge 139 students have to remove all of the clouds in the grid. The given code illustrates how to write a nested loop that covers all the possible row and column index combinations.
- Challenge 140 and 141 both ask students to create clouds in a stair step pattern. In Challenge 140 this is relatively easy because the row and column indices of the clouds that need to be created are the same. Challenge 141 is somewhat trickier. If students are having trouble ask them to list (row, column) pairs for the spaces that need to be turned into clouds and look for a pattern. Both of these challenges only need a single loop, not a nested loop.

Extension

- Students will probably finish Challenges 136 - 141 with time to spare. When they finish, they should work on the following activity:
- Provide each student with a small piece of graph paper. It only needs to be ten squares by ten squares so you can get several from a single sheet. Students should use the 10x10 grid to make a design by shading in squares. The small scale doesn't give them a lot of space to stretch their artistic wings, but it is plenty of space for a student's initials or an emoji-like icon or something similar. Once they have completed their design, they should go to either Challenge 140 or 141 and "draw" it there by creating clouds to represent the shaded spaces. This will probably involve quite a few lines of code but encourage them to use loops and/or chained assignment statements to make the work easier.

Debrief – 5 min

- Close with a few review questions to check for understanding.

1. Assume `clouds` is a 2D list as in the challenges. What Python expression gives the number of rows of `clouds`? (`len(clouds)`)
2. What expression gives the number of columns of clouds? Hint: This is the same as the number of elements in a single row. (`len(clouds[0])`)
3. Look at Challenge 137. How would this challenge be more difficult if the cloud grid was not square? (You would have to write two loops because you couldn't use the same index variable for rows and for columns.)

Lesson 21 - Bubble Sort 1

CSTA Standards

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

Objectives

- Swap the value of two variables
- Apply the swap procedure multiple times to put a list in order
- Write a function to put two list elements in order by swapping them if necessary
- Complete Challenges 142 - 146

Components

- PYTHON
- Temporary variable
- PLATFORM
- Flamingo objects

Introduction - 10 min

- ACTIVITY - 10 MIN
- This activity requires two hand-sized objects that are similar but not identical and a largish piece of paper for each student. In the following description we will assume the objects are different colored plastic cups, but any objects of similar size will work.
- Have students draw three circles on the paper that are just big enough to hold the cups. The circles should be drawn on the bottom left, bottom right, and top middle of the paper. Once the circles are drawn, they should place the two cups inside the two bottom circles.
- Explain that the object of the activity is to switch the two cups. Easy enough, except there are a few rules:

- You can only use one hand
- You can only hold one cup at a time
- If a cup is not in your hand, it must be in one of the circles
- One cup per circle (no stacking cups)

- Even with these rules, this is not a terribly hard challenge. Most students will probably figure it out fairly quickly. Once a student has switched the cups once, have them do it again, this time paying close attention to the steps they are doing. After the second switch, ask them to write down their steps as a procedure that someone else could follow to accomplish this task.
- After everyone has had a chance to figure out and write down the procedure, let students share their ideas. In order to have a common language, suggest that the basic operation for this procedure is “Move”, e.g. “Move the cup in the left circle to the right circle”. Given that, students should be able to converge on a procedure like this:

1. Move the cup in the left circle to the middle circle
2. Move the cup in the right circle to the left circle
3. Move the cup in the middle circle to the right circle

- Tell students to keep this procedure in mind as they begin to work through the challenges in this lesson.
- (If your resources are limited, it is possible to do this activity as a demonstration. Let one student do the switching challenge and then the whole class can discuss the procedure they used.)

Playtime – 27 min

- LOG-IN - 2 MIN
- Students should log-in to their accounts as usual.
- PLAYTIME - 25 MIN

- Challenge 142 introduces flamingos. Let students run the code for Challenge 142 (it is already correct). They should note that the taller flamingos on the right (`flamingos[2]` and `flamingos[3]`) replace the shorter flamingos on the left (`flamingos[0]` and `flamingos[1]`), but that the flamingos *do not swap*.

- Before leaving Challenge 142, have students erase the code that is there and replace it with the following:

```
for flamingo in flamingos:
    print(flamingo.height)
flamingos[0].height = 8
```

- When run, this will display the height of all four flamingos and then cause an error. This is to demonstrate that flamingos do have a `height` property but that it cannot be changed (unlike the `height` property of giraffes).

- Challenge 143 is a one-liner. Let students figure it out on their own and then discuss. Ask them how the procedure used here is similar to what they did in the introductory activity. The three main points they should understand are:

- Both are three step procedures
- The assignment operation in Python corresponds to the move operation in the activity
- The variable `temp_flamingo` corresponds with the middle circle in the activity

- Explain that this three-step procedure for swapping two variables will be used over and over in this lesson and the next, so it is important that they understand how it works.

- After this discussion, let students complete Challenges 144 - 146 on their own.

- Students can complete Challenge 144 by following the advice given in the comments. Based on the procedure previously learned, doing these two swaps would seem to require six (3 + 3) lines of code. But in order to three star this challenge students will need to find a way to do it in four. If they have the six-line solution they should notice that some of the same variables are assigned to twice. If they keep only the second assignment the procedure will work and meet the line requirement.
- Challenge 145 is another two-level challenge. Students will need to make sure that their code works for both levels. If they use the model of the first two `if` statements filling in the third should be easy.
- There is only one line to complete on Challenge 146, but make sure students notice the arguments of the swap function: first the list that contains the objects to be swapped, then the index of the first item, and then the index of the second item. A swap function like this will be a key part of the rest of the challenges in the course.

Debrief - 8 min

- Ask students to compare the solutions to Challenges 144 and 145. Both involving reordering three flamingos, but in Challenge 145 it takes nine lines of code to complete the task while in Challenge 144 we can do it in four. Why is that?
- Part of the reason is that Challenge 145 needlessly repeats the very similar code three times. Challenge 146 does the same thing but in fewer lines of code by moving that repeated code into a function. But there is an even deeper reason why the solution to 145 is lengthier than the solution to 144.
- In Challenge 144 there is only one level. We can see what order the flamingos start in, we know what order we want them to end up in, and we the programmers can plan in advance to use the fewest number of moves to get the flamingos where they belong.
- Challenge 145 has two levels. The moves that put the flamingos in order on level 1 won't work on level 2. That's why we need all those `if` statements to decide if we need to swap two flamingos or leave them alone because they are already in the right order.
- So, while the algorithm in Challenge 145 is longer, it is more general. It will put *any* three-item list of flamingos in order from least to greatest, no matter what order it started in. In the final lesson, we will see how to generalize this procedure even more, so it works on lists of any length.

Lesson 22 - Bubble Sort 2

CSTA Standards

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

Objectives

- Define bubble sort
- Write a bubble sort algorithm to order elements of a list from least to greatest or greatest to least
- Informally analyze the running time of the bubble sort algorithm
- Complete Challenges 147 - 150

Introduction - 10 min

- ACTIVITY - 10 MIN
- For this activity each student will need a set of seven distinctly numbered cards. Playing cards are perfect, or they can just take seven index cards and write a different number on each.
- Students should shuffle their set of cards and then lay them out (face up) in a row from left to right. Now their challenge is to sort the cards in order from least to greatest. The catch is that they can't move cards around

arbitrarily. The only thing they can do to affect the cards' order is to swap two adjacent cards.

- Once students have sorted their cards once, ask them to reshuffle and try again with the same rules. This time have them keep count of the number of swaps they make in the process of sorting the cards. When they finish, they should repeat and reshuffle one more time, again counting the swaps.
- After every student has done the sorting procedure three times, discuss the following questions:

- How many swaps did it take to sort the cards? The average should be around 11 or 12.
- Did you find any strategies to make the sorting easier? Student may say anything here, but one fairly natural strategy is to start by swapping the biggest number to the rightmost position, then the next biggest to the next position, and so on. Of course, it works equally well to sort the lowest numbers to the left first.

- Explain that today we are going to be writing a program to sort lists in order. The basic operation in these programs will be the same as in this activity, swapping a pair of adjacent items.

Playtime – 27 min

- LOG-IN - 2 MIN
- Students should log-in to their accounts as usual.
- PLAYTIME (1) - 10 MIN
- Have students begin Challenge 147 by removing line 13. Rather than deleting it, they should just put a # symbol in front of the line. That way, Python will treat the line as a comment, but it will be easy to "reactivate" the line later by removing the #.
- When students run this code, they will see that what it does is move the tallest flamingo to the rightmost place in the list. This illustrates why the sort algorithm we are developing is called a **bubble sort**: The largest elements of the list move to the end like bubbles rising to the top in a glass of soda.
- Students should study the loop on lines 7 - 9 carefully to understand how it works:

```
for index in range(total_flamingos-1):
    if flamingos[index].height > flamingos[index+1].height:
        swap(flamingos, index, index+1)
```

- The *if* statement compares the heights of two adjacent flamingos (*flamingos[index]* and *flamingos[index+1]*) and swaps them if the taller flamingo is on the left. The loop only runs through *total_flamingos - 1* because *index* refers to the left flamingo in the pair that are being

compared; the loop has to stop one place before the end of the list to "leave room" for the right flamingo.

- Let students restore line 13 and finish writing that loop. The two loops together will sort the tallest two flamingos into the right position. To finish the sort, they will have to write one more version of the loop to sort the last two flamingos.
- Before moving on to the next challenge, have students look at the three loops they have written. How are they the same? How are they different? Students should notice that the only difference between the three loops is the upper limit (second argument) of the `range()` function.
- PLAYTIME (2) - 5 MIN
- Start Challenge 148 by have students check out the two functions defined at the top of the given code. `swap()` is familiar; it swaps two elements of a list. `sort_once()` is new but not really; it is essentially the loop that we wrote out three times in the challenge.
- Remind students that the loop inside the `sort_once()` function needs to be called over and over with the upper limit decreasing each time. The first call "bubbles" the tallest flamingo to the end of the list, the second call bubbles the next tallest flamingo into the second position, and so on. That's what the loop on lines 13 and 14 is doing.
- Have students run the code before making any changes. The `print()` statement inside the loop shows the value of `num_flamingos_to_sort - index` with each iteration. These are exactly the numbers we need as the argument to `sort_once()` to progressively sort the `flamingos` list. So turning this into a solution to the challenge is just a matter of replacing the `print()` statement with `sort_once(flamingos, num_flamingos_to_sort - index)`.
- PLAYTIME (3) - 10 MIN
- Students should be ready now to take on Challenges 149 and 150 on their own.

- The bubble sort algorithm used in Challenge 149 is really the same as the one used in Challenge 148 but it is written a little differently. Instead of using a `sort_once()` function, that code is included directly in the `bubble_sort()` function. But it is really doing the same thing the same way. The hints in the comments of the starter code should steer students in the right direction.
- Challenge 150 asks students to sort from greatest to least instead of least to greatest. They can almost use the exact same code from Challenge 149, but with one tiny change.

Debrief - 8 min

- Have students go back to Challenge 149 and insert the following as the first line of the `bubble_sort()` function:

```
count = 0
```

- They also need to insert the following two lines right before the *if* statement:

```
count = count + 1
print(count)
```

- The effect of this code is simply to count the number of times the *if* statement (and possible swap) is executed while the code is running. Run the code. It will be a good bit slower because of the *print()* statement, but the end result should be that comparison in the *if* statement happens 10 times.
- Now let students make the same code changes to their solutions to Challenge 150. Before they run it, remind them that sorting five flamingos took 10 comparisons. Challenge 150 has 11 flamingos to sort. Ask them to predict how many comparisons it will take. This one sort will take quite a bit longer this time and students will probably be surprised to discover that sorting 11 flamingos takes 55 comparisons.
- The main idea here is that sorting a longer list with bubble sort takes longer, and the amount of time it takes grows a lot faster than the length of the list. Challenge 150 had barely more than twice as many flamingos as Challenge 149 but sorting it took more than five times as many comparisons. While the programs students have written in Banana Tales have been simple enough that we haven't had to worry about their running time, in many situations in the real world the challenge of programming is not just writing code that works but writing code that works efficiently.
- You can also ask your students what will be the minimum/maximum number of swaps that we can perform
    - This would be a different number than the number of comparisons
    - If the list is already sorted, then we will not need to swap any item
    - If the list is sorted the other way around, then each comparison will cause a swap or places
        - You can add the count after the call to the *swap()* function inside the *bubble_sort()* function and then print it at the end of the function.

Conclusion

- This concludes you and your students' adventures with Banana Tales, and it's a good time to congratulate them on how much they have learned. Python is a big and powerful language, and this course has covered a wide array of topics. Completing it successfully is no small feat. Take time to celebrate your students' success, but also remind them that this just a stop along their computer science journey, not the end. The real banana in life is knowledge, and there is always another one to catch.

Great Job!

you have completed ALL Of Banana Tales ☺

Your students can now code in Python!

•