Last updated: 12/19/2016

LESSON PLANS

Computer Science 3

Curriculum Summary

- Recommended Prerequisite: Computer Science 2
- 11 x 45-60 minute coding sessions

Overview

Now that students have a solid foundation in the most useful types of control flow (conditionals, functions, and events), they're prepared to level up both their conditional logic skills and their control flow control. Most of the differences in the programs the students want to write and the programs they know how to write start to fall away in Computer Science 3.

In this course, students will keep practicing their functions, events, and nested conditionals. On top of those, they'll get into more sophisticated operators and keywords. String concatenation will let players modify strings dynamically in their code to produce whatever text they want. Arithmetic will help players become more comfortable with using math in programming. All things in CodeCombat are objects, (that's the "object" part of object-oriented programming,) and these things have accessible attributes, such as a Munchkin's position or a coin's value; both are important to begin visualizing the internal structure of the objects that make up their game world. Alongside properties, students unlock the additional game mechanic of real-time input handling with flags. They then learn to use functions that return values, to break up computations into smaller pieces. The boolean *equality*, *inequality*, *or*, and *and* operators let them express compound conditionals. Combining those with computer arithmetic and properties lets players finally explore relative movement, directing their hero to dynamic locations. They also learn to work with time programmatically, and to manipulate their while-loops with the *break* and *continue* statements.

This guide is written with Python-language classrooms in mind, but can easily be adapted for JavaScript.

Scope and Sequence

Module	**First Level**	**Transfer Goals**
11. String Concatenation	Friend and Foe	Add strings together with `+`
12. Computer Arithmetic	The Wizard's Door	Do arithmetic with code (`+` `-` `*` `/`)
13. Properties	Backwoods Bombardier	Access object properties with `.`
14. Functions with Returns	Burlbole Grove	Write functions that return answers
15. Not Equals	Useful Competitors	Test whether two things are not the same
16. Boolean Or	Salted Earth	Execute if-statements if one of two things are true
17. Boolean And	Spring Thunder	Execute if-statements if both of two things are true
18. Relative Movement	The Mighty Sand Yak	Combine x- and y-properties and arithmetic for movement
19. Time and Health	Minesweeper	Code based on elapsed time and hero health
20. Break and Continue	Hoarding Gold	Skip or end while-loops with break and continue statments
21. Review - Multiplayer Arena	Cross Bones	Synthesize all CS3 concepts

Core Vocabulary

Concatenation - String concatenation is used to add two strings together with the **string concatenation operator:** `+`

Arithmetic - Addition, subtraction, multiplication, and division. Course 3 begins to ease the player into using math while coding. Levels catering to basic arithmetic address how to use math as needed in order to perform different actions effectively.

Property - Data about or belonging to an object. You get to it by specifying the object, then a dot, then the name of the property, like `item.pos`.

Flags - Real-time input devices. Up until now, students' CodeCombat programs haven't been interactive--there hasn't been real-time player input while the level is running. Now with flags, students have a way of sending input to their programs: clicking a mouse plants a flag that the hero can respond to with the `hero.findFlag()` function.

Return - A return statements lets a function compute a result value and return it to the place that called the function. When your functions can return their results, it's easier to break data-producing computations into smaller steps.

Boolean - A binary variable with two possible values: `True` and `False`. The `conditionals` you use in if-statements and even while-loops are evaluated to boolean results. Boolean logic is the way that boolean values combine to form a single boolean value.

Break - A way to exit out of a while loop early. Break statements say, "Break out the loop, we're done with it." You might use a break statement to move on to the rest of your program after a loop.

Continue - A way to skip back to the top of a while loop. Continue statements say, "Let's stop this loop here and continue at the top on the next iteration." If you don't need to finish a loop (because it doesn't need to do anything right now), you can use a continue statement.

Extra activities for students who finish Course 3 early:

- Help someone else
- Refine a multiplayer arena strategy in Cross Bones
- Write a walkthrough
- Write a review of the game
- Write a guide to their favorite level
- Design a new level

MODULE 11

String Concatenation

Summary

String concatenation is used to add, or combine, two strings together. Strings are structures of `"text inside quotes"` . Students will use the **string concatenation operator**, `+` to build a longer string out of two shorter strings, or to combine a string and a variable.

In CodeCombat, using strings and `hero.say()` is useful for communicating with friends in the game. These levels will prepare the student for more sophisticated communication using concatenated strings.

Transfer Goals

- Concatenate two strings with `"string1" + "string2"`
- Concatenate strings and variables, like `"string1" + variable1`
- Use proper spacing around concatenated variables

Standards

CCSS.Math.Practice.MP1 Make sense of problems and persevere in solving them.
CCSS.Math.Practice.MP2 Reason abstractly and quantitatively.
CCSS.Math.Practice.MP6 Attend to precision.

Instructive Activity: String Chuck (12 mins)

Explain (2 mins)

Strings are pieces of text inside quotes. The students have been using strings ever since Course 1. For example, in the `buildXY()` function, students use the string `"fence"` , to build a fence as in `hero.buildXY("fence", 34, 30)` . In the `attack()` function, the can choose to attack a chest by passing the string, `"Chest"` as a parameter, with `hero.attack("Chest")` .

In these levels, students will need to combine two strings together to form a longer string. In programming, this is referred to as **string concatenation**. The students will learn how to concatenate, or add, two strings together by using the string concatenation operator, `+` .

The syntax to concatenate two strings is as follows:

```
# results in the hero saying "Come at me, Treg!"
hero.say("Come at me, " + "Treg!")
```

Notice in the code above that each of the separate strings has its own set of quotes, but the `+` is not in quotes. The `+` is not part of either string, but is instead the operator that is placed between the two strings.

Notice as well the extra space in the first string, `"Come at me, "` . When concatenating strings, the second string is appended to the end of the first one, and both strings appear exactly as they are shown between the quotation marks. Thus, without the extra space, the hero would say `"Come at me,Treg!"` .

In addition to concatenating two strings, the students will learn to concatenate a string and a variable, for any variable is storing a string.

The following code shows the concatenation of a string and a variable:

```
ogre = hero.findNearestEnemy()
hero.say("Come at me, " + ogre.id)
```

By using a variable, the students are able to call out an ogre without hard-coding its name into their code. This will allow them to call out ogres without knowing their names first.

Notice again in the code above the extra space after the first string. Just as with two strings, concatenating a string and a variable simply appends the two strings together as they are written in the code. It is important to remember to include an extra space if the strings are meant to be separated by a space when they are concatenated.

Note that concatenation will only work with strings on either side of the `+`. Trying to concatenate a string with a different type, such as an integer, or with a variable that is not storing a string, will result in an error.

Interact (8 mins)

This activity will demonstrate how to use the string concatenation operator and also the importance of correctly using spaces when concatenating strings.

Guide the class through concatenating strings to make a common phrase. The goal of this exercise is for the class to collectively write a program like this:

```
noun = "wood"
verb = "chuck"
teacher.write("How much " + noun + " would a " + noun + verb + " " + verb + " if a " + noun + verb + "
    could " + verb + " " + noun + "?")
```

Start by writing this string on the board:

```
goal = "How much wood would a woodchuck chuck if a woodchuck could chuck wood?"
```

Explain to the class that you want to make this string by only writing the words `"wood"` and `"chuck"` once. Ask the students how this can be done, and guide them towards the idea of using variables to store the strings for reuse.

Then write the following lines on the board:

```
noun = "wood"
verb =
teacher.write("How much " + noun)
```

Ask students to fill in the verb and the rest of the phrase one string or variable at a time. Add an output variable under your goal variable to record the output as you go:

```
output = "How much wood"
```

Let the students find their own mistakes in the output. They will likely forget to add spaces in the strings at first and will get output like this:

```
output = "How much woodcould a woodchuckchuckif a"
```

Remind them that in order to get a space to appear in the concatenated output string, it needs to be included in the string before or after the variable. Additionally a string that is just a single space can be used to add spaces.

Once the program is complete, ask the class for a new noun and a new verb. Rewrite the variables and the final output accordingly. Example:

```
noun = "cheese"
verb = "spray"
teacher.write("How much " + noun + " would a " + noun + verb + " " + verb + " if a " + noun + verb + "
    could " + verb + " " + noun + "?")

goal = "How much wood would a woodchuck chuck if a woodchuck could chuck wood?"
output = "How much cheese would a cheesespray spray if a cheesespray could spray cheese?"
```

If the students are grasping the concept well and seem to be ready for an extra challenge, encourage them to try and write code that outputs the goal string by only writing the sequence `"ould"` once. Push them to see that they can create a variable for `"ould"` then concatenate it as so:

```
noun = "wood"
verb = "chuck"
ould = "ould"
teacher.write("How much " + noun + " w" + ould + " a " + noun + verb + " " + verb + " if a " + noun +
    verb + " c" + ould + " " + verb + " " + noun + "?")
```

Reflect (2 mins)

When have you used strings before in CodeCombat? (To attack by name, like `hero.attack("Treg")`; to `buildXY` by type, like `hero.buildXY("fence", 34, 30)`; to say passwords, like `hero.say("Hush!")`; etc.) **What kind of text can you put in a string?** (Any text you want!) **What does string concatenation mean?** (Adding a string to the end of another string.)

Coding Time (30-45 mins)

Allow the students to go through the game at their own pace, keeping notes about every level on paper or digital document. We recommend using following format, which you can also print out as templates: Progress Journal [PDF] (http://files.codecombat.com/docs/resources/ProgressJournal.pdf)

```
Level #: _____  Level Name: ____________________________________
Goal: __________________________________________________________
What I did:

What I learned:

What was challenging:
```

Circulate to assist. Draw students' attention to the instructions and tips, and especially for string concatenation, the error messages. You may need to reinforce that each string needs opening and closing quotes, and that in between strings and other strings or strings and variables, you always need a `+` to concatenate. Remind the students to double check the spacing and the types on either side of the `+`.

Students may run into errors with code like this:

```
hero.say("Take " + numToTakeDown " down, pass it around!")  # Missing second +
hero.say("Take " + numToTakeDown + down, pass it around!")  # Missing second opening "
```

If student have trouble figuring out an error, ask them to carefully review their string and concatenation syntax, or see if a classmate can spot the mistake. Encourage them as well to carefully read through each line and write down on paper what they think the output will be.

Written Reflection (5 mins)

When do you use the string concatenation operator, `+`?

> When you have to put two strings together, or when you have to put a string together with a variable. For example, if you don't know what the variable is ahead of time, but you need to do something with it, like sing it in a song, you can put the song lyric with a `+` and the variable.

How do you combine two variables into a string with a space between them?

> You can't just put the variables together like `x + y`, because they won't have a space. You have to put a string that just has a space in between, like `x + " " + y`.

Why do you think that the people who designed Python chose the `+` sign to represent concatenating strings together?

> Because what you really doing is adding one string to the other string, and the symbol for addition is `+`.

MODULE 12

Computer Arithmetic

Summary

Just like calculators, computers can be used to perform mathematical calculations. In fact, the word computer stems from the act of computing, in a mathematical sense. **Computer arithmetic** is writing code to have a computer perform mathematical operations.

Computers can be used to add, subtract, multiply, and divide numbers. Additionally, they can be used to perform operations on variables representing numbers, and the results of functions that return numbers.

Computer arithmetic is used in these levels to allow the students to dynamically calculate the magic numbers needed to get past a series of wizards. Students will have to edit and run their programs a number of times to get the instructions from each wizard and compute each of the magic numbers.

Transfer Goals

- Learn how to use the addition, subtraction, multiplication, division, and modulo operators: `+`, `-`, `*`, `/`, and `%`
- Perform arithmetic on "numeric literals" (like `2 + 2`)
- Perform arithmetic on variables (like `x - 5`)
- Perform arithmetic on properties (like `hero.pos.y + 10`)

Standards

CCSS.Math.Practice.MP1 Make sense of problems and persevere in solving them.
CCSS.Math.Practice.MP2 Reason abstractly and quantitatively.
CCSS.Math.Practice.MP5 Use appropriate tools strategically.
CCSS.Math.Practice.MP6 Attend to precision.
CCSS.Math.Practice.MP7 Look for and make use of structure.

Instructive Activity: Hailstone Numbers (15 mins)

Explain (2 mins)

Just as arithmetic can be done by hand or with a calculator, it can also be done with the use of computer code. The same operators that can be found on a basic calculator can also be coded quite easily.

Adding two numbers can be done with `+`, as so:

```
5 + 2  # answer: 7
```

Similarly, subtracting two numbers can be done with `-`:

```
5 - 2  # answer: 3
```

For multiplication, `*` is used instead of `x` to avoid confusion with variables or strings:

```
5 * 2  # answer: 10
```

Finally, division can be done with `/`:

```
5 / 2  # answer: 2.5
```

In addition to performing operations on numbers, operations can also be performed on variables that are storing numbers:

```
x = 3
5 * x  # answer: 15
```

An additional operation that can be performed with code is **modulo**. This operation does not have to be used in these CodeCombat levels, but may still be of use or interest to the students. Modulo is used to find the remainder after dividing two numbers:

```
5 % 3  # answer: 2
9 % 4  # answer: 5
6 % 2  # answer: 0 (no remainder)
```

The benefit of **computer arithmetic** is that computers are very fast, and thus the answers can be calculated almost instantly. This allows for programs to perform a large number of calculations while still running very quickly.

Interact (13 mins)

Explain to the class that you want their help writing code for an mathematical game called hailstone sequences. The general flow of the game is as follows:

- The class will pick a number
- If the number is even, it will be divided by 2
- If the number is odd, it will be multiplied by three then increased by 1
- The prior two steps are repeated until the number left over is 1

Demonstrate the activity to the class with the following example starting with the number 5. Be sure to write the steps as you go along. You may also choose to have a student write each step, if desired.

```
5 is odd
5 * 3 + 1 = 16, which is even
16 / 2 = 8, which is even
8 / 2 = 4, which is even
4 / 2 = 2, which is even
2 / 2 = 1, so we're done
-------------------------
5 total steps
```

Now tell the class that you want to write a function called hailstone that takes in a number as a parameter and performs hailstone (the steps above) on the number. The function should print out the sequence of steps generated along the way. Get the students to help create the function as you write it on the board:

```
def hailstone(number):
    teacher.write("Sequence: " + number)
    while number != 1:
        if isEven(number):
            number = number / 2
        else:
            number = number * 3 + 1
        teacher.write(" " + number)
```

Ensure the students understand the code, and particularly the arithmetic before moving on. They should understand that the line `number = number / 2` reassigns the variable `number` by dividing the original value by 2. The `number` on the left of the `=` holds the new value and the `number` on the right side holds the original one. This is true for the line `number = number * 3 + 1` as well.

Ask the students for a number between 2 and 10 to start with and then run through the program with them, writing out the numbers as they go:

```
hailstone(10)
Sequence: 10 5 16 8 4 2 1
```

If you explained the modulo operator and believe the students are up for an extra challenge, encourage them to replace the line `isEven(number)` with one line of code that will determine if the number is even. Push them to think about all of the operators they learned about before the activity.

The appropriate line of code to use is:

```
if number % 2 == 0:
```

Explain to the students that this code works because all even numbers are divisible by 2 and thus have a remainder of 0 when dividing by 2.

If time permits, walk the students through adding a counter to the code to keep track of how many steps it took:

```
def hailstone(number):
    teacher.write("Sequence: " + number)
    steps = 0
    while number != 1:
        if isEven(number):
            number = number / 2
        else:
            number = number * 3 + 1
        teacher.write(" " + number)
        steps = steps + 1
    teacher.write("Steps: " + steps)

hailstone(3)
Sequence: 3 10 5 16 8 4 2 1
Steps: 7
```

Ensure the students understand how the counter works in the code above. Share that `hailstone(27)` takes 111 steps and gets as high as 9232 before falling back down to 1.

Explain that these are called hailstone numbers because like hailstones, they go up and down a number of times before inevitably falling all the way. However, no one has been able to prove that this has to happen every time, even though computers can calculate the number of hailstone steps for numbers with thousands of digits instantly with the code on the board. If someone found a number that didn't eventually fall back to 1, they would be famous.

Reflect (2 mins)

What operations can you perform on a computer? What are the operators to use for them? (You can use a computer to do addition, subtraction, multiplication, and division. The operators you use are `+`, `-`, `*`, and `/`, respectively.) **What is the proper syntax to multiply a variable called `number` by 5 and store the result in `number`?** (`number = number * 5`)

Coding Time (25 mins)

Allow the students to go through the game at their own pace, keeping notes about every level on paper or digital document. We recommend using following format, which you can also print out as templates: Progress Journal [PDF] (http://files.codecombat.com/docs/resources/ProgressJournal.pdf)

```
Level #: _____  Level Name: ____________________________________
Goal: __________________________________________________________
What I did:

What I learned:

What was challenging:
```

Circulate to assist. Draw students' attention to the instructions and tips, and remind them that they'll have to edit and run their programs multiple times to get all the instructions. You may have to remind some students that the operator for multiplication is `*` and not `x`.

Written Reflection (5 mins)

Select appropriate prompt(s) for the students respond to, referring to their notes.

When does it make sense to use a computer to do math?

> When you have to do a lot of math really fast, like to calculate a big number. Or when you don't know what the values are ahead of time, so the computer can do the math on a variable.

What kind of math do you know how to do yourself but don't know how to use a computer to do, and how do you think you can do it with a computer?

> I can square numbers. Maybe there is a square function, like square(number)?

MODULE 13

Properties

Summary

Students have used **properties** in prior modules to do things like move their hero to a specific position and to check an enemy's type. Properties are specific attributes of objects that can be used to distinguish them. In these levels, students will see the importance of using properties with the use of **flags**.

Flags give the game a real-time element. Students place flags on the game screen and have their hero respond to them. Flags are placed after the game is already running, and thus properties must be used to access them since the students cannot predict exactly where they will be placed.

Transfer Goals

- Access a property using dot notation.
- Save a property in a variable.
- Understand the difference between a property and a function.

Standards

CCSS.Math.Practice.MP1 Make sense of problems and persevere in solving them.
CCSS.Math.Practice.MP2 Reason abstractly and quantitatively.
CCSS.Math.Practice.MP6 Attend to precision. **CCSS.Math.Practice.MP7** Look for and make use of structure.

Instructive Activity: Toy Properties (10 mins)

Explain (3 mins)

A **property** is an attribute, or trait, of an object. For example, an enemy object has properties such as type and position. The flag object, which students will use extensively in these levels, has a position property.

Properties are similar to functions, because both functions and properties are things that belong to the object. They differ, however, because functions are like actions or verbs and properties are like aspects (adjectives) or possessions (nouns).

Properties can be accessed by specifying the object, then `.` then the name of the property. The following code returns the position property of a flag object:

```
flag.pos
```

Some properties are also objects and thus have properties of their own. For example, the position property is an object with two additional properties, one for the x position and one for the y position. These can be accessed by adding another dot and the second property name as so:

```
flag.pos.x
```

Once a property is accessed, its value can be found and used in the code. For each object type, different instances of each object have the same properties that are accessed in the same way, but those properties can, and likely will, have different values.

For example, different flags have the same way of accessing their position property, but the values of each flag's position may be different. The differences in these values allow each object to be distinguished from each other and acted on separately.

Interact (5 mins)

For this activity, bring in a unique stuffed animal or doll to class. It would be best if the animal or doll has many distinct features, such as different colored hair than its fur or skin, a tail, unique clothing, etc. The more fun or wacky the animal or doll is, the more likely it is that the children will have fun with the activity.

With help from the students, record different properties of the doll on the board. Be sure to use proper syntax when recording the properties, as so:

```
doll.hair
```

Encourage the students to suggest properties of other properties, such as:

```
doll.hair.color
```

After each property is written correctly, have the students help you fill in the values, so that each line is built to look like these below:

```
doll.hair.color = "blue"
doll.fur.length = "short"
doll.legs.amount = 4
```

Build a list of at least ten different properties by following the same pattern of having the students suggest a property, writing the property correctly on the board, then adding the correct value. If desired, you may have a student help you write everything on the board. Your list should

Once the list is complete, ask the students if there are any properties on the list that may be shared by all dolls similar to the one you brought. Encourage them to think about whether or not every doll has the same value for that property. If possible, you may wish to bring in a second doll that has the same property, but a different value for that property (such as green hair instead of blue hair).

Reflect (2 mins)

What is a property? (An attribute of an object) **How can you tell the difference between a function and a property?** (Functions have parentheses () and properties do not. Also, functions perform actions, while properties describe attributes about objects.)

Can two objects of the same type have the same property? Explain. (Yes because they are the same type, they likely have the same properties. For example, every enemy in the game has a type.)

Do two objects' properties always have the same value if the objects are of the same type? Explain. (No. For example, there are many different types of enemies in the game, so even though all enemies have the type property, the value of this property can be different between them.)

Coding Time (30-45 mins)

Allow the students to go through the game at their own pace, keeping notes about every level on paper or digital document. We recommend using following format, which you can also print out as templates: Progress Journal [PDF] (http://files.codecombat.com/docs/resources/ProgressJournal.pdf)

```
Level #: _____  Level Name: ___________________________________
Goal: __________________________________________________________
What I did:

What I learned:

What was challenging:
```

Circulate to assist. Draw students' attention to the instructions and tips. Flags can be tricky for some students, so allow them to pair up to beat the levels. Each student should write their own code, but it's ok for another student to place the flags for them.

If students are having trouble placing the flags, be sure to direct them to the instructions. Students can place flags by clicking on the flag color, or typing the first letter of the color, then clicking on the screen to place the flag. Students should use `hero.pickUpFlag()` to make the hero go to the flag and clear it.

Written Reflection (5 mins)

How did you use properties today?

> I used properties to determine where flags were so the hero could move to them. To do this, I used the flag's property called pos and the pos properties, x and y to determine where the flag was.

Tell me about flags.

> You use flags to tell the hero what to do when the game is running. You can write code to say if there’s a flag, then go to it. Flags have a pos that has x and y. X is right-left and y is up-down.

MODULE 14

Functions with Returns

Summary

Return statements are used to create functions that compute and return a value, rather than just perform an action. If a function contains a `return` statement, it will be equal to whatever value it returns whenever it is called.

When a function gets to a `return` statement, the value is returned immediately and the flow of control is also returned to the place at which the function was called. This causes the function to end immediately.

In these levels, the students will use `return` statements to return boolean values and numbers for enemy distances.

Transfer Goals

- Write functions that `return` answers
- Use `return` statements to exit functions
- Use the value that is returned by a function

Standards

CCSS.Math.Practice.MP1 Make sense of problems and persevere in solving them.
CCSS.Math.Practice.MP2 Reason abstractly and quantitatively.
CCSS.Math.Practice.MP6 Attend to precision. **CCSS.Math.Practice.MP7** Look for and make use of structure.

Instructive Activity: Vending Machine (12 mins)

Explain (2 mins)

Previously, the students have been writing functions that make their hero or pet perform an action. For example, the `goFetch()` function makes the pet fetch an item. The `moveXY` function makes the hero move to a specific position.

In these levels, the students will learn how to write code using **return statements**. `return` statements will allow the studetns to create functions that instead of performing an action, perform a computation and return the result with a `return` statement.

The following code shows an example of a function that uses `return` statements:

```
def howMany(things):
    if things == 1:
        return "a"
    if things == 2:
        return "a couple"
    if things <= 4:
        return "a few"
    if things <= 7:
        return "several"
    return "a lot of"

teacher.say("I see " + howMany(hats) + " hats.")
```

Notice in the code above that each `if` statement contains a `return` statement. This ensures that only one value will be returned by the function, and that value is conditional based on the number that is passed in as a parameter.

Notice as well the last `return` statement located outside of the `if` statements. This `return` statement is executed if none of the conditionals of the `if` statements above are met. If using conditionals and `return` statements, it is important to make sure that the function will always return something, even when the conditions are not met.

The value that is returned by a function can be used just as any variable can. Notice the final line in the code segment above. `howMany(hats)` is concatenated with the rest of the string to form a full sentence as output from the teacher.

It is important to note that when a function returns a value, it also returns the flow of control back to the place from which it was called. This ensures that only one value will be returned by the function. Note that because of this, any code written within a function but below a `return` statement is unreachable.

For example, the following code would generate an error:

```
def howMany(things):
    if things == 1:
        return "a"
        things += 1  # this line is unreachable

    # more code below here
```

Because the function is stopped once it hits a `return` statement, no additional code within the function is executed once a value is returned. Thus proper planning and indentation are particularly important when writing functions with `return` statements.

Interact (8 mins)

In this activity, you will work with the students to write the code for a simple vending machine. To make the activity more interactive, gather the following materials:

- 4 or more boxes
- One different snack for each of the boxes

You may also choose to put beverages, or even toys, in each of the boxes. If you do choose an item besides food, be sure to change the variable and function names in your sample code so that the appropriate noun and verb are used.

Set up the boxes so that the bottom of each box is facing the class and the open top is facing you. Place the boxes in at least two rows so that you have some sort of a grid system. Label the bottom of the boxes (the side facing the students) with A1 in the top left corner, A2 to the right of A1, B1 below A1, and so on. The end result should be a representation of a vending machine.

Get the students to help you write the code for the vending machine. Start with this vending machine skeleton:

```
def vend(button):
    if button == "A1":
        return ""

while True:
    button = class.pressButton()
    food = vend(button)
    class.eat(food)
```

Be sure that the students understand the code above. Currently A1 is still returning something, but it is just an empty string. Thus, if the students were to try and vend the item now to eat it, they would be eating nothing.

Get a student to come up and "press" the A1 button. Reveal what is behind that button then write it in as the first string to `return` . Feel free to have fun with this by having the students give you some form of money for the items or by choosing silly or unexpected items to be behind each button.

Have the class help you push the rest of the buttons to discover the additional food items and write the rest of the `if` and `return` statements. You might end up with something like this:

```
def vend(button):
    if button == "A1":
        return "Cheetos"
    elif button == "A2":
        return "Apple"
    elif button == "B1":
        return "Slime"
    elif button == "B2":
        return "A bear"

while True:
    button = class.pressButton()
    food = vend(button)
    class.eat(food)
```

Ensure the students understand how the code above works and why `elif` statements were used (because only one condition can be true). Quiz them briefly by asking which item they will get if they press each of the buttons. It may be helpful to point to each line as the students "press" each button to show the flow of control. Remember the flow is as follows:

- Press button

- Vend button
- Check each conditional until right button is found
- Return appropriate string
- Assign `food` variable to the string
- Eat the food

Reiterate how the `vend` function is using `return` statements to return food values when the function is called. Ask the students if there is any scenario in which they would not get something returned from the vending machine (there is not because each condition has a `return` statement).

Next, show the class how to modify the code by moving all of the `return` statements to just `return` a `result` variable:

```
def vend(button):
    result = "money"
    if button == "A1":
        result = "Cheetos"
    elif button == "A2":
        result = "Apple"
    elif button == "B1":
        result = "Slime"
    elif button == "B2":
        result = "A bear"
    return result

while True:
    button = class.pressButton()
    food = vend(button)
    if food != "money":
        class.eat(food)
```

Be sure the students understand the new code and why returning a single variable at the end will work. Again, have the students "press" buttons while pointing to the corresponding line of code to show the flow of control. The flow of control is now as follows:

- Press button
- Vend button
- Initialize `result` variable as money
- Go through conditionals until the right one is found
- Set `result` to the appropriate string
- Return `result`
- Set `food` to be the return value (`result`)
- Eat the food (if it's not money)

It may be necessary to remind the students that variables cannot be used outside of the function in which they were defined, unless they are passed as parameters. Thus, in the example above, `food = result` would not be a valid line of code. Using `return` statements allows for virtually the same outcome, however.

If you do not have boxes for this activity, you can simply draw a picture of a vending machine with four buttons on the board, and label the buttons A1, A2, B1, and B2 in a 2x2 grid. Instead of having students discover which item is behind each button, you can have them simply suggest different items for each of the buttons.

Reflect (2 mins)

What is a return value used for? (return values are used so that functions can be created to perform computations and return them to another function. This allows the code to be better organized.)

What are some built-in CodeCombat functions you use that `return` values? (`hero.findNearestEnemy()` , `hero.isReady("cleave")` , `hero.distanceTo(target)` , `hero.findNearestItem()`)

Why does a return statement immediately exit a function? (Because if you called `return` twice, you wouldn't know which value to use.)

Coding Time (30-45 mins)

Allow the students to go through the game at their own pace, keeping notes about every level on paper or digital document. We recommend using following format, which you can also print out as templates: Progress Journal [PDF] (http://files.codecombat.com/docs/resources/ProgressJournal.pdf)

```
Level #: _____  Level Name: ____________________________________
Goal: __________________________________________________________
What I did:

What I learned:

What was challenging:
```

Circulate to assist. Draw students' attention to the instructions and tips. Whenever a student is having trouble with a function, have them go execute the function themselves so they can say exactly what value it will return.

Remind students to make sure that they do something with the return value of the function once they have called it. The following examples show correct and incorrect usage of return values:

```
# Correct: storing return value in a variable, then using if
canAttack = inAttackRange(nearestEnemy)
if canAttack:
    hero.attack(nearestEnemy)

# Correct: using return value directly in if
if inAttackRange(nearestEnemy):
    hero.attack(nearestEnemy)

# Incorrect: not doing anything with return value
inAttackRange(nearestEnemy)
hero.attack(nearestEnemy)
```

Written Reflection (5 mins)

When are functions with returns useful?

> When you want to figure something out, like whether to attack an enemy or pick up a coin, instead of just attacking it directly inside the function. Or when you want to get some return value from outside your code, like with findNearestEnemy().

Naming functions that return values is important. Come up with three function names that would return a value useful in daily life, and write some example values they would return to make sure the names make sense.

> `whatTimeIsIt()` would return the time, like `"7:30 am"` or `"1:11 pm"` . `findColor(thing)` would return what color something is, like `"red"` or `"mahogany"` . `isFriend(person)` would return whether someone likes you, either `True` or `False` .b

MODULE 15

Not Equals

Summary

The students have used the **equality operator**, `==`, in prior modules to test conditional expressions for `if` statements. The equality operator checks to see if the values on either side are equal to each other. The expression returns `True` if they are and `False` if they are not.

In these levels, the students will learn about the **inequality operator**, `!=`. This operator works similar to `==`, but instead of checking to see if the values on either side are equal to each other, it checks to see if they are *not* equal to each other. The expression returns `True` if they are not equal and `False` if they are.

The students will use `!=` in these levels to keep their heroes safe by helping them avoid dangers such as poison.

Transfer Goals

- Test whether two things are not the same
- Use `!=` and `==` appropriately in code
- Read `!=` as "not equals"

Standards

CCSS.Math.Practice.MP1 Make sense of problems and persevere in solving them.
CCSS.Math.Practice.MP2 Reason abstractly and quantitatively.
CCSS.Math.Practice.MP6 Attend to precision. **CCSS.Math.Practice.MP7** Look for and make use of structure.

Instructive Activity: Picky Eating (10 mins)

Explain (2 mins)

In the prior course and in prior modules of this course, the students have written a number of `if` statements similar to this:

```
if gem.pos.x == 34:
        hero.say('left!')
```

Notice that the conditional expression in the `if` statement uses the **equality operator**, `==` to check if `gem.pos.x` is equal to `34`. The students have used this operator many times and should be familiar with it. Expressions containing `==` return `True` if the two values being compared are equal and `False` if they are not.

In this module, the students will learn about a new operator called the **inequality operator**. Rather than checking to see if two values are equal to each other, the inequality operator checks to see if the two values are *not* equal to each other.

The symbol for the inequality operator is `!=`. `!` is the equivalent of `not`, and is thus placed before `=` to translate as "not equal".

Below is an example of code that uses `!=` to see if an item is not a gem:

```
item = hero.findNearestItem()
    if item:
        if item.type != "gem":
            hero.moveXY(item.pos.x, item.pos.y)
```

The code above first looks for an item, then if one is found checks to see that the item is *not* a gem. If the item is not a gem, then the expression `item.type != "gem"` returns `True`. If the item is a gem then the expression returns `False`.

Just like `==`, `!=` can be used with numbers, variables, strings, and properties on either side of the operator. In these levels, the students will practice using `!=` to compare these different data types.

Interact (6 mins)

Tell the class to imagine it's 4:00 am and they wake up for a snack, going to the fridge in zombie mode. As a zombie, they aren't thinking straight, so we need to write a simple algorithm for them to follow in their snacking. Write the following code on the board:

```
fridge = zombie.findNearestFridge()
zombie.moveXY(fridge.pos.x, fridge.pos.y)
while True:
    food = zombie.ransack(fridge)
```

Ask the class what do to next with the `food` variable, guiding them to create a function call like `zombie.eat(food)` . Ask if they want to eat just any food, or if they want to eat a specific food. The students should mention a number of different food items they want to eat. You can get their help to start coding this as so:

```
fridge = zombie.findNearestFridge()
zombie.moveXY(fridge.pos.x, fridge.pos.y)
while True:
    food = zombie.ransack(fridge)
    if food.type == "cake":
        zombie.eat(food)
    if food.type == "cookies":
        zombie.eat(food)
    if food.type == "fruit":
        zombie.eat(food)
    if food.type == "ice cream":
        zombie.eat(food)
```

The students should see that coding a solution in this manner would take a lot of time and require many lines of code, especially when considering all the different options that the students may want to eat.

Ask the students if instead of specifying foods they *do* want to eat, if there is a specific food they want to *avoid* eating. Take the first food item mentioned and adapt the code to include that food item with an inequality operator, as shown below:

```
fridge = zombie.findNearestFridge()
zombie.moveXY(fridge.pos.x, fridge.pos.y)
while True:
    food = zombie.ransack(fridge)
    if food.type != "broccoli":
        zombie.eat(food)
```

Explain that as with the example above, you could create a number of conditions for all of the food items that the zombie would not want to eat. Ask the students if perhaps they can instead just make sure to avoid foods with any given attribute, like shape or color. Then add a nested inequality comparison to the code to incorporate that:

```
fridge = zombie.findNearestFridge()
zombie.moveXY(fridge.pos.x, fridge.pos.y)
while True:
    food = zombie.ransack(fridge)
    if food.type != "broccoli":
        if food.color != "green":
            zombie.eat(food)
```

Explain that because they want to eat most foods they shuold not explicitly name each food they *do* want to eat with `==` . Instead, they can use `!=` to avoid the items they *don't* want to eat. The students should understand that now the zombie will eat all other foods that are not green and not broccoli.

Note that the students have not learned how to do a compound conditional yet, but if students ask, they will learn how to do that in the next two modules. This will allow them to write statements such as: `if food.type != "broccoli" and food.color != "green"` .

Reflect (2 mins)

What would be returned from 1 * 2 != 3? (True because 2 does not equal 3.) **How would you write an `if` statement to check if an item is not a gem?** (`if item.type != "gem":`)

Coding Time (30-45 mins)

Allow the students to go through the game at their own pace, keeping notes about every level on paper or digital document. We recommend using following format, which you can also print out as templates: Progress Journal [PDF] (http://files.codecombat.com/docs/resources/ProgressJournal.pdf)

```
Level #: _____  Level Name: ____________________________________
Goal: __________________________________________________________
What I did:

What I learned:

What was challenging:
```

Circulate to assist. Draw students' attention to the instructions and tips. When students are checking `item.type` and `enemy.type`, remind them to make sure they are spelling the types correctly: `if enemy.type != "peon"`, `if item.type != "poison"`, and `if item.type != "gem"`.

Have students pay close attention to the yellow arrows indicating where to code, since they sometimes need to modify existing `if` conditions.

Written Reflection (5 mins)

Select appropriate prompt(s) for the students respond to, referring to their notes.

What are `==` and `!=` and how do you pronounce them?

> `==` is the equality operator, and you say "is equal to".
> `!=" is the inequality operator, and you say "is not equal to". `!=` is the opposite of `==`.

Where do you use `==` and `!=` in your code?

> You use them in if-statements, because you have to decide whether to do something or not based on whether two values are the same or different.

MODULE 16

Boolean Or

Summary

A **boolean** is a data type with two possible values, `True` or `False`. Students have used booleans in prior modules with the use of `if` statements and `while` loops. The conditionals used in these are expressions that must be either `True` or `False`, and thus are boolean expressions. The statement or loop is executed if the condition is `True`, and not executed if the condition is `False`.

Boolean logic is a form of arithmetic that is performed on boolean values. The result of boolean logic is always a single boolean value, which is either `True` or `False`.

One operator that is used in boolean logic is the **boolean or** operator, `or`. When using `or`, if one or both values in the expression is `True`, then the entire expression evaluates to `True`. If both values in the expression are `False`, then the entire expression evaluates to `False`.

Transfer Goals

- Define what a boolean value is
- Understand how to use the boolean `or` operator
- Execute `if` statements if one of two conditions are true

Standards

CCSS.Math.Practice.MP1 Make sense of problems and persevere in solving them.
CCSS.Math.Practice.MP2 Reason abstractly and quantitatively.
CCSS.Math.Practice.MP6 Attend to precision. **CCSS.Math.Practice.MP7** Look for and make use of structure.

Instructive Activity: Simon Says - Or (10 mins)

Explain (2 mins)

Booleans are data types that have two possible values, `True` and `False`. Although the students have not extensively learned about boolean values, they have used them many times in CodeCombat before.

For example, every `while` loop seen thus far has been set to `True`, as so:

```
while True:
    # do something
```

In addition, students have used functions that return boolean values, such as the `isReady` function:

```
hero.isReady("cleave") # this function returns True if the cleave is ready and False if not
```

They have then used the result of such functions to perform different actions based on the return value:

```
# if hero.isReady("cleave") returns True then the hero will cleave
if hero.isReady("cleave"):
    hero.cleave()
```

In the code above, the hero will cleave if `hero.isReady("cleave")` returns `True`. If the function returns `False`, then the hero will not cleave and the control flow will move to the next line outside of the `if` statement.

Expressions can also be evaluated to a boolean value. The students saw this in the past few levels, with code such as this:

```
if item.type == 'coin'
    # do something
```

Just as with boolean values that are returned, in the code shown above, the code within the `if` statement will be executed if `item.type == 'coin'` evaluates to `True`. If it evaluates to `False` then the code in the `if` statement will not be executed.

In this module and the next, the students will learn how to perform operations on boolean values with the use of **boolean logic**. Boolean logic is a form of algebra in which the operands (values being operated on) and result are all boolean values.

In these levels, the students will use an operator called **boolean or**. In Python, this operator is written by typing the word `or` between two boolean expressions or values, as so:

```
enemy.type == 'thrower' or enemy.type == 'munchkin'
```

Note that both sides of the `or` are entire expressions that can be evaluated to either `True` or `False`. `or` will only work with boolean values, so it is necessary when using `or` between expressions that they are both boolean expressions that are fully written out.

The result of a boolean `or` operation is determined by the values on either side of the `or`. If one or both values are `True`, then the expression returns `True`. If both values are `False` then the expression returns `False`.

```
hero.say(False or False) # Hero says 'False'
hero.say(False or True) # Hero says 'True'
hero.say(True or False) # Hero says 'True'
hero.say(True or True) # Hero says 'True'
```

Interact (7 mins)

This activity is an adaptation of the popular game Simon Says. Instead of beginning instructions with "Simon says", however, you will begin each instruction with an `if` statement containing `or`. Instead of listening for the phrase "Simon says", the students have to determine if either of the conditionals is `True` for them before following or not following the instruction.

For example, one instruction could be, "if your name starts with "A" or your name starts with "B", put your hands on your head". Only those students whose names start with "A" or "B" should put their hands on their head.

Each statement should be written on the board after each move. Be sure to use the proper syntax when writing the statements. For example, code for the statement above could be written as so:

```
if name.startsWith("A") or name.startsWith("B"):
    hands.putOn(head)
```

As with Simon Says, there are a few ways for the students to be eliminated:

- Perform the wrong action
- Perform an action when they are not supposed to (i.e. neither expression is `True` for them)
- Fail to perform an action when they are supposed to

When students are eliminated, instead of having them return to their seats, you may choose to have them help with any of the following tasks:

- Write the `if` statements on the board
- Provide additional `if` statements or boolean expressions for the instructions
- Act as additional eyes to help see when other students should be eliminated

Feel free to be creative with the conditions and instructions throughout this activity. You may also choose to end the game whenever you feel that the students have a good grasp on the concept. Additionally, you can run the game again if you think they need more practice.

Reflect (1 min)

What is `or` useful for? (To determine if one or more conditions in an expression is `True`.) **What is a boolean value?** (A value that is either `True` or `False`.)

Coding Time (30-45 mins)

Allow the students to go through the game at their own pace, keeping notes about every level on paper or digital document. We recommend using following format, which you can also print out as templates: Progress Journal [PDF] (http://files.codecombat.com/docs/resources/ProgressJournal.pdf)

```
Level #: _____  Level Name: ____________________________________
Goal: __________________________________________________________
What I did:

What I learned:

What was challenging:
```

Circulate to assist. Draw students' attention to the instructions and tips. Remind the students that both sides of the `or` need to have entire expressions that can be evaluated to either `True` or `False` :

```
# incorrect, since the computer sees (enemy.type == "thrower") or "munchkin"
if enemy.type == "thrower" or "munchkin":

# correct
if enemy.type == "thrower" or enemy.type == "munchkin":
```

Written Reflection (5 mins)

What did you use `or` for in these lessons?

> I used `or` to pick up coins or gems, but not harmful objects. I also used it to attack only certain kinds of enemies.

What is the `type` property? What types of things have you seen in CodeCombat so far?

> The `type` property is a string telling you what kind of object something is, like `"munchkin"` , `"thrower"` , `"burl"` , `"gem"` , `"coin"` , and `"poison"` .

MODULE 17

Boolean And

Summary

In the next few levels, the students will learn about a second boolean operator, **boolean and**. Just as the boolean or is written as `or` , boolean and is written as `and` .

When using `and` , if both values in the expression are `True` , then the entire expression evaluates to `True` . If one or both of the values is `False` , then the entire expression evaluates to `False` . In this sense, `and` is almost the opposite of `or` .

Students will use `and` to execute actions only when two conditions are `True` . In the later levels, they will need to combine their knowledge of `and` , `or` , and `not` to communicate with wizards.

Transfer Goals

- Understand how to use the boolean `and` operator
- Understand the difference between the `and` and `or` operator
- Execute if-statements if both of two things are true

Standards

CCSS.Math.Practice.MP1 Make sense of problems and persevere in solving them.
CCSS.Math.Practice.MP2 Reason abstractly and quantitatively.
CCSS.Math.Practice.MP6 Attend to precision. **CCSS.Math.Practice.MP7** Look for and make use of structure.

Instructive Activity: Simon Says - And (10 mins)

Explain (2 mins)

In the past few levels, the students used the boolean `or` to find the result of expressions containing `==` and `!=` , like so:

```
if item.type == "gem" or item.type == "coin":
    hero.moveXY(item.pos.x, item.pos.y)
```

In these levels, the students will learn about boolean `and` , a boolean operator that works similarly but provides different results.

The syntax for using `and` is the same as `or` , except with a different operator:

```
if item.type == "coin" and item.value == 2:
    # do something
```

Note that both sides of the `and` are expressions that can be evaluated to boolean values.

The result of a boolean `and` operation differs from that of `or` . If both values on either side of the `and` are `True` , then the expression returns `True` . If one or both values are `False` , then the expression returns `False` .

```
hero.say(False and False) # Hero says 'False'
hero.say(False and True) # Hero says 'False'
hero.say(True and False) # Hero says 'False'
hero.say(True and True) # Hero says 'True'
```

Because a boolean `and` will always result in `False` if either operand (value on either side of `and`) is `False` , if the first value is found to be `False` , the entire expression is immediately evaluated as `False` , without checking the rest of the boolean operation. This allows for code such as this to be written:

```
# checks if there is an enemy and if the enemy is a dragon
if enemy and enemy.type == "dragon":
    # do something
```

If there is no enemy present, then the first part of the boolean operation is deemed to be `False` . Thus, the second part of the `if` statement will simply not be checked since it can already be determined that the expression will evaluate to `False` . Hence, a variable that may not even be present can be referenced without the code generating an error.

Interact (6 mins)

This activity is a repeat of the Simon Says one used in the last module. Instead of using `if` statements containing `or` though, you will use `if` statements containing `and` to instruct the students.

For example, one instruction can be, "if your name starts with 'A' and you have brown hair, put your hands on your hips". Only the students whose names start with "A" **and** who have brown hair should place their hands on their hips.

As with the last activity, be sure to write the statement on the board after each move, using proper syntax. For example, code for the statement above could be written as follows:

```
if name.startsWith("A") and hair.color == 'brown':
    hands.putOn(hips)
```

Students can be eliminated for any of the following reasons:

- Performing the wrong action
- Performing an action when they are not supposed to (i.e. either expression is `False` for them)
- Fail to perform an action when they are supposed to

When students are eliminated, instead of having them return to their seats, you may choose to have them help with any of the following tasks:

- Write the `if` statements on the board
- Provide additional `if` statements or boolean expressions for the instructions
- Act as additional eyes to help see when other students should be eliminated

You should be sure to have some `if` statements that apply to none of the students in order to make sure they fully grasp the concept. For example, you could say "if your name starts with 'A' and your name starts with 'B' put your hands on your hips". In this instance, none of the students should perform the action, since each name can only start with one letter.

Feel free to be creative with the conditions and instructions throughout this activity. You may also choose to end the game whenever you feel that the students have a good grasp on the concept. Additionally, you can run the game again if you think they need more practice.

Reflect (2 mins)

What is the `and` operator used for? (To perform an action if two conditionals are both `True`) **What is some code you have written in CodeCombat that you can use `and` to simplify?** (Checking whether there is an enemy and cleave is ready, or if cleave is ready and the enemy is close enough.)

Coding Time (35-45 mins)

Allow the students to go through the game at their own pace, keeping notes about every level on paper or digital document. We recommend using following format, which you can also print out as templates: Progress Journal [PDF] (http://files.codecombat.com/docs/resources/ProgressJournal.pdf)

```
Level #: _____  Level Name: ____________________________________
Goal: __________________________________________________________
What I did:

What I learned:

What was challenging:
```

Circulate to assist. Draw students' attention to the instructions and tips. Remind students to read their compound `if` statements aloud to make sure they make sense. Also remind them, if necessary, that both sides of a boolean operator have to be full expressions that evaluate to boolean values.

Written Reflection (5 mins)

Select appropriate prompt(s) for the students respond to, referring to their notes.

Challenge: what happens in code like `if item and item.type == "gem":` ?

> `item` gets converted to `True` or `False` , depending on whether it exists, and `item.type == "gem"` gets converted to `True` or `False` depending on whether it's a gem, and then the `and` combines them into `True` if the item exists and is a gem, otherwise `False` .

Given an `enemy` variable, can you think of a way to use boolean `and` to both check if there's an enemy and to check if the enemy is closer than 10 meters, in one line?

```
if enemy and hero.distanceTo(enemy) < 10:
```

Make up an `if` example, either in CodeCombat or real life, that uses both `and` and `or` on the same line to combine three boolean values.

```
if fridge.hasFood() and (me.isHungry() or me.isBored()): me.open(fridge)
```

MODULE 18

Relative Movement

Summary

In prior modules, the students learned how to move their heroes to a particular spot by using `hero.moveXY()` and passing in numbers or properties as the coordinate values.

In this module, the students will combine their knowledge of computer arithmetic and properties to learn about **relative movement**. This will allow the students to move their hero dynamically by specifying coordinates that are relative to a known position.

The students will execute relative movement by adding and subtracting from properties and values to create new x and y position values for their hero to move to. They will use this new skill in these levels to move their hero relative to the current position in order to dodge obstacles.

Transfer Goals

- Use `moveXY()` to move relative to dynamic positions with coordinate arithmetic
- Internalize how positive and negative `x` and `y` coordinates relate to movement up, down, left, and right
- Combine relative movement with loops and conditionals to produce desired movement patterns

Standards

CCSS.Math.Practice.MP1 Make sense of problems and persevere in solving them.
CCSS.Math.Practice.MP2 Reason abstractly and quantitatively.
CCSS.Math.Practice.MP6 Attend to precision. **CCSS.Math.Practice.MP7** Look for and make use of structure.

Instructive Activity: Teacher Patrol (12 mins)

Explain (2 mins)

Up until now, the students have used three types of movement:

```
hero.moveRight()
hero.moveXY(34, 20)
hero.moveXY(item.pos.x, item.pos.y)
```

Now that they know how to do computer arithmetic, the students can use those skills to make their hero move dynamically with **relative movement**. Using relative movement, students can move their heroes relative to something else, such as their previous position or an enemy unit.

Below is code that will move the hero to a new position relative to their current position:

```
x = hero.pos.x
y = hero.pos.y

x += 10
hero.moveXY(x, y)
```

Notice in the code above that the variables `x` and `y` are set to the hero's original position. `x` is then incremented by 10, meaning that its value is now 10 greater than it was originally. Calling `hero.moveXY()` with `x` and `y` now moves the hero 10 units to the right. Because the `y` value did not change, the hero moves only horizontally.

It is important to note that when `x` is decreased the hero moves to the left; when it is increased the hero moves to the right. When `y` is decreased the hero moves down; when it is increased the hero moves up.

Interact (8 mins)

Explain to the class that the goal is to write a program to make you (the teacher) walk in a square around a student whenever the student claps.

Ask for a volunteer to stand in front of the class and clap. Have the class help write the event handler from scratch, prompting them for a function name and the code to start listening for a clap event from a student. The code should look similar to this:

```
def heardClap():


student.on("clap", heardClap)
```

Now draw a diagram on the board of a square with a dot in the middle, and label the dot as `{x: 0, y: 0}` . Say you're going to start in the top right. Label it `{x: 5, y: 5}` , and write the first line of your function:

```
def heardClap():
    teacher.moveXY(student.pos.x + 5, student.pos.y + 5)

student.on("clap", heardClap)
```

Have the student face the board and tell the chosen student to clap. Move to the corresponding coordinate about five feet to the right and five feet in front of the student.

Get the class to work through the rest of the program on the board to create to a solution that correctly has you walking in a square. Have the student clap every time a new line of code is added to test the solution. Act out the new line of code each time, even if it is wrong, so the students can see the outcome of their code.

If you are moving clockwise, your code might look like this. However, it is up to you and the class which order to move in and which way your axes are aligned.

```
def heardClap():
    teacher.moveXY(student.pos.x + 5, student.pos.y + 5)
    teacher.moveXY(student.pos.x + 5, student.pos.y - 5)
    teacher.moveXY(student.pos.x - 5, student.pos.y - 5)
    teacher.moveXY(student.pos.x - 5, student.pos.y + 5)
    teacher.moveXY(student.pos.x + 5, student.pos.y + 5)

student.on("clap", heardClap)
```

Pay attention to the code they suggest, since it may not be what they mean. The students will probably make mistakes that involve you walking diagonally across the square, bumping into the student in the center. Pretend to throw an error message and then have them debug what happened to fix the code.

Be sure the students understand that the position you move to is always relative to the student who is clapping. If you have additional time, choose another student to stand elsewhere in the room and clap. Move around the second student in a square to show that the position is relative to the clapping student and not to a fixed point.

Reflect (2 mins)

What would happen if the student moved while the teacher was moving around her? (The teacher would walk in a different shape depending on where the student was when each `moveXY` started.) **What two new Course 3 concepts do you have to combine to do relative movement?** (Properties and computer arithmetic.) **In CodeCombat, which directions are -x, +x, -y, and +y?** (Left, right, down, and up.)

Coding Time (30-45 mins)

Allow the students to go through the game at their own pace, keeping notes about every level on paper or digital document. We recommend using following format, which you can also print out as templates: Progress Journal [PDF] (http://files.codecombat.com/docs/resources/ProgressJournal.pdf)

```
Level #: _____  Level Name: ____________________________________
Goal: __________________________________________________________
What I did:

What I learned:

What was challenging:
```

Circulate to assist. Draw students' attention to the instructions and tips. If they aren't moving like they expected to, have them drag the time scrubber to the moment where it all went wrong and pause the code, then think about exactly what coordinates are being calculated at that time.

Remind the students to think about the direction you moved when `x` and `y` were added to and subtracted from.

Written Reflection (5 mins)

How would you implement `hero.moveRight()` , where the hero moves 12 meters to the right, using `hero.moveXY()` and relative movement? What about `hero.moveLeft()` , `hero.moveUp()` , and `hero.moveDown()` ?

> For `hero.moveRight()` : `hero.moveXY(hero.pos.x + 12, hero.pos.y)` For `hero.moveLeft()` : `hero.moveXY(hero.pos.x - 12, hero.pos.y)` For `hero.moveUp()` : `hero.moveXY(hero.pos.x, hero.pos.y + 12)` For `hero.moveDown()` : `hero.moveXY(hero.pos.x, hero.pos.y - 12)`

Make up a story: why do you think the yaks are so violent in CodeCombat that they would attack you if you ever got too close to them?

> Probably they have learned to defend themselves against ogre poachers so they have a built-in fight response when they get close to anything with two legs. Before the ogres came, they would come right up to you and ask for food instead, but now they living in paranoia and fear after the ogres started hunting them.

MODULE 19

Time and Health

Summary

Time is a basic input for many programs. For example, programs can be made to perform an action at a certain time. They can also be constructed to execute a statement after enough time has passed.

In this module, students will learn how to respond to time passing with the `hero.now()` function. They will use this function to perform actions at a certain time by using the amount of time that has passed since the level started.

In this module, students will also practice using `hero.health` to determine when to do something. This property allows the students to perform an action when their hero's health reaches a certain threshold.

Transfer Goals

- Write code based on elapsed time by using the `hero.now()` function
- Write code based on thresholds using `hero.health`
- Learn when to change overall strategies in code

Standards

CCSS.Math.Practice.MP1 Make sense of problems and persevere in solving them.
CCSS.Math.Practice.MP2 Reason abstractly and quantitatively.
CCSS.Math.Practice.MP6 Attend to precision. **CCSS.Math.Practice.MP7** Look for and make use of structure.

Instructive Activity: Silent Game (12 mins)

Explain (3 mins)

The students have already learned how to use events to determine when to do things in their programs. For example, they have learned how to make their pets react to events like hearing the hero speak:

```
def speak(event):
    pet.say("Meow!")

pet.on("hear", speak)
```

Additionally, the students have used `while` loops in conjunction with `if` statements to decide when to do one thing or another based on a certain condition. For example, they have written code to attack if there are enemies close by, or to use the cleave if it is ready to be used:

```
while True:
    if hero.isReady("cleave"):
        hero.cleave()
```

Today the students will learn how to perform actions based on time, rather than on events and conditions. One way they can do this is with the function `hero.now()` . `hero.now()` returns the amount of time, in seconds, that has passed since the "Run" button was pressed. Each time the button is pressed, the time starts again from 0 seconds.

The `hero.now()` function can be used as so:

```
if hero.now() < 10:
    enemy = hero.findNearestEnemy()
    if enemy:
        hero.attack(enemy)
```

The code above ensures that in the first 10 seconds of the level, the hero will attack as long as there is an enemy present. By combining `if` and `elif` statements that have conditionals for different time amounts, the students can set different actions to occur at specific times.

In addition to the function `hero.now()`, students will learn about the properties `hero.health` and `hero.maxHealth`. The students can set conditionals using these properties so that certain actions occur when their health reaches a certain amount:

```
healingThreshold = hero.maxHealth / 2
if hero.health < healingThreshold:
    hero.say("Can I get a heal?")
```

The first line of the code segment above uses `hero.maxHealth` to set a threshold at which the hero should perform a certain action. Notice that the variable `healingThreshold` is created in the first then used in the `if` statement just below it. This `if` statement will execute only when the hero's health is below the `healingThreshold`.

Interact (7 mins)

This activity is a modified version of the Quiet Game, in which the entire class tries to be quiet for as long as they can.

Tell the class you're going to write a program to score them on how long they can be quiet. Start with this code on the board:

```
def calculateScore():
    endTime = now()

startTime = now()
students.on("noise", calculateScore)
```

Say that you're going to test the program. Tell the students that you will count down from 3 and they should be silent. Once they are quiet, record the current time, including the seconds on the board as so:

" `startTime` is 10:05:30".

For the first round, the goal is for the students to only be silent for a few seconds. If your students are generally good at being silent, you may want to start making funny faces or make a loud noise to startle them so they are not silent for too long. This also helps to add a fun aspect to the game.

Once the students have made noise, note the current time and say it out loud for the students (i.e. "10:05:35"). Tell them that because they made noise, the `calculateScore` event listener fired for the "noise" event.

Ask the students what the `endTime` should be, and guide them to see that it is the current time, 10:05:35. Note that is not the current time in the present, but the time at which they started making noise.

Get the students to help you figure out how many seconds they were silent for. Guide them to see that you should subtract the two times and then start grading them:

```
def calculateScore():
    endTime = now()
    duration = endTime - startTime
    if duration < 10:
        score = "amateurs"
```

Ask the students for other time thresholds and scores until you have a program like this:

```
def calculateScore():
    endTime = now()
    duration = endTime - startTime
    if duration < 10:
        score = "amateurs"
    elif duration < 20:
        score = "acceptable"
    elif duration < 30:
        score = "good"
    elif duration < 40:
        score = "professionals"
    else:
        score = "robots"
    return score

startTime = now()
students.on("noise", calculateScore)
```

Give the students between 5 and 10 rounds to see how they long they can stay silent for. Once again, you may feel free to make funny faces or startling noises if it is appropriate for your class.

Record the duration of each round. Explain to the class that because you now know the range, you can use computer arithmetic to adjust the program to adapt to the best score.

Adapt the code on the board so that it now appears as follows:

```
maxTime = 55
def calculateScore():
    endTime = now()
    duration = endTime - startTime
    if duration < 1 / 5 * maxTime:
        score = "amateurs"
    elif duration < 2 / 5 * maxTime:
        score = "acceptable"
    elif duration < 3 / 5 * maxTime:
        score = "good"
    elif duration < 4 / 5 * maxTime:
        score = "professionals"
    else:
        score = "robots"
    return score

startTime = now()
students.on("noise", calculateScore)
```

Explain that if the students were silent four-fifths as long as their longest try, they would be "robots". Note that because the `elif` statement is written as `elif duration < 4 / 5 * maxTime`, it does not include `4 / 5 * maxTime`. Similarly, if the students were silent for at least three-fifths the length of their longest try, they would be "professionals", and so on.

Reflect (2 mins)

How do you get a time duration from two absolute times? (Subtract the beginning time from the end time.) **How do you pronounce `if duration < 1 / 5 * maxTime:` ?** ("If the duration is less than one-fifth of the maxTime...")

Coding Time (30-45 mins)

Allow the students to go through the game at their own pace, keeping notes about every level on paper or digital document. We recommend using following format, which you can also print out as templates: Progress Journal [PDF] (http://files.codecombat.com/docs/resources/ProgressJournal.pdf)

```
Level #: _____  Level Name: ____________________________________
Goal: __________________________________________________________
What I did:

What I learned:

What was challenging:
```

Circulate to assist. Draw students' attention to the instructions and tips. Remind them to compare their current `hero.health` to some fraction of `hero.maxHealth`. Also encourage the students to look for common mistakes in their code, such as typos or incorrect relational operators (`<` , `<=` , `>` , and `>=`).

Written Reflection (5 mins)

Apart from choosing when to fight and to get healed, what else do you think you will use `hero.now()` or `hero.health` to do?

> I would use `hero.now()` to run in a square for the first 30 seconds, and then once all the ogres are chasing me I would stop and cleave them all at once. Or I could shield until I was about to die and then cleave everyone.

When you call `hero.now()` , it returns a number. What does that number mean? Why doesn't it return today's real-world date and time?

> The time it returns is the number of seconds since the level started. It is easier to work with than the real time since you don't have to change your code to have a different time threshold every time you run it, and you don't have to keep track of the start time in your code.

MODULE 20

Break and Continue

Summary

Programmers use **break statements** to exit out of a while loop early. `break` statements indicate to break out the loop, and move on to the next line of code. They are used to move on to the rest of the program after a loop, because it is determined that the rest of the loop should not be run.

Continue statements are like break statements, but they skip to the top of the while loop instead of exiting. `continue` statements indicate to stop this loop here and continue at the top with the next iteration. If a loop doesn't need to be finished (because it doesn't need to do anything right now), a `continue` statement is used.

Transfer Goals

- End while loops with break statements
- Skip while loop iterations with continue statements
- Understand when break/continue are cleaner than nested if/else

Standards

CCSS.Math.Practice.MP1 Make sense of problems and persevere in solving them.
CCSS.Math.Practice.MP2 Reason abstractly and quantitatively.
CCSS.Math.Practice.MP6 Attend to precision. **CCSS.Math.Practice.MP7** Look for and make use of structure.

Instructive Activity: Duck Duck Goose (17 mins)

Explain (5 mins)

The students have been using and writing `while True:` loops throughout the game thus far. Because the loops are always true, there is no way to end the loops and they are infinite.

Now the students will learn how to stop running a loop and start doing something else. For example, perhaps they have defeated all of the enemies and want to send their hero home. This can be done with the use of a `break` statement:

```
# this loop stops running if there is not an enemy
while True:
    enemy = hero.findNearestEnemy()
    if enemy:
        hero.attack(enemy)
    else:
        break

hero.say("My job here is done!")
hero.retire()
```

Psuedocode for the code written above is as follows:

```
while True:
    find the nearest enemy
    if there is an enemy:
        attack the enemy (and return to the top of the loop)
    or if there are no enemies:
        break and move out of the loop to the next line of code

say "My job here is done!"
retire and go home
```

Notice that the code above has a conditional within the while loop that determines whether the loop should continue running or not. Without the `break` statement, the loop would continue to run forever, even when there are no enemies present, and the two lines of code below the loop would not be reachable.

Similar to `break` statements are `continue` statements. Instead of breaking out of a loop, `continue` statements are used to stop the current iteration of a loop and continue to the next iteration. Students can use `continue` statements to write a loop that has a lot of code to run if there are enemies, but does nothing if there are no enemies:

```
# this loop continues to run but starts again from the beginning if there is not an enemy
while True:
    enemy = hero.findNearestEnemy()
    if not enemy:
        continue
    # ...
    # ... lots of code here to deal with the enemy
    # ...
```

The code above uses a conditional to determine whether to complete the current iteration of the loop or to stop the current iteration and move on to the next. Notice that because the loop is not broken with a `break` statement, it will run infinitely.

It is important to note that both `continue` and `break` change the flow of control and prevent the next lines of code from being run, at least in that current loop iteration. Thus, when combining them with conditionals, they can help to eliminate the number of conditionals needed.

For example, the code above could be written as so without the use of `continue`:

```
while True:
    if not enemy:
        # do something
    else:
        # do something else
```

For that scenario, `continue` helps to not only allow the hero to do nothing if he does not see an enemy, but also to avoid the extra `else` clause because the code will not be reached in that iteration once `continue` is executed.

Interact (10 mins)

This activity is a modified version of the popular game Duck, Duck, Goose. The goal is to write code that represents the game and also incorporates both `continue` and `break`.

Have the students sit in a circle to prepare for the game. Write the first line of code on the board to get the program started:

`while True:`

Ask for one student to volunteer to be "it". Get the student to slowly walk around a say "duck" a few times before pausing the game. Ask the class to help you write code to simulate what happens in the game when "it" says "duck". Push them to arrive at code that is similar to this:

```
while True:
    it.moveTo(nextStudent)   # note: this first line is optional
    if it.say("duck"):
        continue
```

Ensure that the students understand why `continue` is used here. Because the loop just starts again (i.e. it just moves to the next student then says either "duck" or "goose" again, it is appropriate to stop the current iteration of the loop and begin again at the top for the next move.

Resume the game reminding "it" to move slowly from person to person. Point to the corresponding line of code with each move and statement until it says "goose". Quickly yell "Pause!" when this happens to have the students help you finish writing the code.

Ask the class what happens in the game now. Many students will likely mention that the "goose" now chases "it" around the circle. Be sure to guide the discussion of the two possible outcomes from the chase - either the "goose" catches "it" or he doesn't.

Now ask the students what happens if the "goose" catches "it". They should respond that "it" then resumes from where he left off, repeating the same process as before. Ask them how to represent that as code, and push them to see that since the loop starts again from the top, this could be represented with `continue`:

```
while True:
    it.moveTo(nextStudent)   # note: this first line is optional
    if it.say("duck"):
        continue
    if goose.catch(it):
        continue
```

Then ask them what happens if the "goose" does not catch "it". They should respond that "it" takes the "goose's" spot and the "goose" then becomes "it". You should emphasize that if the "goose" does not catch "it", the current round is over, and the game could even end at that point. Have them help you complete the code so that you have something similar to this:

```
while True:
    it.moveTo(nextStudent) # note: this first line is optional
    if it.say("duck"):
        continue
    if goose.catch(it):
        continue
    else:
        break

# determine whether to play another round
# if you play another round, it = goose
```

Ensure the students understand why `break` is used when "it" is caught. Because there is a new person for it, the round is completed so the loop is broken out of. This allows the code below the `while` loop to be run and for the teacher to decide if another round should be played.

Allow the students to play a few more rounds of the game, ensuring to point to the corresponding line of code with each move. Feel free to pause the game when necessary to explain the flow of control. Feel free to end the game after a few minutes.

Reflect (2 mins)

When does it make sense to use `break` ? (When you want to stop doing a while loop and do something else.) **When does it make sense to use `continue` ?** (When you don't want to have everything nested inside an `else` .)

Coding Time (25-40 mins)

Allow the students to go through the game at their own pace, keeping notes about every level on paper or digital document. We recommend using following format, which you can also print out as templates: Progress Journal [PDF] (http://files.codecombat.com/docs/resources/ProgressJournal.pdf)

```
Level #: _____  Level Name: ____________________________________
Goal: __________________________________________________________
What I did:

What I learned:

What was challenging:
```

Circulate to assist. Draw students' attention to the instructions and tips. If they get stuck, have them drag the timeline scrubber to the point where their code stopped doing what they expected, and have them reconstruct what their code is trying to do at that time.

To help with debugging, this could be a good time to use the Engineering Cycle worksheet again if students haven't tried that recently.

Written Reflection (5 mins)

Select appropriate prompt(s) for the students respond to, referring to their notes.

What's some code that you have been writing in CodeCombat that could be simpler with `break` or `continue` ?

> A lot of times I check whether there is an enemy or an item. If there's not, I could use `continue` to wait until there is instead of using an `else` . Also, if I wanted to break down a strong door and then keep moving afterward, I could use `break` .

Make up a story: your hero seems to have more and more soldiers and peasants on their side. Why? Who are the humans, who is your hero, and why are they on the same team?

> My hero is the one who led the human expedition into these lands ten years ago, since our people were being persecuted in our original country and we wanted freedom to listen to the rhythmic drumming music we like 24/7. But our rhythmic drumming attracted the attention of the ogres, who love to mosh, and our people blame my hero and rely on her for protection and to basically do everything for them, like defending them from ogres and powering their entire coin-collecting economy.

MODULE 21

Review - Multiplayer Arena

Summary

This is a boss level that will require ingenuity and collaboration to solve it. The goal of the level is to defeat the main boss, but students will also have to collect coins, hire mercenaries, and heal their champion.

Have students work in pairs and share their tips with other teams. The students should make observations about the level on scratch paper, and then use them to make a plan.

The arena level is a reward for completing the required work. Students who have fallen behind in the levels or who have not completed their written reflections should use this time to finish. As students turn in their work, they can enter the Cross Bones arena and attempt multiple solutions until time is called.

See the Arena Levels Guide (/teachers/resources/arenas) for more details.

Transfer Goals

- Synthesize all CS3 concepts.

Instructive Activity: Review & Synthesis (10 mins)

Interact (10 mins)

Get the students to help list and define all of the new vocabulary words they've learned thus far. As a class, decide on both a definition and an example. Have students write these on the board and correct each other’s work. Consult the game where there are disputes.

Object - a character or thing can can do actions, `hero`
Function - an action that an object can do, `hero.cleave()`
Argument - additional information for a function, `hero.attack(enemy)`
Loop - code that repeats, `while True:`
Variable - a holder for a value, `enemy = ...`
Conditional - code that checks if, `if hero.isReady()` :
Concatenation - adding two strings together, `"string1" + "string2"`
Arithmetic - using Python to do math, like `2 + 2`
Property - attribute belonging to an object, like `item.pos`
Flags - objects you put down to send input to your program
Return - when a function computes a value and returns it
Boolean - a value that is either true or false
Break - a way to exit a `while` loop
Continue - a way to skip to the top, or next iteration, of a `while` loop

Coding Time (30-45 mins)

Have students who have completed the rest of Course 3 work in pairs and navigate to the last level, **Cross Bones**, and complete it at their own pace.

Note that the player area is in the bottom left, and the tents may be obscured by the status bar. Students can press SUBMIT to see the full screen.

For students having problems, remind them of all the debugging strategies they have learned so far. Tell them to carefully read the instructions and remember the hints. Encourage them to sit and think about how to solve the problem and to write down a plan for solving it before they begin coding.

Students should approach these levels with the habits and mindset of a good programmer and problem solver by doing the following:

- Define the problem
- Break the problem down into parts
- Make a plan on how to solve the problem
- Pay attention to syntax
- Debugging to find the cause of errors
- Ask for hints when needed

Rankings

Once students beat the default computer they will be put in for the class ranking. Red teams only fight against blue teams and there will be top rankings for each. Students will only compete against the computer and other students in your CodeCombat class (not strangers).

Note that the class rankings are plainly visible. If some students are intimidated by competition or being publicly ranked, give them the option of a writing exercise instead:

- Write a walkthrough or guide to your favorite level
- Write a review of the game
- Design a new level

Dividing the Class

Students must choose a team to join: Red or Blue. It is important to divide the class as most students will choose red. It doesn’t matter if the sides are even, but it is important that there ARE players for both sides.

- Divide the class into two randomly by drawing from a deck of cards.
- Students who turn in their work early join the blue team, and latecomers play red.

Refining the Code

Code for Cross Bones can be submitted more than once. Encourage the students to submit code, observe how it fares against their classmates, and then make improvements and resubmit. In addition, students who have finished the code for one team can go on to create code for the other team.

Written Reflection (5 mins)

Write a chronicle of your epic battle from the point of view of either the hero or the boss.

> I am Tharin Thunderfist, the great hero of the battle of Cross Bones. Together with my guardian, Okar Stompfoot, I attacked the ogres and freed the valley from their tyranny. I gathered coins to pay archers and fighters to join the battle. Then I cured Okar when he was injured.

How did you break down the problem? What challenges did you come up against? How did you solve them? How did you work together?

> First we saw that the code already did collecting coins. So we made it go to the tents when we could afford to hire fighters. Then we had to get the potion, but we messed up the code. The teacher helped us fix it. But we still didn’t win, so we asked another team for help and they showed us how to defeat the enemy. We worked well together. It was fun and hard.