the rules

Equipment
Code Monkey Island Gameboard, 12 monkey figurines (three of each color), 54 Guide cards, 16 Fruit cards, 10 Boost in a Bottle cards.

The Goal
Get all three of your monkeys around the board and into the Banana Grove before anyone else can!

Setup
Select three monkeys of the same color. Place two in the corresponding start circle, and one on the first tile leading from the start circle.

Shuffle each deck, place them down on the marked spots on the board, then deal 3 Guide Cards to each player, and choose a player to start. Monkeys move counter-clockwise around the board.

gameplay

How to earn and use moves
Guide Cards allow your monkeys to move around the board, but only when the conditions are just right! Look at ALL of the monkeys on the board (even other players' monkeys) to see if your Guide Cards will earn you moves. Remember that only your monkeys can move forward on your turn.

On each turn
Select one Guide Card from your hand, read it out loud, move ONE of your monkeys, then discard that card and draw a new one.

Moving backwards

If one of your monkeys is moved backwards past the exit of your Start Circle, that monkey can then enter the Banana Grove without circling the board.

Knocking back

If you land on or are moved back to a tile with another monkey already on it, that monkey is knocked backwards to the nearest open quicksand tile.

Quicksand

If one of your monkeys lands on or is moved back to a quicksand tile, it becomes stuck and can't be moved for one round! Lay the monkey on its side while it's stuck.

Landing on Fruit

If one of your monkeys lands on a Fruit tile, draw a Fruit card, read it out loud, play it immediately, and then discard it. The effects of the card only apply to the monkey that landed on the tile. Fruits will often give your monkey a significant boost, but beware of the rotten fruits that make your monkey sick!

If you are moved to a Fruit tile on another player's turn, do not draw a Fruit card.

Landing on Bottle Tiles

If a monkeys lands on a Bottle tile, draw a Boost in a Bottle card and set it face up at the edge of the board. The next monkey (of any color) that lands on that tile will receive a boost equal to the number "in the bottle".

Once a monkey has used the boost, put the card at the bottom of the Boost in a Bottle deck.

If you are moved to a Bottle tile on another player's turn, do not draw a Boost in a Bottle card or use a boost that is already in the bottle.

Entering the Banana Grove

Once one of your monkeys makes it all the way around the island and back near its Start Circle, it can then be moved up it's colored path into the Banana Grove. You may only enter the Banana Grove through your colored path. When one of your monkeys enters the Banana Grove, you earn 10 bonus moves that you must use immediately on another one of your monkeys.

the guide cards

Count Cards

Some Count Cards will ask you to move a certain number of spaces for each time a condition is true.

For instance, a Count Card might say "For each monkey on a rock, move forward 3 spaces." If you look at the board and see that there are 3 monkeys sitting on rock tiles, then you get 9 moves!

Check Cards

Some Check Cards allow you to move however many spaces you want within a range when a condition is true.

For instance, a Check Card might say "If a monkey is on a rock, move between 1 and 8 spaces forward." If you look at the board and see that there's at least one monkey on a rock, then you get to move UP TO 8 spaces forward! So if your monkey is only 5 spaces away from a fruit tile, then you could choose to move 5 spaces forward out of the possible 8.

Bug Cards

Bug cards can send monkeys backwards - even your own! Play them carefully!

the explorers guide

The Explorer's Guide helps players ages 10+ learn how the game mechanics of Code Monkey Island can be used to write a real computer program. It introduces players to basic programming syntax and application, and utilizes "pseudo code" (code written in plain English) to make these sometimes-complex concepts as understandable as possible.

Parents: we highly suggest that you read the Explorer's Guide with your child. We also recommend reading through the Explorer's Guide a few times on your own first, so that you can more easily help your child along if they get stuck at any point.

Once you and your child have become comfortable with the concepts taught in this Guide, you can visit the following websites to continue your programming education for free!

1.) http://codecampus.com

2.) http://codecademy.com

3.) http://scratch.mit.edu

chapter 1
what is a program?

Imagine that your friend has asked you to write down a set of instructions for making a PB&J sandwich. Your instructions might look like this:

Step 1: If you have all the right ingredients (peanut butter, jam, and bread), begin making the sandwich. Otherwise, go buy the right ingredients!

Step 2: Take out two slices of bread.

Step 3: For each ingredient, use a knife to spread that ingredient on its own slice of bread

Step 4: Put one slice of bread on top of the other

Step 5: Take one bite at a time until the sandwich is gone!

You can think of a program as a set of instructions like the one above. Computers use programs to make things happen, just like your friend will use your instructions to make a delicious sandwich!

chapter 2 variables

Now that we know that a program is just a set of instructions, let's start writing a program that asks the computer to make us a virtual PB&J sandwich.

Before we can do that, though, we need our program to know that we have all the right ingedients. The first thing we need to learn about are variables.

What are variables?

When you are playing Code Monkey Island and your monkey lands on a Bottle tile, you get to draw a card with a boost on it, like this one:

Then, you put the boost inside the bottle for another monkey to find. The bottle stores the boost for later! When we write programs, we use things called **variables** to store values (like numbers or words) so that we can use those values later. The bottle is a great example of a variable. It stores the value of the boost so that another monkey can use that boost later.

If we wanted to tell another player how big of a boost the bottle is storing, we might say:

> **"The boost in the bottle is worth 12 spaces"**

But how would we tell that to a computer? Since computers do not speak the same language that we do, we will use a program to help us translate.

How to create a variable

Remember that the bottle is our variable, and that it is storing the value of the boost. There are three steps to creating a variable in a program.

First, we need to give a name to our variable so that we can easily remember what it is storing. There are only two rules when naming a variable:

1.) A variable name must begin with a letter

2.) A variable name cannot contain spaces

Other than these two rules, we can name our variable whatever we like. Since our bottle is the variable, and what it is storing is the boost, let's call our variable bottle_boost. Our new name begins with a letter. It does not contain spaces (we can replace spaces with an underscore). It helps us remember that our variable is the bottle, and that the value it will store is the boost. Perfect!

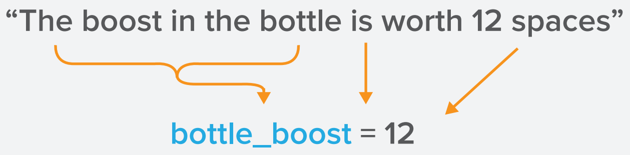
Second, we will use an equals sign to tell the program that bottle_boost is going to store something inside of itself.

```
bottle_boost =
```

Third, we will tell it the value of the boost. We drew a card with a value of 12, so we will tell our variable to hold on to that value by putting 12 on the other side of the equals sign:

```
bottle_boost = 12
```

Now our program understands that our bottle is storing a value of 12. Compare this with the sentence we used to tell the other player how many spaces were in the bottle:



“The boost in the bottle is worth 12 spaces”

```
bottle_boost = 12
```

They are saying the exact same thing, just differently!

Variables can store a lot of things, like equations. For example, we could store this equation in a variable:

```
total_moves = 5 - 3
```

The program can do math, so it understands that the total value stored inside of **total_moves** is equal to 2.

We can also store *other variables* inside of our variable. For example, let’s say that we are playing a round of Code Monkey Island, and draw a Check card worth 6 moves:

```
check_card = 6
```

Now, let’s add that to our boost from earlier, and store the total amount of moves we can earn in **total_moves**:

```
total_moves = check_card + bottle_boost
```

Since we know that **bottle_boost** is equal to 12, and **check_card** is equal to 6, we can add them together and store their total value of 18 inside of **total_moves**.

Finally, we can also store words inside of variables. Let's say that our friend Ben wants to store the name of the city he lives in inside of a variable. Ben is going to use what we just learned to create a name for his variable and store a value inside of it.

```
my_city = "New York City"
```

Programs have a much smaller vocabulary than us, and they do not understand a lot of the words we use. We put regular words inside of quotation marks to tell the program that it does not need to understand those words, it just has to store them so we can use them later.

Now that Ben has created a variable, he could write a program about where he lives, and use his **my_city** variable instead of the full name of his city:

```
Hi! My name is Ben. I live in my_city, and I love it there!
my_city is one of the biggest cities in the world.
```

Now, when Ben asks the program to read his sentence back to him, it will look like this:

```
Hi! My name is Ben. I live in New York City, and I love it
there! New York City is one of the biggest cities in the
world.
```

Your turn!

Create one varable to store your name, and another variable to store how old you are. Then, write in the correct values in each variable after the equals sign. You can name your variables whatever you like, but try to give them a name that will help you remember what they are storing.

______________________ = ______________________

______________________ = ______________________

Now that you have created your variables, insert the names of your variables into the paragraph below:

Hello there! My name is ____________________________.
name variable

I am ____________________________ years old, and I am
age variable

already learning how to use variables! Isn't that cool?

Now, pretend that you are the program! Replace the names of the variables above with the values they store to translate the message:

Hello there! My name is ____________________________.

I am ____________________________ years old, and I am

already learning how to use variables! Isn't that cool?

the story so far...

Before we go on, let's take a moment to review everything we have learned so far:

We learned what a program is

A program is a set of instructions we write for computers. The computer reads those instructions, and then does what the instructions tell it to do!

We learned what a variable is

A variable is like a bottle - it's an object that stores something inside of itself so that it can be used later. In programs, variables store values like numbers or words.

We learned how to create a variable

Variables have three parts: a name, an equals sign, and the value being stored. Variable names must start with a letter and cannot contain any spaces, and they can store either numbers or words.

```
my_age = 12

color = “blue”

dads_age = my_age + 30
```

We're making great progress! Keep going and we will have a program that makes us sandwiches in no time.

chapter 3

true or false?

Remember: before making our sandwich, we need to tell our program that we have all the right ingredients. Now that we know about variables, we are one step closer.

A program has a much smaller vocabulary than you and me. But it does know two special words - **true** and **false** - that can help us tell it all sorts of things.

Read these statements, then circle the correct response - go to your kitchen and take a look around if you are not sure:

I have jam.	TRUE or FALSE
I have peanut butter.	TRUE or FALSE
I have bread.	TRUE or FALSE

Great job! Now that you have selected the correct responses to those statements, we can use variables to store your responses. For example, here is how Ben responded to these statements:

I have milk.	[✓ TRUE] or FALSE
I have cookies.	TRUE or [✓ FALSE]

And here is how he would store those responses: inside of variables:

```
has_milk = true

has_cookies = false
```

Remember from the last chapter when Ben was storing the name of his city inside of a variable, and he had to use quotation marks around the words?

```
my_city = “New York City”
```

He had to do this because the program does not know what words like New York City, banana, monkeys, etc. mean. The quotation marks tell the program that it does not have to understand these words, it just has to store them inside of the variable so that we can use them later.

true and **false** are special words, though. Since the program *does* know what these words mean, Ben will not put them in quotation marks. If he does put them in quotation marks, the program will think they are regular words and ignore them!

Your turn!

Now that Ben has shown us how to store the values of his responses inside of variables with true and false, let’s do the same for your responses from before.

Look back at your responses on the previous page, and store them inside of the variables below without using quotation marks:

```
has_jam = ____________________
has_peanut_butter = ____________________
has_bread = ____________________
```

Great work! Now our program knows what ingredients we have and don’t have.

chapter 4
conditional statements

Let's say that Ben wants to go play outside. When he goes to ask his parents for permission, they say, "If your homework is done, go play outside!" Here, Ben's parents have set a condition: Ben's homework must be done before he can go outside.

CHECK
If any monkey is on a tree,
move between 3 and 8 spaces forward.

Here's another scenario: a Check Card from Code Monkey Island. This Check Card says "If any monkey is on a tree, move between 3 and 8 spaces forward." This means that if there is at least one monkey on a tree tile, you will be able to earn moves from this card. But if there are no monkeys on tree tiles, then you will not be able earn moves from this card.

Both of these situations are examples of **conditional statements**, which are used to help people and programs make decisions based on certain conditions. As you can see, a basic conditional statement has two parts: one part checks if the condition is met, and the other part is an action that will be taken only if it is met. Let's learn how conditional statements can help us with our sandwich program.

How to write a conditional statement

Programs use conditional statements the same way that we do. For example, here is how we would write out Ben's parents' conditional statement in a program:

```
if homework_done == true
    say "Go play outside"
```

There are a lot of new things happening here! Let's go through this conditional statement step by step.

First, we have a special word if that the program understands, just like the words true and false. This special word tells the program that we are starting to write a conditional statement.

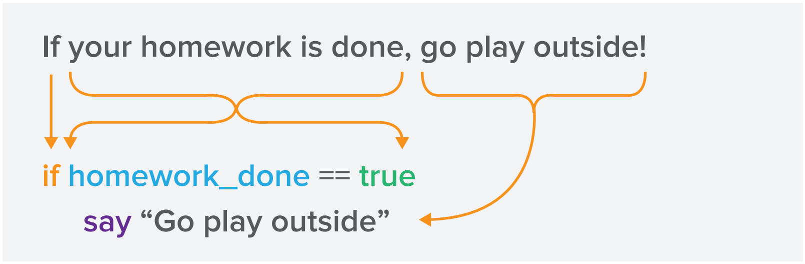
Next, we created a variable called homework_done that stores a value of either true or false.

Our program does not know what is stored inside of a variable until we tell it to check. When we want our program to check what's already inside a variable, we use two equals signs right next to each other, which the program translates as *"is equal to"*. The program checks to see whether the value inside of homework_done *is equal to* true using two equals signs.

If (and only if) the variable homework_done is equal to true, then the program will perform the action written on the next line. It uses another special word say, which tells the program to say something to us.

But if homework_done is equal to false, then the program will not say or do anything underneath the first line.

That is all there is to it! Let's compare our conditional statement with what Ben's parents told him:


If your homework is done, go play outside!

```
if homework_done == true
    say "Go play outside"
```

Your turn!

Translate the following sentence into a conditional statement that a program could understand (look at the example above if you get stuck):

If it is raining, bring an umbrella.

Else / Otherwise

A conditional statement can include an extra part that tells the program what to do if the condition is not met. For example, Ben's parents might say, "If your homework is done, go play outside! Otherwise, go read a book."

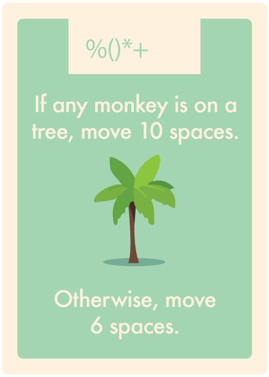
%()*+
If none of your monkeys are in the banana grove, move 12 spaces.
Otherwise, move 5 spaces.

As another example, think about this Check Card. It says, "If none of your monkeys are in the banana grove, move 12 spaces. Otherwise, move 5 spaces." Here, the condition is that none of your monkeys can be in the banana grove. If that's true, then you get to move 12 spaces. But if it's not true - for instance if you have one or more monkeys in the banana grove - then you will perform the second action, which gives you 5 moves.

This works in programs, too. Programs use the word else instead of "otherwise" to perform an alternative action when the first condition is not met, like this:

```
if homework_done == true
    say "Go play outside"
else
    say "Go read a book"
```

Your turn!


%()*+
If any monkey is on a tree, move 10 spaces.
Otherwise, move 6 spaces.

To practice what we learned in this chapter, let's translate this Check Card into a conditional statement that a program could understand. Fill in the missing words to complete the conditional statement! Then, choose a new Check card and translate it into a conditional statement on a new sheet of paper.

```
__________ monkey_on_tree == ________________
    say __________________________

__________

    say "move 6 spaces"
```

the story so far...

Let's review what we learned in the last two chapters:

We learned about the words true and false

We make decisions all the time about whether things are true or false. We can have programs remember our decisions and use them later by storing the words true and false inside of variables.

```
programming_is_fun = true
```

We learned about conditional statements

Conditional statements help programs make decisions by asking them to check if a condition is met before performing an action. We use two equals signs to check what is stored inside of a variable.

```
if raining == true
    say "Better grab an umbrella"

if my_city == "New York City"
    say "You should visit the Statue of Liberty!"
```

When we want the program to do some other action if the condition is not met, we use the word else:

```
if raining == true
    say "Better grab an umbrella"
else
    say "Leave the umbrella at home"
```

chapter 5

AND, OR, and NOT

We have learned about a few special words that programs can understand and use, like true, false, if, and print. Now, let's learn about AND, OR, and NOT. These words are called **logical operators**, because programs use these words to compare the trueness or falseness of two or more statements at once.

Let's say that Ben is making a shopping list:

Buy broccoli AND carrots
Buy sugar OR honey
Do NOT buy peanuts (dad is allergic!)
Buy milk AND eggs

You can think about the items on Ben's list as variables. Since he does not have any of the items on his list in his kitchen, here is what his variables would look like:

```
got_broccoli = false    got_peanuts = false
got_carrots = false     got_milk = false
got_sugar = false       got_eggs = false
got_honey = false
```

When Ben goes to the grocery store, he accidentally forgets his list at home. He does his best to remember what was on it, and here is what he ends up buying:

Broccoli, Carrots, Honey, and **Milk.**

Once Ben goes home, he decides to write some conditional statements to make sure he got everything.

The first sentence on his list is, “Buy broccoli AND carrots”. The word **AND** means that both parts of the sentence have to be **true**. Ben’s first conditional statement looks like this:

```
if got_broccoli == true AND got_carrots == true
    say “Great! I remembered the vegetables.”
else
    say “Uh oh! Have to go back to the store!”
```

Remember that we use two equals signs to tell our program to check what is stored inside of a variable. It does not know what is inside of the variable until we tell it to check! Since Ben got both broccoli and carrots, both of those variables become **true**, and he does not have to go back to the store.

The second sentence on his list is “Buy sugar OR honey”. The word **OR** means that at least one part of the sentence has to be **true**, even if the other part is **false**. Ben’s second conditional statement looks like this:

```
if got_sugar == true OR got_honey == true
    say “Great! I remembered the sweets.”
else
    say“Uh oh! Have to go back to the store!”
```

Since Ben remembered to buy honey, and since his list said he could buy either honey or sugar, he does not have to go back to the store.

The third sentence is “Do NOT buy peanuts.” When the word **NOT** is used in a program, it flips the value in front of it. So **NOT got_honey** == **true** would be read by the program as **got_honey** == **false**.

Now, here is Ben’s conditional statement using **NOT**:

```
if NOT got_peanuts == true
    say “Great! I remembered not to buy peanuts.”
else
    say “Uh oh! Have to go return the peanuts!”
```

Your turn!

Ben is tired from all this shopping and writing! Write the final conditional statement for the milk and eggs, then circle the statement that the program will **say** based on what Ben bought.

Hint: remember that Ben’s list said, “Buy milk AND eggs”. Since he only bought milk, his variables are now **got_milk** = **true** and **got_eggs** = **false**.

____

____

____

____

____

chapter 6

loops

Take a look at this Count card. It says, “For each monkey on a rock, move 3 spaces.” Let’s say that three monkeys are sitting on rock tiles. To help us calculate how many moves we get from this Count card, let’s imagine that we put all three of these monkeys into a list, like this:

COUNT
For each monkey on a rock,
move 3 spaces.

```
monkey_list:
- Monkey 1
- Monkey 2
- Monkey 3
```

Now, let’s rewrite our Count card with our new list:

```
For each monkey in monkey_list, move 3 spaces.
```

Our next step is to move our monkey. So, we will read through each of the items in monkey_list one at a time, and move three spaces for each one, like this:

```
monkey_list:
- Monkey 1
    Move 3 spaces
- Monkey 2
    Move 3 spaces
- Monkey 3
    Move 3 spaces
```

Did you notice how we repeated the same action for each item on the list, and then stopped when we ran out of items? This is an example of a **loop**.

What is a loop?

A loop is a set of instructions that is repeated each time a condition is met, or until a condition is met. Once the condition is no longer met, the loop ends. It is important to give our loops a way to end, otherwise they would go on forever.

We use loops to save ourselves a lot of time when writing programs. As you read on in this chapter, you will see how!

Creating a foreach loop

Creating a loop is easy. Let's use the Count card example to learn how to make our first loop.

Remember that our first step was to create a list to store all of the monkeys on rocks inside of. A list is one way that a program can know when to end a loop, because the program knows how big the list is. *The loop repeats its set of instructions for each item in the list.*

To create a list, first we will make a variable. Then, we will set it equal to all of the items in our list, separated by commas, like this:

```
monkey_list = Monkey 1, Monkey 2, Monkey 3
```

Now, our loop can look through this list. It knows that there are three items in our list, which means that it has to repeat itself three times before it ends.

Our second step is to write the loop itself. Our Count card is an example of a foreach loop. A foreach loop is a type of loop that repeats itself *each time* a condition is met. Let's take the sentence we rewrote earlier:

For each monkey in monkey_list, move 3 spaces.

and rewrite it again so that a program can understand it:

```
monkey_list = Monkey 1, Monkey 2, Monkey 3

foreach monkey in monkey_list
    move monkey 3 spaces forward
```

First, we have the special word foreach that tells our program that we are starting to write a foreach loop.

Next, we tell our loop that it will be looking through all of the items in the list monkey_list.

Finally, we tell the loop what it has to do every time it moves forward in the list. Our loop says that the program has to move our monkey 3 spaces forward each time it goes through the list. Since there are three items in our list, it will perform the action three times and then stop.

Your turn!

COUNT
For each monkey on a tree,
move 3 spaces.

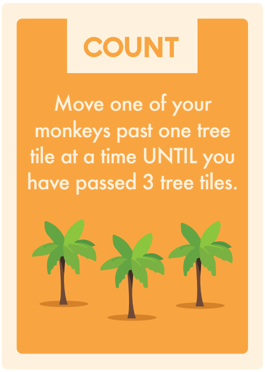
Turn this Count card into a foreach loop by creating a list and a loop. It says, "For each monkey on a tree, move 3 spaces." For this exercise, let's pretend that there are three monkeys on tree tiles. Write your loops in the box on the next page. Look at the example above for hints!

```
monkey_list = __________, __________, __________,

____________ monkey in _______________
    move forward 3 spaces
```

Until loop

We mentioned earlier that there are a few different types of loops. Besides the **foreach** loop, we will also need to learn about the **until** loop to make our sandwich program.


COUNT
Move one of your monkeys past one tree tile at a time UNTIL you have passed 3 tree tiles.

Take a look at this count card. It says, “Move one of your monkeys past one tree tile at a time UNTIL you have passed 3 tree tiles.” We can turn this into a loop, just like we did with our other Count card!

Just like before, we must first create something that tells the loop when to stop. Instead of a list, this time we will use a **counter**. A counter is a regular variable that keeps count of how many times the loop has repeated itself. A counter is set to store a value of 0, and then after every time the loop repeats itself, it adds 1 to itself. Then, we can tell the loop that once the counter reaches a certain number, it should stop. Since the card says that we must pass three tree tiles before we can stop, let’s first create a variable and store a 0 inside of it:

```
tree_tiles_passed = 0
```

Now, let's create our until loop:

```
tree_tiles_passed = 0

until tree_tiles_passed == 3
    move monkey past a tree tile
    tree_tiles_passed = tree_tiles_passed + 1
```

First, we have our counter tree_tiles_passed. Before we start the loop, we set the counter equal to 0.

Then, we start our loop. The first line of the loop basically says, "Until tree_tiles_passed is equal to 3, do this stuff below." We use the special word until to tell our program that we are starting an until loop, and then we set a condition: our counter *has to be* storing a value of 3 before the loop can end.

Next, we tell the loop what actions we want it to perform each time it repeats. The first action is "move monkey past a tree tile". Once the loop has done that, the next action we ask it to perform is to add 1 to tree_tiles_passed.

Remember from Chapter 2 how variables can store other variables? Here, we are asking tree_tiles_passed to store its own value plus one. Since its value is 0 before the loop starts, we can actually think of it like:

```
tree_tiles_passed = 0 + 1
```

Now, once the loop restarts, the value stored inside of tree_tiles_passed is equal to 1. It will then move the monkey past another tree tile, and tree_tiles_passed will go up by one again, so that it is now equal to 2. After the third time that the loop repeats and the monkey is moved past the third tree tile, tree_tiles_passed will be equal to 3. Since the loop was waiting until tree_tiles_passed was equal to 3, it will now stop.

Your turn!

Let's say that we draw a Count card. It says "Move one of your monkeys past one vine tile at a time UNTIL you have passed 3 vine tiles." Let's turn this card into a loop! First, let's create our counter. Create a variable called vine_tiles_passed and store a 0 inside of it:

```
______________________ = ____________
```

Next, let's create the top line of our loop. Remember that we have to pass three vine tiles before the loop ends:

```
___________ vine_tiles_passed == ___________
```

Finally, let's add the actions that the loop must repeat each time it starts. The first action has been provided. Now, make sure the counter goes up by 1 each time the loop repeats!

```
    move monkey past a tree tile
    vine_tiles_passed = __________________ + 1
```

the story so far...

Here is what we have learned in the past two chapters:

We learned about AND, OR, and NOT

These words compare the trueness or falseness of two statements at once, so that we can decide whether both of them together are either true or false. When using AND, both statements must be true for it all to be true. When using OR, only one of the statements has to be true for it all to be true. The word NOT flips the trueness or falseness of the whole statement.

We learned about loops

Loops allow us to repeat the same action every single time a condition is met, or until a condition is met. We have to tell a loop how many times it repeats, or else it will go on forever. To help our loops know when they have to end, we can use lists and counters.

The two types of loops that we learned about are foreach loops and until loops. foreach loops repeat a set of actions each time a condition is met. Typically, we use lists with foreach loops.

until loops repeat a set of actions until a certain condition is met. Typically, we set a counter equal to 0, and then increase its value by 1 each time the loop repeats itself.

Phew - now we are ready to make our sandwich program!

chapter 7

making our sandwich

Great work! In the past 6 chapters, we have learned everything we need to know to write a program that makes us tasty, virtual sandwiches. Before going on, though, we recommend going back and re-reading the entire Explorer's Guide to make sure you understand everything. If you are ready to go, let's get started!

Remember the instructions we wrote in Chapter 1 for our friend to help them make a peanut butter and jelly sandwich?

Step 1: If you have all the right ingredients (peanut butter, jam, and bread), begin making the sandwich. Otherwise, go buy the right ingredients!

Step 2: Take out two slices of bread.

Step 3: For each ingredient, use a knife to spread that ingredient on its own slice of bread

Step 4: Put one slice of bread on top of the other

Step 5: Take one bite at a time until the sandwich is gone!

Let's turn these instructions into a program using everything we have learned in the Explorer's Guide.

First, we need to check for all the right ingredients. In Chapter 3, we checked our kitchens for all of the right ingredients. We created variables for each ingredient -

jam, peanut butter, and bread - and then put **true** or **false** inside of each one depending on whether we had the ingredients or not.

In the spaces below, recreate those variables and store the correct value inside of them. If you need a hint, go back and re-read Chapter 3!

```
has_jam = ____________________

____________________ = ____________________

____________________ = ____________________
```

Great job! Now that we have set our ingredient variables, let's write some conditional statements using the word **AND** so that our program can check to see if we have *all* of the right ingredients or not. If we have all of the right ingredients, we can keep making our sandwich! But if not, we will have to go to the store.

```
_____ has_jam == true AND __________ == __________

AND __________ == __________

    say "We have all the right ingredients!"

    say "Take out two slices of bread."

_____

    say "Uh oh! We have to go to the store."
```

Next, let's create a list to contain our jam and our peanut butter (but not our bread!):

```
ingredients = has_jam , ______________________ ,
```

Last step! Now that we have our list, let's create a foreach loop that will make our program spread each of these ingredients on a piece of bread:

```
_______________ ingredient in _______________
    spread ingredient on bread
    say “Put one piece of bread on top of the other”
```

Wonderful! Now that our sandwich is complete, let's take a big bite of...wait...where is our sandwich?

Uh oh. Since our program made us a virtual PB&J, we won't really be able to eat it. But maybe our program can!

Let's write one last step in our program that tells the program how to eat the sandwich it just made. Our program can finish a sandwich in about 10 bites, so let's create a counter that will start counting up to 10 from 0:

```
bites = ______________
```

Next, let's setup our until loop:

```
until ____________________ == 10
    take another bite
    bites = _____________________ + _______
```

Well done! We have successfully written a program that creates (and eats) a PB&J sandwich! Give yourself a pat on the back. There is a completed version of the program you wrote on the next page for comparison.

```
has_jam = true
has_peanut_butter = true
has_bread = true

if has_jam == true AND has_peanut_butter == true
AND has_bread == true
    say “We have all the right ingredients!”
    say “Take out two slices of bread.”
else
    say “Uh oh! We have to go to the store.”

ingredients = has_jam, has_peanut_butter

foreach ingredient in ingredients
    spread ingredient on bread
    say “Put one piece of bread on top of the other”

bites = 0

until bites == 10
    take another bite
    bites = bites + 1
```

If you want to keep practicing what you have learned, try to write a program that will make you another kind of sandwich, like a turkey sandwich or a ham and cheese sandwich. Use ingredients like lettuce, tomato, mustard, pickles, or anything else you like. Enjoy!