

Table of Contents

[This is CS50.](#)

[Algorithms](#)

[About CS50](#)

[Problem Set 0](#)

Week 0

This is CS50.

- David tells his own story of shopping CS50 long ago during his sophomore fall and taking it pass/fail, only find out how empowering the field of Computer Science was and how it intersects with the humanities, arts, and other sciences.
- 73% of CS50 students have never taken a CS course before, so rest assured that you are not alone, especially if you have no prior experience in CS.
- "what ultimately matters in this course is not so much where you end up relative to your classmates but where you, in **Week 11**, end up relative to yourself in **Week 0**"
- At the end of the day, computer science is about problem solving: taking some inputs (the problem we might be trying to solve) and creating some outputs (perhaps a solution to the problem).
- Between those inputs and outputs is what we might call a "black box," something that solves the problem for us even if we might not know or care *how* it solves them.
- The inputs and outputs for a computer, for example, are just electricity (from a plug or battery) and changes on its screen.
- But let's think about how we might represent information. If David wanted to count the number of students, he might use his fingers (and quickly run out), or use tally marks:



- This is called **unary**, where there is one symbol and you simply add another one for each thing you're counting.

- Computers use a similar system called **binary**, where there are two symbols, 0 and 1. Humans typically use the **decimal** system, with the digits 0-9.
- In decimal, 123 is one hundred and twenty-three. And if we think way back to grade school, that's because each column has a place value, and we add them up like so:

100	10	1
1	2	3
100 x 1	10 x 2	1 x 3

- Binary represents numbers in the same pattern, but using powers of 2 instead of powers of 10 that decimal uses. The first row shows the value of each column, like the 100, 10, and 1 above, and the second row is our current binary number.

4	2	1
0	0	0

- To represent a 1, we simply place a 1 in the ones column:

4	2	1
0	0	1
1 x 1		

- And a 2 like so:

4	2	1
0	1	0
2 x 1		

- And a 3 by combining the previous two steps:

4	2	1
0	1	1
2 x 1		1 x 1

- We can continue this pattern:

4	2	1
---	---	---

1	0	0
---	---	---

4 x 1

4	2	1
---	---	---

1	0	1
---	---	---

4 x 1

1 x 1

4	2	1
---	---	---

1	1	0
---	---	---

4 x 1

2 x 1

4	2	1
---	---	---

1	1	1
---	---	---

4 x 1

2 x 1

1 x 1

- But once we have used up all the places, we need more **bits**, or binary digit, which stores a 0 or 1. With additional digits, we can represent larger numbers.
- So our computer has many many switches (called transistors) that can be turned on or off given electricity, and a switch that is on will represent a 1 and a switch that is off will represent a 0.
- Eight bits, by the way, is called a **byte**.
- So far we've represented numbers, but we also know that computers can represent letters of the alphabet. Some years ago the world decided on a standard mapping of numbers to letters, called **ASCII** (<http://en.wikipedia.org/wiki/ASCII>).
- In ASCII, the letter A is the decimal number 65, B 66, C 67, and so on. And lowercase letters are another set of numbers. Numbers, too, that you type, would be represented as different numbers according to the table.
- We can also represent colors using a system called **RGB** (https://en.wikipedia.org/wiki/RGB_color_model), where three values that each represent the amount of red, green, and blue, mixing together to represent some color.
- And computer screens, if you lean in really close, are made up of lots and lots of dots, or **pixels**, each with some RGB value that together form a picture or text.

- And with lots of images, one after another, we get videos.
 - Now that we can represent inputs and outputs, we can start solving problems.
-

Algorithms

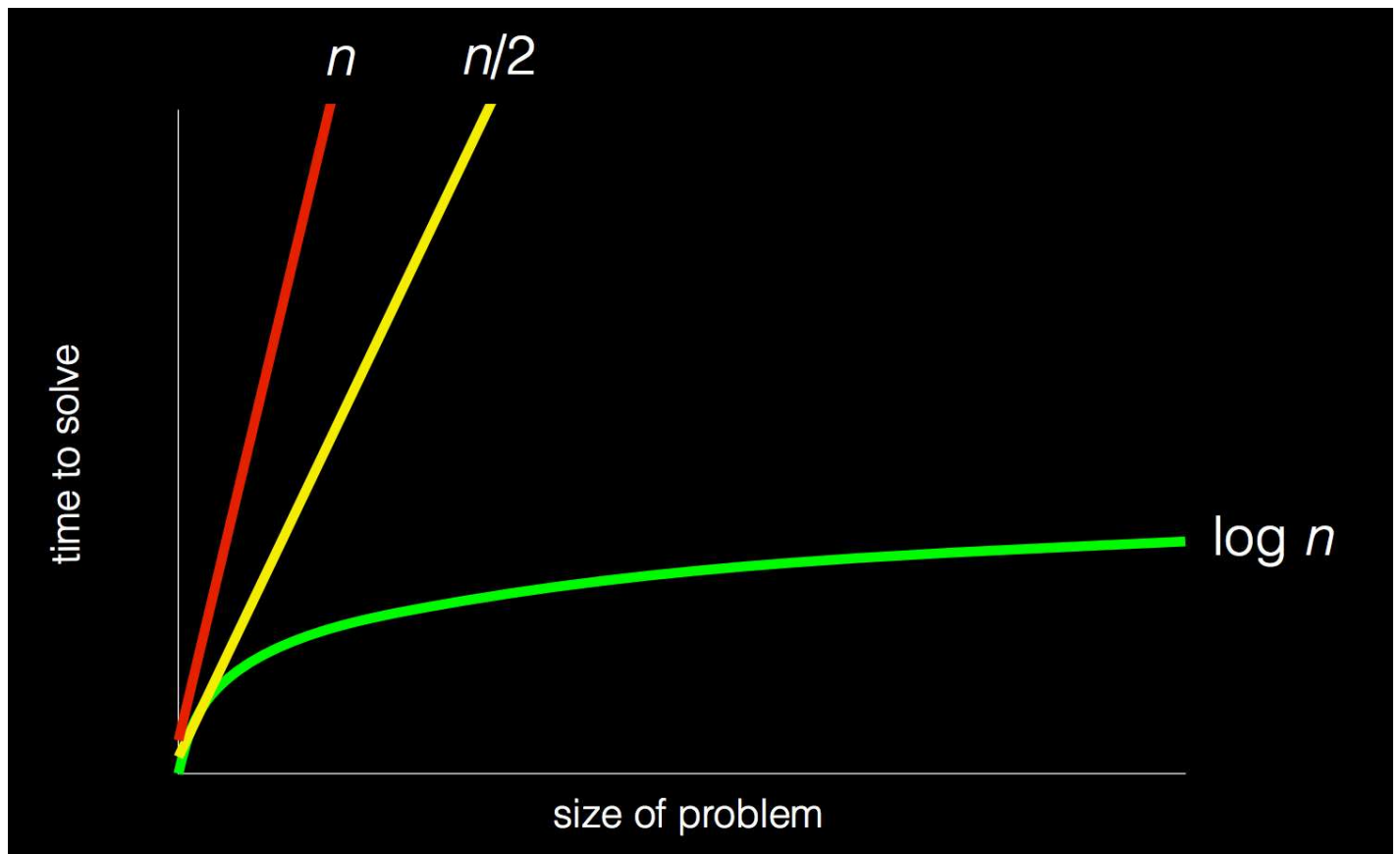
- **Algorithms** are sets of instructions to solve particular problems.
- If David wanted to take attendance (for real this time), he could start counting people one at a time. He could count two people at once, and that would be a little faster.
- But we can use everyone in the room to help, by following these steps:

```
0  Stand up and think of the number 1
1  Pair off with someone standing.
   Add your numbers together.
2  One of you should sit down.
   If you're still standing, go back to step 1.
```

- At the end, our count from this was 546, but the TFs counted 820 students in Sanders. So perhaps there were bugs with the execution of this algorithm!
- Let's move on to another problem. Say we have a large phone book, with thousands of pages. We might be looking for someone, say, Mike Smith. We could open the book to the first page, and the second, and the third, and continue looking like this until we find him. This algorithm is correct, since we will eventually find him, but it's not very fast.
- We can flip two pages at a time, and it is twice as fast as the previous algorithm, but it's possible that we might miss him if he is on an odd page.
- We can go straight to the middle, and find ourselves in the M section, and know that Mike Smith is in the right half of the book, and throw the left half away. We can repeat this again and again, and eventually find one page. With 1000 pages, it would only take about 10 steps of division to reach that one page.
- These are the steps we might follow:

1	0	pick up phone book
2	1	open to middle of phone book
3	2	look at names
4	3	if Smith is among names
5	4	call Mike
6	5	else if Smith is earlier in book
7	6	open to middle of left half of book
8	7	go back to step 2
9	8	else if "Smith" is later in book
10	9	open to middle of right half of book
11	10	go back to step 2
12	11	else
13	12	quit

- Some of these lines are actions we might take, like `pick up` or `open to` or `look at` or `call`. We'll call these functions.
- `if`, `else if`, and `else` are leading to branches, or decision points, based on whether or not the expression, like `Smith is among names`, is true.
- These expressions are called Boolean expressions (named after someone with the last name Bool), and can either be true or false.
- We also have lines like `go back to step 2` that induce a loop, where there is a cycle that does something over and over again.
- So we can also look at how efficient this algorithm is. The x-axis is the size of the problem, and the y-axis is the time to solve. With `n` as the size of the problem, the red line represents the first algorithm in which time to solve increases with the size of the problem:

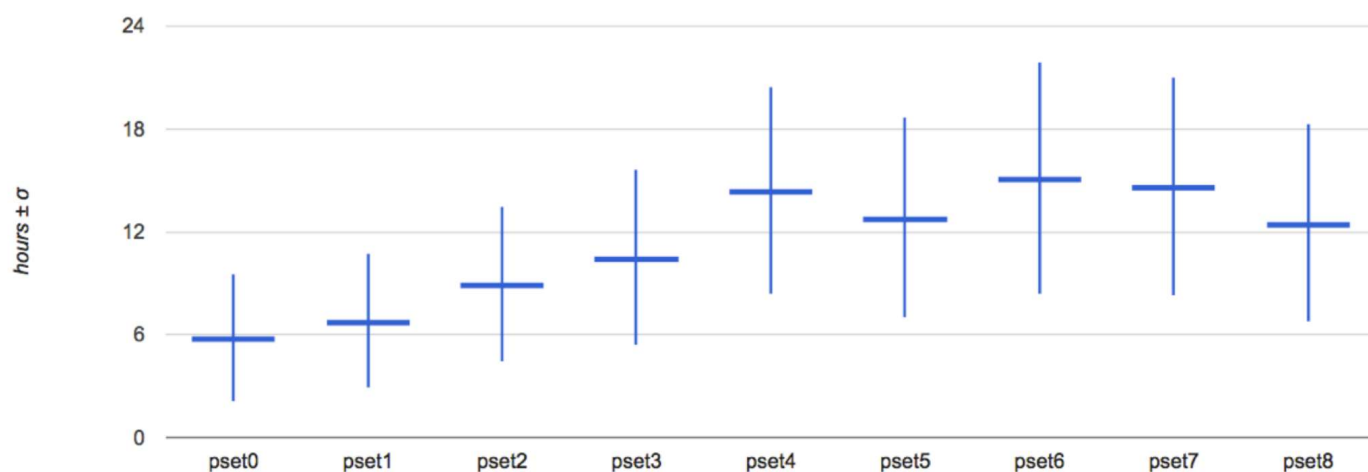


- The yellow line is the second algorithm, which, though twice as fast, still increases linearly with the size of the problem.
- The green line will have a **logarithmic slope** that doesn't increase in height as much as the other lines. With the phone book, even if the size of the phone book doubled, it would only take one more step to solve the problem.
- In fact, even if the phone book were 4 billion pages long, we would only need to divide it 32 times before we had just one page left.
- A good algorithm will solve a problem correctly and efficiently.

About CS50

- To read more in-depth about what changes we've introduced to CS50 this fall, visit cs50.ly/new (<http://cs50.ly/new>).
- We'll summarize some major changes:
 - The only lectures you need to attend are the first and last. The rest will be recorded as the course goes on, to incorporate current events. The schedule has been posted on the course website.
 - Walkthroughs will be held on Wednesdays at 1pm, where David and head TFs will offer some guidance to that week's problem set. The same information will also be recorded and included in the problem sets themselves.

- Problem sets will be released on Fridays and due 10 days later on Mondays at noon, to allow more flexibility.
- We'll start the semester with Scratch, then C, then Python, among others.
- Here are some statistics from Fall 2015 regarding the average number of hours spent on problem sets:



- There's definitely variance to the hours students spend on problem sets, but we do encourage students to take CS50 as a first year and with other courses, as we believe we have a support structure for students more and less comfortable alike.
- Sections, too, have different tracks for students more comfortable, less comfortable, and in between, held Mondays, Tuesdays, and Wednesdays with our amazing teaching staff.
- Office hours will be every day, with a schedule that will also be posted online.
- Maria, Walter, and Doug introduce themselves as the Head TF, Head CA, and Preceptor.
- Rob and Zamyra also introduce themselves as senior members on staff.
- This coming Saturday is an event we've held for a few years, CS50 Puzzle Day, where no knowledge about computer science is needed to solve puzzles in teams for exciting prizes.
- CS50 Lunches are also held on Fridays, where friends from industry talk about
- This year we'll also have the CS50 Coding Contest, where teams will be competing against each other online with just the skills we learn from the course.
- The CS50 Hackathon is an overnight event at the end of the semester where everyone works to finish (or start) their final projects.
- And a few days later is the CS50 Fair, an epic display of those final projects and how far students with no prior CS experience have come after just one semester.
- We watch a video of staff talk about how excited they are for students to take CS50 and learn from the course!

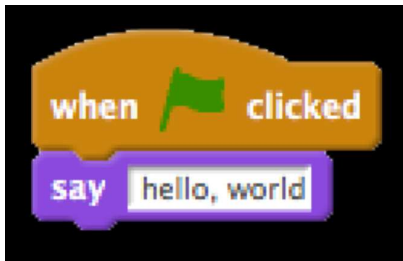
Problem Set 0

- Our first problem set will introduce some common building blocks with a graphical language called Scratch.
- Next week we'll transition to a language called C, which looks like this:

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

- We might be able to guess that this prints `hello, world` to the screen, but there's a lot of weird symbols we don't really understand yet.
- So we'll use Scratch, which allows us to drag-and-drop blocks that look like this:



- We'll be able to express lots of programming concepts with Scratch, and use it to build interactive games or art.
- For example:
 - **functions**, or actions that do something
 - **loops**, which do something over and over again
 - **variables**, a placeholder for values
 - **Boolean expressions**, true or false questions
 - **conditions**, branches or forks in the road
 - **arrays, threads, events**, fancier constructs we'll look at as we go along
- So functions like `say` in Scratch will be a purple puzzle piece:



- Notice that there's a argument, or parameter, in this case the white text box that we type in `hello, world`, to customize what the function does.
- A loop will look like this, and indicate something happening over and over again in a cycle:



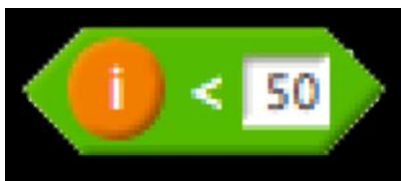
- We could also repeat a specific number of times:



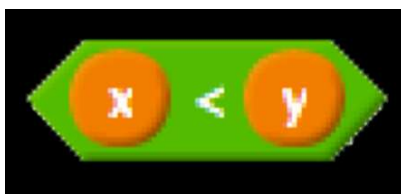
- We can create a variable, call it `i`, for an integer, and set its value to `0`:



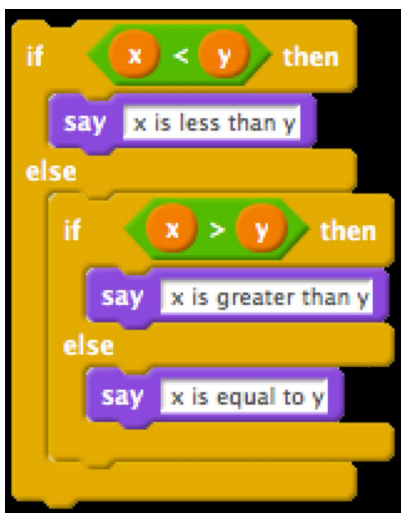
- Here is what a Boolean expression looks like in Scratch:



- We can also compare any two variables:



- With conditions, we can build something like this:



- This will tell us if `x` is less than, greater than, or equal to `y`.

- Notice how initially there are two forks, whether `x < y`, and if not, we have two more forks inside, to form a total of three possible forks.
- An array is like a list, storing multiple pieces of information:



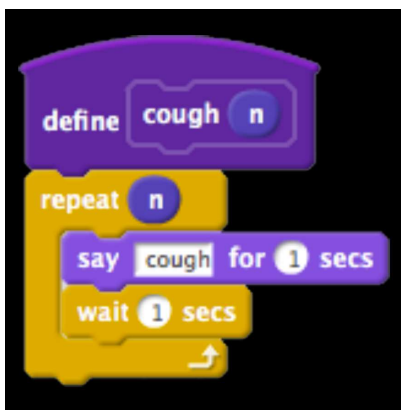
- These two blocks represent what we call multithreading, which means that a program can do multiple things at the same time because modern computers themselves are multithreaded.



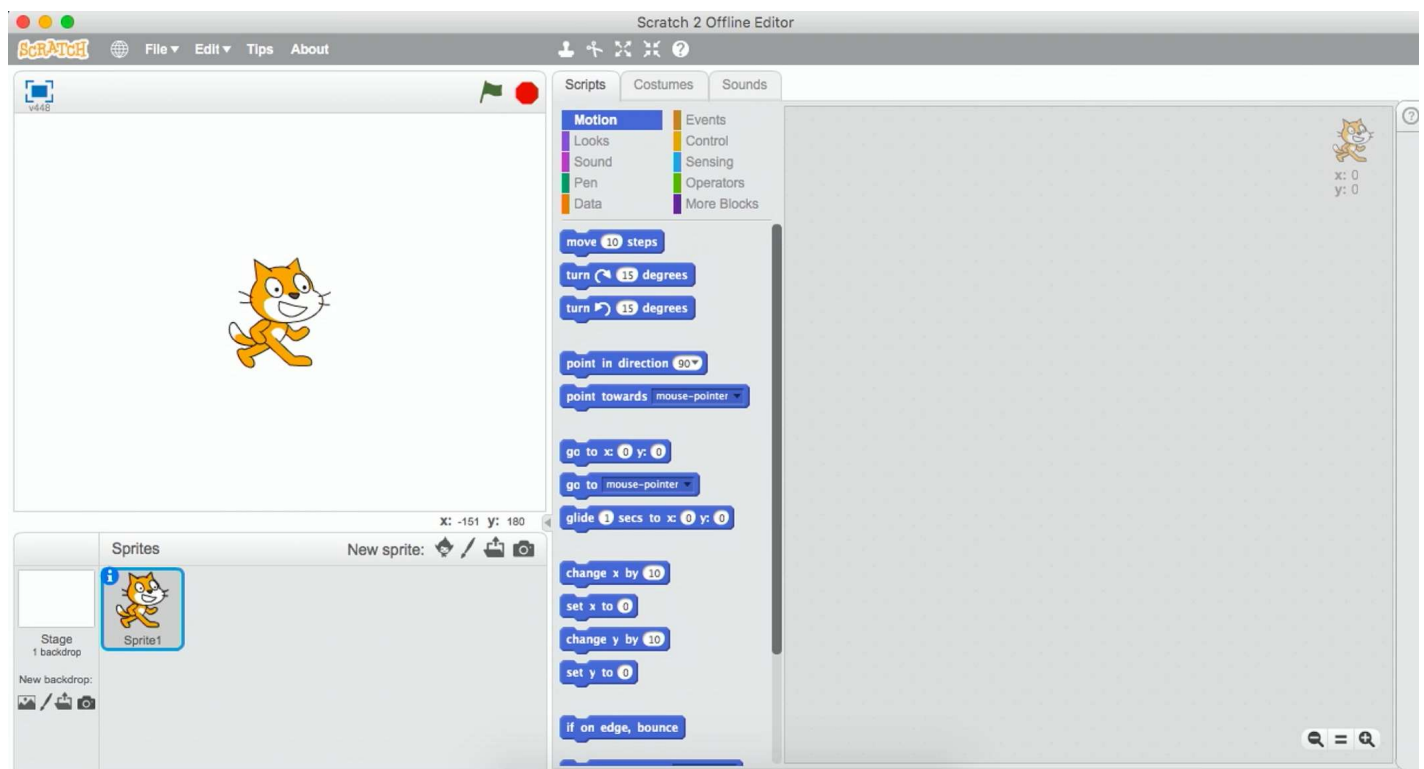
- There are also events in Scratch:



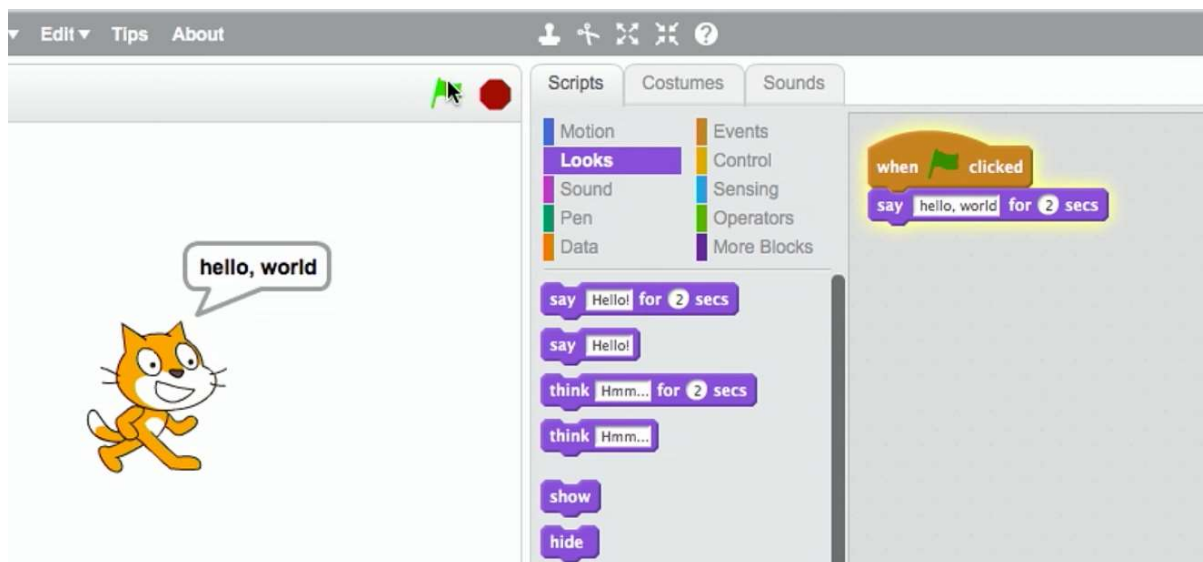
- And a way to make our own custom functions, too:



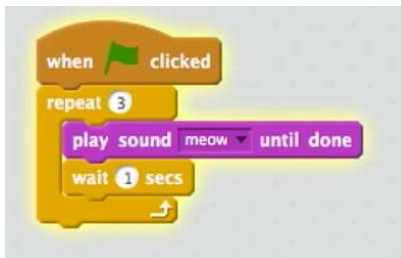
- Back in the day, David created his own game when he discovered Scratch, and now we have a modern recreation of it. Check that out, among other examples, [here](https://scratch.mit.edu/studios/3003963/) (<https://scratch.mit.edu/studios/3003963/>).
- We notice that, even though there are many moving pieces to this game, we can build each component individually, perhaps just having each item of trash fall from the sky first.
- Then we could add dragging, and then reacting to being near a trash can.
- We take a look at the Scratch editor:



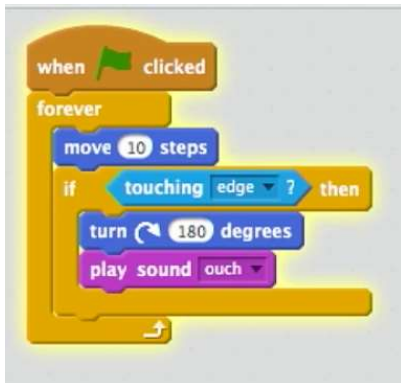
- The box to the left is the stage, or the area of the program we're working on, and right now it has a default character, Scratch the cat.
- To the center is a toolbox of blocks we can choose from, in various categories.
- And to the right is the area where we might drag and drop blocks in to do things.
- We can drag a few blocks to have Scratch say something (<https://scratch.mit.edu/projects/26329230/>):



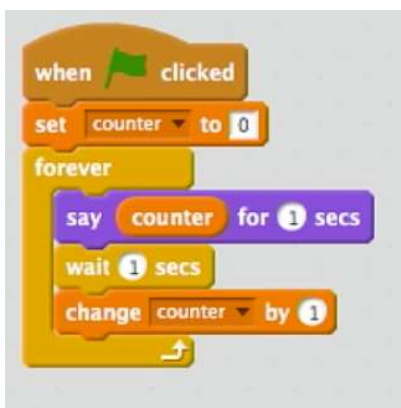
- We can experiment with other blocks like `play sound` and `wait` to have Scratch meow.
- And to have it repeat, instead of dragging the same blocks in over and over again, we can use the `repeat` block (<https://scratch.mit.edu/projects/119656319/>):



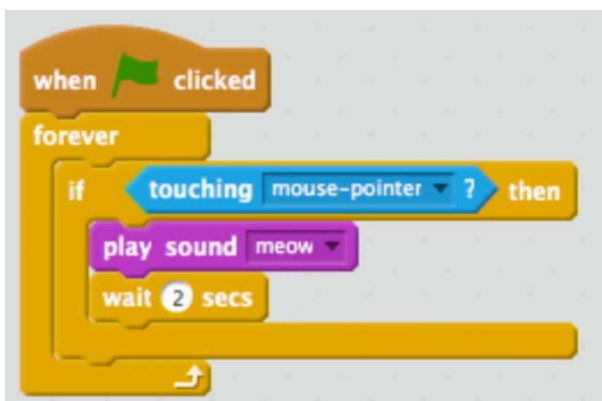
- With a little more experimentation, we can have Scratch go back and forth on the screen and even say "ouch" (<https://scratch.mit.edu/projects/119656285/>):



- Now we'll demonstrate how to use a variable (<https://scratch.mit.edu/projects/26329219/>):

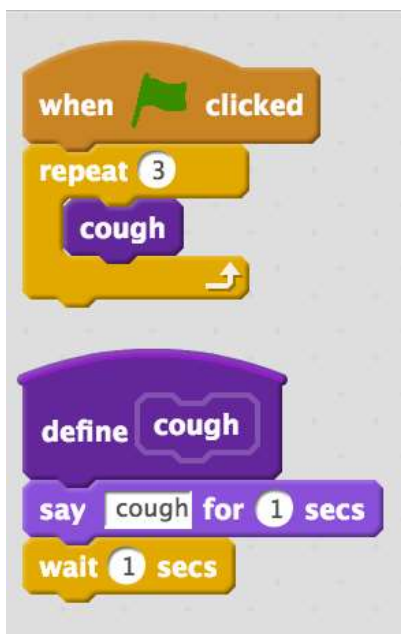


- First, we give our variable a name like `counter`, and then set it to `0`. Then, each time after we say it, we `change counter by 1` to increase it.
- We can do something a little more interactive, where Scratch will say "meow" when the cursor touches it (<https://scratch.mit.edu/projects/26329249/>).

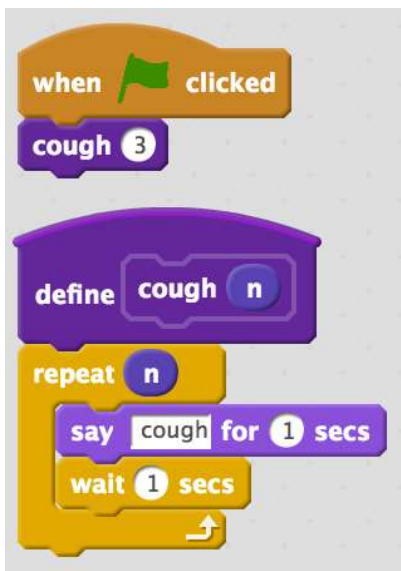


- And a variation where it doesn't (<https://scratch.mit.edu/projects/26329222/>) ...

- A thread is just a series of things that a computer is doing in order, so multiple threads mean that multiple series of things being done at the same time. In threads (<https://scratch.mit.edu/projects/26329252/>), we have two characters, each of which have a series of blocks associated with them, that both start with a `when green flag clicked`. So when we click the green flag, both characters will start moving simultaneously.
- And another example (<https://scratch.mit.edu/projects/26329236/>) has two threads for the same character, where the seal continues to bark until the `muted` variable is set to `1`, or true. Both threads are checking for either the space key being pressed or the value of the `muted` variable, over and over again.
- And finally, someone went through a lot of effort to implement Pokemon Go (<https://scratch.mit.edu/projects/116739847/>) in Scratch, and there are lots of building blocks but none much fancier than what we've already played with.
- If we look at another version (<https://scratch.mit.edu/projects/26329207/>) of the program we started with, we can see how we can make a custom `cough` puzzle piece:



- We defined `cough` with the chunk of blocks below, so our main block up top can simply use `cough` without worrying about how it is implemented.
- We can go even further (<https://scratch.mit.edu/projects/26329214/>) and have `cough` itself take in some argument for the number of times to cough, and now we can simply call that block and tell it how many times to cough:



- Finally, we can add yet another layer, the `say (word) (n) times` function, so we can generalize this action and create a `cough` and `sneeze` function on top of that, without repeating code (or blocks):



- And notice that at top, our program is quite simple in just calling `cough` and `sneeze`, rather than include all the `repeat` and `say` blocks over and over again.
- Finally, we take a look at what awaits us (<https://www.youtube.com/watch?v=bQtE1Fn8tTA>) in CS50.

