# Variables, Conditionals, Loops, and Functions

## The Absolute Basics of Making a Computer Do Stuff

Tim Jackman

BU Summer Challenge

July 7th, 2025

# Algorithms and Programs

- An *algorithm* is a series of precise instructions on how to do something

# Algorithms and Programs

- An *algorithm* is a series of precise instructions on how to do something
- A *program* is an implementation of an algorithm in a specific computer language

# Algorithms and Programs

- An *algorithm* is a series of precise instructions on how to do something
- A *program* is an implementation of an algorithm in a specific computer language
- Today we'll be programming in Scratch

# Algorithms and Programs

- An *algorithm* is a series of precise instructions on how to do something
- A *program* is an implementation of an algorithm in a specific computer language
- Today we'll be programming in Scratch
  - The concepts will extend to other languages

# Data Comes in All Types

- *Data* is the information a program uses to do stuff

# Data Comes in All Types

- *Data* is the information a program uses to do stuff
- Data comes in different *types*

# Data Comes in All Types

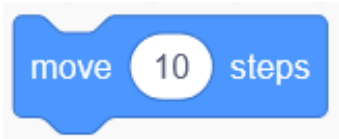
- *Data* is the information a program uses to do stuff
- Data comes in different *types*
  - Numbers, e.g. 1, -1, 3.5, 10000

# Data Comes in All Types

- *Data* is the information a program uses to do stuff
- Data comes in different *types*
  - Numbers, e.g. 1, -1, 3.5, 10000
    - In Scratch, shaped like an oval/pill

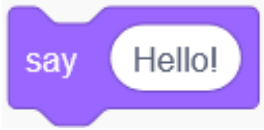
# Data Comes in All Types

- *Data* is the information a program uses to do stuff
- Data comes in different *types*
  - Numbers, e.g. 1, -1, 3.5, 10000
    - In Scratch, shaped like an oval/pill
  - Strings: text, e.g. 'abc', 'hello', 'It's 100 degrees out!'



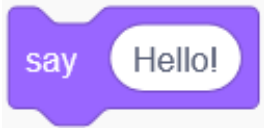
# Data Comes in All Types

- *Data* is the information a program uses to do stuff
- Data comes in different *types*
  - Numbers, e.g. 1, -1, 3.5, 10000
    - In Scratch, shaped like an oval/pill
  - Strings: text, e.g. 'abc', 'hello', 'It's 100 degrees out!'
    - In Scratch, also shaped like an oval

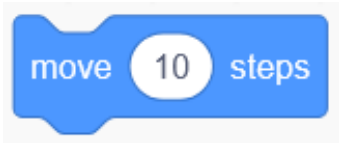
# Data Comes in All Types

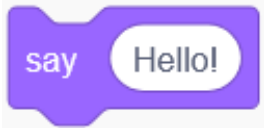
- *Data* is the information a program uses to do stuff
- Data comes in different *types*
  - Numbers, e.g. 1, -1, 3.5, 10000
    - In Scratch, shaped like an oval/pill
  - Strings: text, e.g. 'abc', 'hello', 'It's 100 degrees out!'
    - In Scratch, also shaped like an oval
  - Booleans: true, false e.g. an expression like $5 + 5 = 10$ is true while $7 < 5$ is false

# Data Comes in All Types

- *Data* is the information a program uses to do stuff
- Data comes in different *types*
  - Numbers, e.g. 1, -1, 3.5, 10000
    - In Scratch, shaped like an oval/pill
  - Strings: text, e.g. 'abc', 'hello', 'It's 100 degrees out!'
    - In Scratch, also shaped like an oval
  - Booleans: true, false e.g. an expression like $5 + 5 = 10$ is true while $7 < 5$ is false
    - Shaped like hexagons




# We Can Manipulate Data

- We use *operators* to manipulate data to produce new data

# We Can Manipulate Data

- We use *operators* to manipulate data to produce new data
- There are arithmetic operators that take two numbers and produce a new number

# We Can Manipulate Data

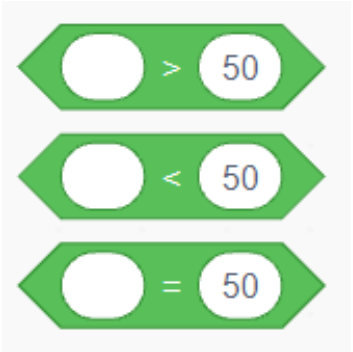
- We use *operators* to manipulate data to produce new data
- There are arithmetic operators that take two numbers and produce a new number

# We Can Manipulate Data

- We use *operators* to manipulate data to produce new data
- Sometimes operators can take in one type of data and output another type

# We Can Manipulate Data

- We use *operators* to manipulate data to produce new data
- Sometimes operators can take in one type of data and output another type

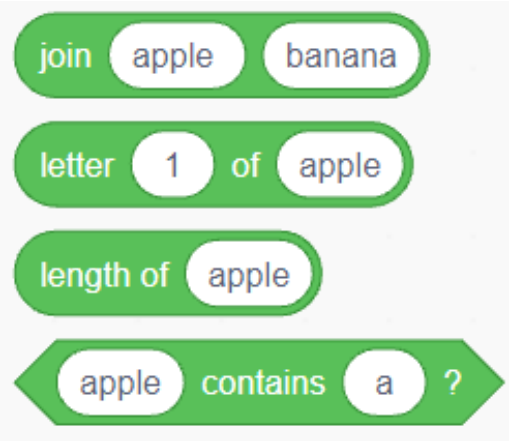


# We Can Manipulate Data

- We use *operators* to manipulate data to produce new data
- There are operators for Strings

# We Can Manipulate Data

- We use *operators* to manipulate data to produce new data
- There are operators for Strings



# A Closer Look at Mod

- An important arithmetic *operator* that is very commonly used in programming is *modulo* (mod).

# A Closer Look at Mod

- An important arithmetic *operator* that is very commonly used in programming is *modulo* (mod).
- If $p$ and $q$ are two integers, $p$ mod $q$ returns the *remainder* when $p$ is divided by $q$

# A Closer Look at Mod

- An important arithmetic *operator* that is very commonly used in programming is *modulo* (mod).
- If $p$ and $q$ are two integers, $p$ mod $q$ returns the *remainder* when $p$ is divided by $q$
  - For example, 17 mod 5 is 2, because when you divide 17 by 5, 5 goes into 17 three times with $\underline{2}$ leftover.

# A Closer Look at Mod

- An important arithmetic *operator* that is very commonly used in programming is *modulo* (mod).
- If $p$ and $q$ are two integers, $p$ mod $q$ returns the *remainder* when $p$ is divided by $q$
  - For example, 17 mod 5 is 2, because when you divide 17 by 5, 5 goes into 17 three times with 2 leftover.
  - In Scratch, it's computed as $p - q \cdot \text{roundDown}(p/q)$, but this can vary slightly between languages

# A Closer Look at Mod

- An important arithmetic *operator* that is very commonly used in programming is *modulo* (mod).
- If $p$ and $q$ are two integers, $p$ mod $q$ returns the *remainder* when $p$ is divided by $q$
  - For example, 17 mod 5 is 2, because when you divide 17 by 5, 5 goes into 17 three times with 2 leftover.
  - In Scratch, it's computed as $p - q \cdot \text{roundDown}(p/q)$, but this can vary slightly between languages
- Useful for doing actions at regular intervals
- In some programming languages, it is denoted as %

# A Closer Look at Mod

- An important arithmetic *operator* that is very commonly used in programming is *modulo* (mod).
- If $p$ and $q$ are two integers, $p$ mod $q$ returns the *remainder* when $p$ is divided by $q$
  - For example, 17 mod 5 is 2, because when you divide 17 by 5, 5 goes into 17 three times with 2 leftover.
  - In Scratch, it's computed as $p - q \cdot \text{roundDown}(p/q)$, but this can vary slightly between languages
- Useful for doing actions at regular intervals
- In some programming languages, it is denoted as %
- Typically only used with integers, in some languages (for example Scratch and Java) you can use it with decimals, but it might not be fully accurate.

# The Boolean Operators

- There are three basic Boolean operators that take in Booleans and output new Booleans: *not*, *and*, and *or*

# The Boolean Operators

- There are three basic Boolean operators that take in Booleans and output new Booleans: *not*, *and*, and *or*
- *not* flips the value of the given Boolean: true becomes false, false becomes true

# The Boolean Operators

- There are three basic Boolean operators that take in Booleans and output new Booleans: *not*, *and*, and *or*
- *not* flips the value of the given Boolean: true becomes false, false becomes true
  - Example: *not* $5 < 2$ evaluates to true since $5 < 2$ is false

# The Boolean Operators

- There are three basic Boolean operators that take in Booleans and output new Booleans: *not*, *and*, and *or*
- *not* flips the value of the given Boolean: true becomes false, false becomes true
  - Example: *not* $5 < 2$ evaluates to true since $5 < 2$ is false
  - Alternative symbols used for *not* in other languages: ! (Java, Javascript), $\neg$ (Mathematics), `not` (Python), ~(Matlab)

# The Boolean Operators

- *and* and *or* are binary operators: they take in two Boolean inputs and return a single Boolean

# The Boolean Operators

- *and* and *or* are binary operators: they take in two Boolean inputs and return a single Boolean
- *and* evaluates to true only if both its inputs are true

# The Boolean Operators

- *and* and *or* are binary operators: they take in two Boolean inputs and return a single Boolean
- *and* evaluates to true only if both its inputs are true
  - Example: $(5 > 2)$ *and* $(1 + 1 = 2)$ evaluates to true since both inputs are true

# The Boolean Operators

- *and* and *or* are binary operators: they take in two Boolean inputs and return a single Boolean
- *and* evaluates to true only if both its inputs are true
  - Example: $(5 > 2)$ *and* $(1 + 1 = 2)$ evaluates to true since both inputs are true
  - Alternative symbols used for *and* in other languages: && (Java, Javascript), $\wedge$ (Math), `and` (Python)

# The Boolean Operators

- *and* and *or* are binary operators: they take in two Boolean inputs and return a single Boolean
- *and* evaluates to true only if both its inputs are true
  - Example: $(5 > 2)$ *and* $(1 + 1 = 2)$ evaluates to true since both inputs are true
  - Alternative symbols used for *and* in other languages: && (Java, Javascript), $\wedge$ (Math), `and` (Python)
- *or* evaluates to true if at least one (or both) inputs is true

# The Boolean Operators

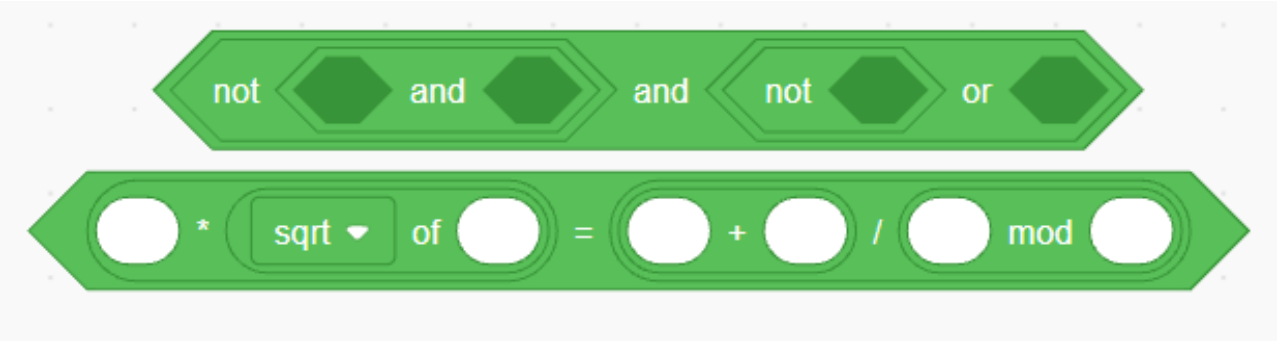
- *and* and *or* are binary operators: they take in two Boolean inputs and return a single Boolean
- *and* evaluates to true only if both its inputs are true
  - Example: $(5 > 2)$ *and* $(1 + 1 = 2)$ evaluates to true since both inputs are true
  - Alternative symbols used for *and* in other languages: && (Java, Javascript), $\wedge$ (Math), `and` (Python)
- *or* evaluates to true if at least one (or both) inputs is true
  - Example: $(5 < 2)$ *or* $(1 + 1 = 2)$ is true since $1 + 1 = 2$ is true

# The Boolean Operators

- *and* and *or* are binary operators: they take in two Boolean inputs and return a single Boolean
- *and* evaluates to true only if both its inputs are true
  - Example: $(5 > 2)$ *and* $(1 + 1 = 2)$ evaluates to true since both inputs are true
  - Alternative symbols used for *and* in other languages: && (Java, Javascript), $\wedge$ (Math), `and` (Python)
- *or* evaluates to true if at least one (or both) inputs is true
  - Example: $(5 < 2)$ *or* $(1 + 1 = 2)$ is true since $1 + 1 = 2$ is true
  - Alternative symbols used for *or* in other languages: || (Java, Javascript), $\vee$ (Math), `or` (Python)

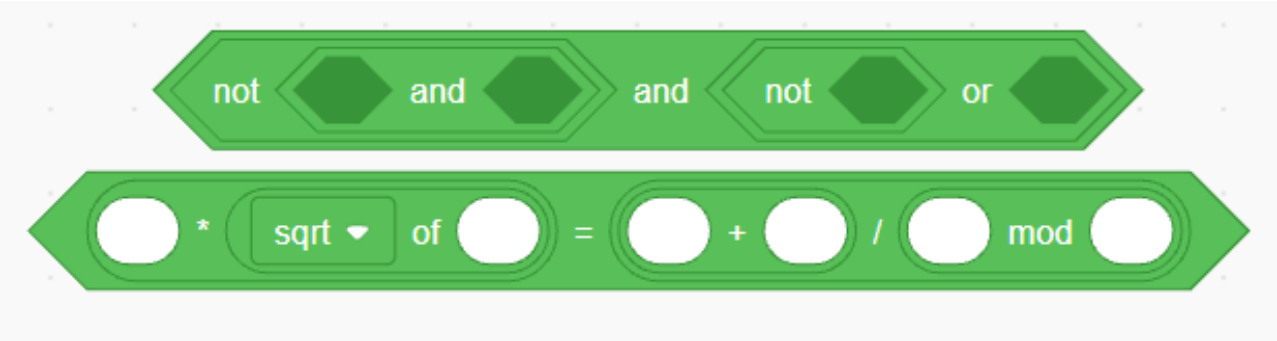
# Nested Operators

- We can create complex expressions by nesting these operators



# Nested Operators

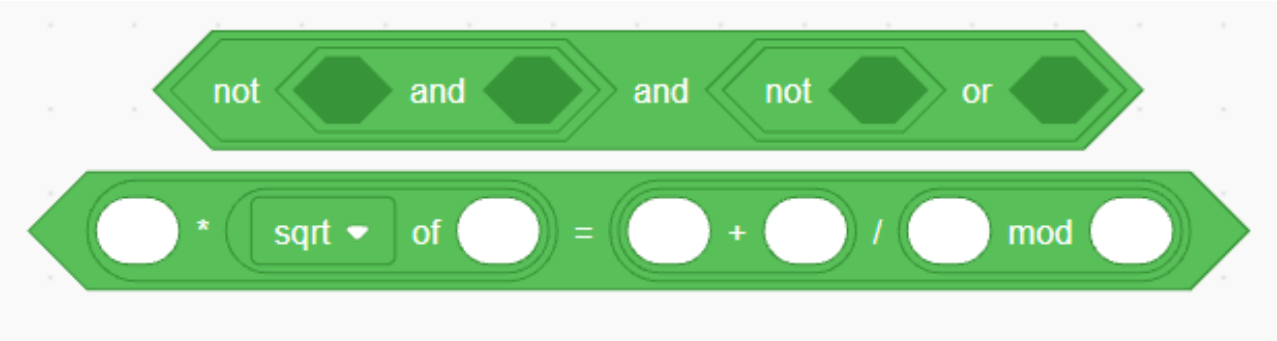
- We can create complex expressions by nesting these operators




- In Scratch, the order of evaluation is clear (evaluate from the innermost thing out)

# Nested Operators

- We can create complex expressions by nesting these operators




- In Scratch, the order of evaluation is clear (evaluate from the innermost thing out)
- In other languages, there are varying orders of operations (like PEMDAS in Math), but typically we'll use parentheses to make avoid both *ambiguity* and *bugs* (like in Math)

# Sometimes Types Can Be "Dynamic"

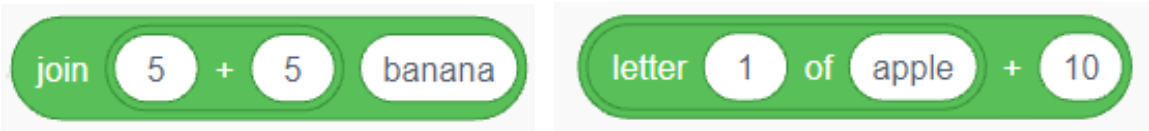
- Notice in Scratch that Numbers and Strings are both pill shaped!

# Sometimes Types Can Be "Dynamic"

- Notice in Scratch that Numbers and Strings are both pill shaped!
- While Scratch won't let you write a String where it expects a Number, it's very easy for a String to end up where a Number is expected or vice versa

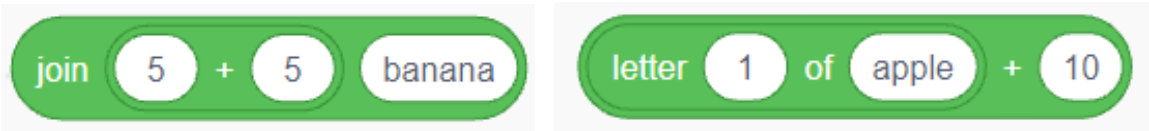
# Sometimes Types Can Be "Dynamic"

- Notice in Scratch that Numbers and Strings are both pill shaped!
- While Scratch won't let you write a String where it expects a Number, it's very easy for a String to end up where a Number is expected or vice versa



# Sometimes Types Can Be "Dynamic"

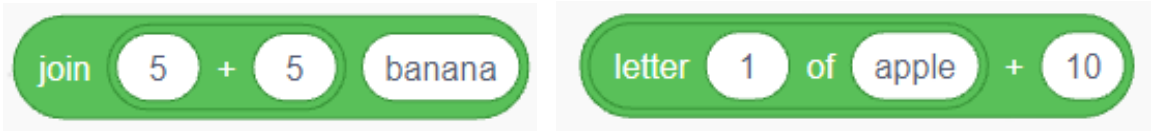
- Notice in Scratch that Numbers and Strings are both pill shaped!
- While Scratch won't let you write a String where it expects a Number, it's very easy for a String to end up where a Number is expected or vice versa




- Scratch tries to *coerce* (automatically convert) Data between types

# Sometimes Types Can Be "Dynamic"

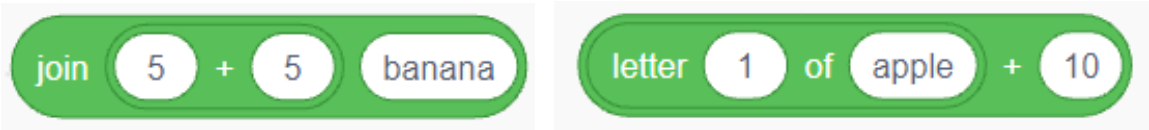
- Notice in Scratch that Numbers and Strings are both pill shaped!
- While Scratch won't let you write a String where it expects a Number, it's very easy for a String to end up where a Number is expected or vice versa



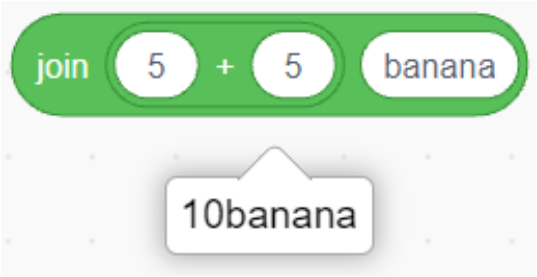

- Scratch tries to *coerce* (automatically convert) Data between types
- Numbers become Strings in the natural way when a String is expected

# Sometimes Types Can Be "Dynamic"

- Notice in Scratch that Numbers and Strings are both pill shaped!
- While Scratch won't let you write a String where it expects a Number, it's very easy for a String to end up where a Number is expected or vice versa

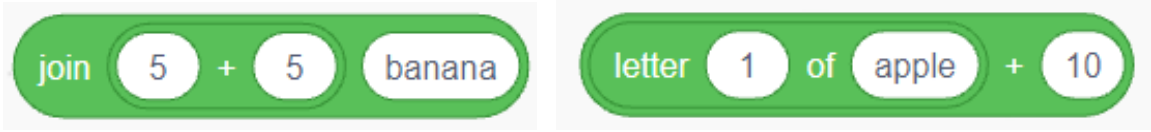



- Scratch tries to *coerce* (automatically convert) Data between types
- Numbers become Strings in the natural way when a String is expected



# Sometimes Types Can Be "Dynamic"

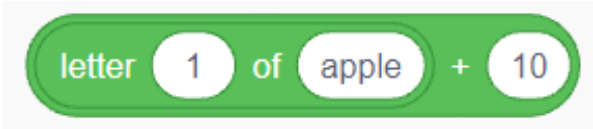
- Notice in Scratch that Numbers and Strings are both pill shaped!
- While Scratch won't let you write a String where it expects a Number, it's very easy for a String to end up where a Number is expected or vice versa



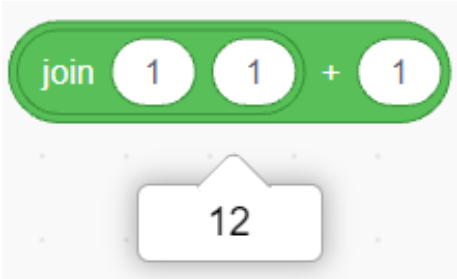

- Scratch tries to *coerce* (automatically convert) Data between types
- Numerical Strings are coerced back into Numbers if a Number is expected

# Sometimes Types Can Be "Dynamic"

- Notice in Scratch that Numbers and Strings are both pill shaped!
- While Scratch won't let you write a String where it expects a Number, it's very easy for a String to end up where a Number is expected or vice versa

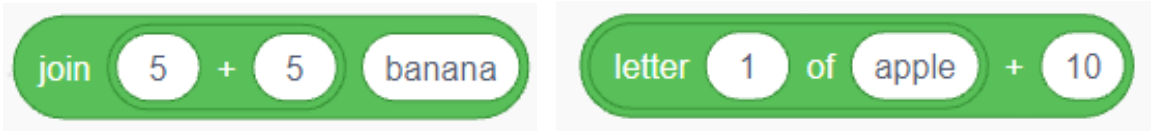




- Scratch tries to *coerce* (automatically convert) Data between types
- Numerical Strings are coerced back into Numbers if a Number is expected



# Sometimes Types Can Be "Dynamic"

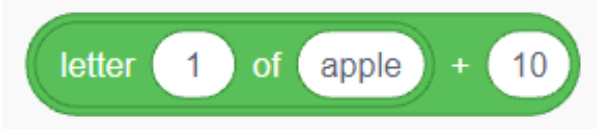
- Notice in Scratch that Numbers and Strings are both pill shaped!
- While Scratch won't let you write a String where it expects a Number, it's very easy for a String to end up where a Number is expected or vice versa



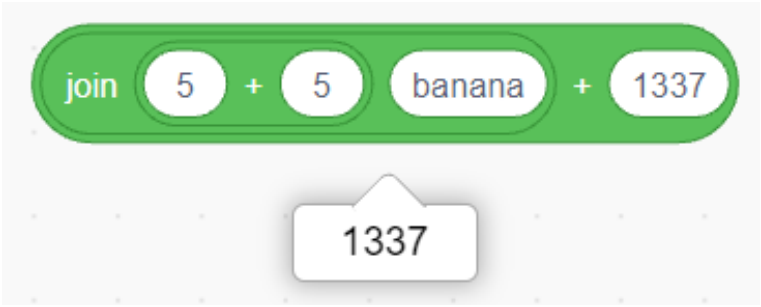

- Scratch tries to *coerce* (automatically convert) Data between types
- Non-numerical Strings are coerced to 0 if a Number is expected

# Sometimes Types Can Be "Dynamic"

- Notice in Scratch that Numbers and Strings are both pill shaped!
- While Scratch won't let you write a String where it expects a Number, it's very easy for a String to end up where a Number is expected or vice versa


- Scratch tries to *coerce* (automatically convert) Data between types
- Non-numerical Strings are coerced to 0 if a Number is expected



# Variables Store "Data"

- Our first program was a bit boring: very simple, everything was *hard-coded*, no interaction

# Variables Store "Data"

- Our first program was a bit boring: very simple, everything was *hard-coded*, no interaction
- Interesting programs need to work with *more* data that changes overtime

# Variables Store "Data"

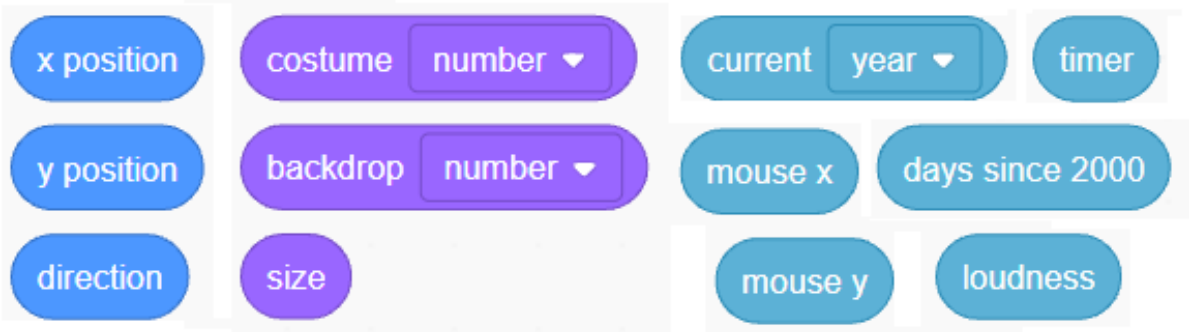
- Our first program was a bit boring: very simple, everything was *hard-coded*, no interaction
- Interesting programs need to work with *more* data that changes overtime
- We can use *variables* to store and work with more complex data.

# Variables Store "Data"

- Our first program was a bit boring: very simple, everything was *hard-coded*, no interaction
- Interesting programs need to work with *more* data that changes overtime
- We can use *variables* to store and work with more complex data.
  - Scratch has a number of built-in variables for the sprite (x position, y position, rotation, size) and the system (mouse position, data and time, loudness)

# Variables Store "Data"

- Our first program was a bit boring: very simple, everything was *hard-coded*, no interaction
- Interesting programs need to work with *more* data that changes overtime
- We can use *variables* to store and work with more complex data.
  - Scratch has a number of built-in variables for the sprite (x position, y position, rotation, size) and the system (mouse position, data and time, loudness)



# Variables Store "Data"

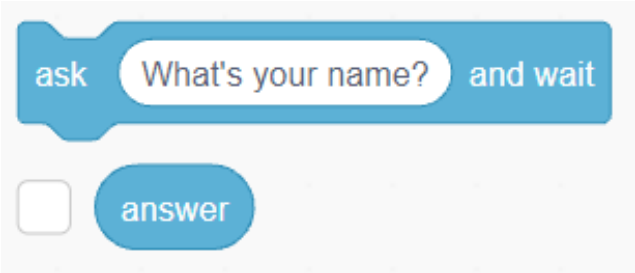
- Our first program was a bit boring: very simple, everything was *hard-coded*, no interaction
- Interesting programs need to work with *more* data that changes overtime
- We can use *variables* to store and work with more complex data.
  - Scratch has a number of built-in variables for the sprite (x position, y position, rotation, size) and the system (mouse position, data and time, loudness)

- Variables let us work with user inputs

# Variables Store "Data"

- Our first program was a bit boring: very simple, everything was *hard-coded*, no interaction
- Interesting programs need to work with *more* data that changes overtime
- We can use *variables* to store and work with more complex data.
  - Scratch has a number of built-in variables for the sprite (x position, y position, rotation, size) and the system (mouse position, data and time, loudness)

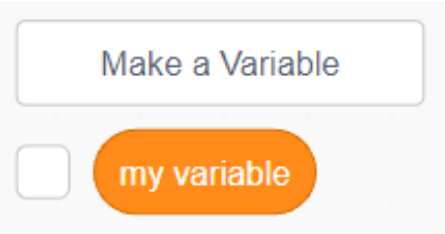
- Variables let us work with user inputs




- We can even make our own variables

# Variables Store "Data"

- Our first program was a bit boring: very simple, everything was *hard-coded*, no interaction
- Interesting programs need to work with *more* data that changes overtime
- We can use *variables* to store and work with more complex data.
  - Scratch has a number of built-in variables for the sprite (x position, y position, rotation, size) and the system (mouse position, data and time, loudness)
- Variables let us work with user inputs
- We can even make our own variables



# We Can Manipulate Variables

- One really powerful thing about variables is they are *mutable* (can change them over time)

# We Can Manipulate Variables

- One really powerful thing about variables is they are *mutable* (can change them over time)
- Scratch built-in variables update as sprites and the system update

# We Can Manipulate Variables

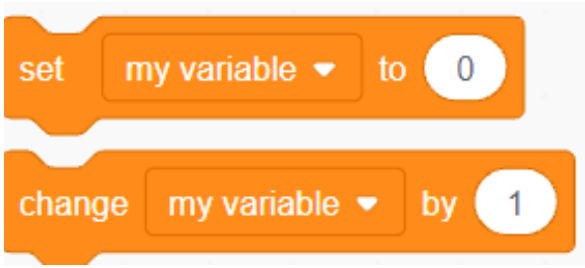
- One really powerful thing about variables is they are *mutable* (can change them over time)
- Scratch built-in variables update as sprites and the system update
- The answer variable updates whenever we use an ask block

# We Can Manipulate Variables

- One really powerful thing about variables is they are *mutable* (can change them over time)
- Scratch built-in variables update as sprites and the system update
- The answer variable updates whenever we use an ask block
- We can manually update our own variables

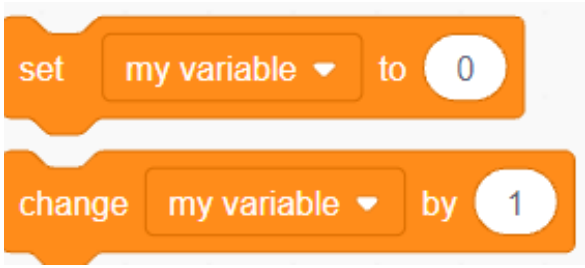
# We Can Manipulate Variables

- One really powerful thing about variables is they are *mutable* (can change them over time)
- Scratch built-in variables update as sprites and the system update
- The answer variable updates whenever we use an ask block
- We can manually update our own variables



# We Can Manipulate Variables

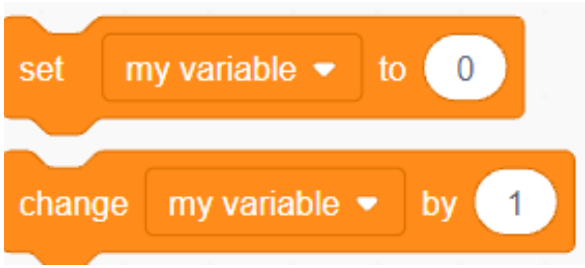
- One really powerful thing about variables is they are *mutable* (can change them over time)
- Scratch built-in variables update as sprites and the system update
- The answer variable updates whenever we use an ask block
- We can manually update our own variables




  - set allows us to change a variable to any Number or String

# We Can Manipulate Variables

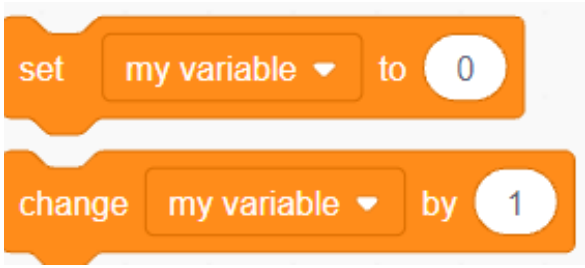
- One really powerful thing about variables is they are *mutable* (can change them over time)
- Scratch built-in variables update as sprites and the system update
- The answer variable updates whenever we use an ask block
- We can manually update our own variables




  - set allows us to change a variable to any Number or String
  - change increments variables by the given amount

# We Can Manipulate Variables

- One really powerful thing about variables is they are *mutable* (can change them over time)
- Scratch built-in variables update as sprites and the system update
- The answer variable updates whenever we use an ask block
- We can manually update our own variables




  - set allows us to change a variable to any Number or String
  - change increments variables by the given amount
  - "change v by x" is just *syntatic sugar* for "set v to (v + x)" (it's easier to read and use)

# Demo Time!

# Conditionals: Doing Stuff... Sometimes!

- Interesting programs are reactive - they make decisions based on the changing data

# Conditionals: Doing Stuff... Sometimes!

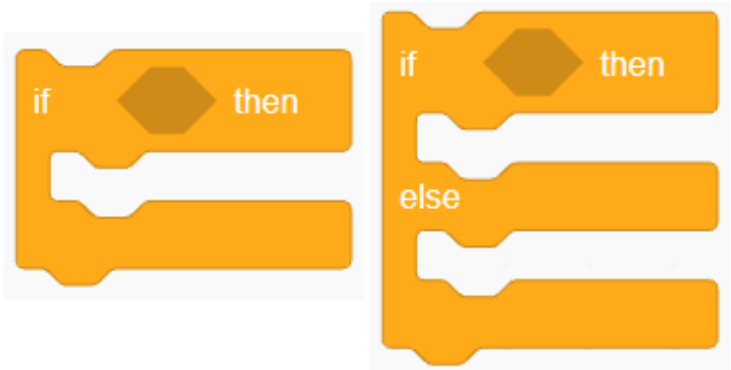
- Interesting programs are reactive - they make decisions based on the changing data
- Use conditionals to run code *sometimes* if a condition is met:

# Conditionals: Doing Stuff... Sometimes!

- Interesting programs are reactive - they make decisions based on the changing data
- Use conditionals to run code *sometimes* if a condition is met:
- Two basic kinds of conditionals are "if..." and "if... else..."

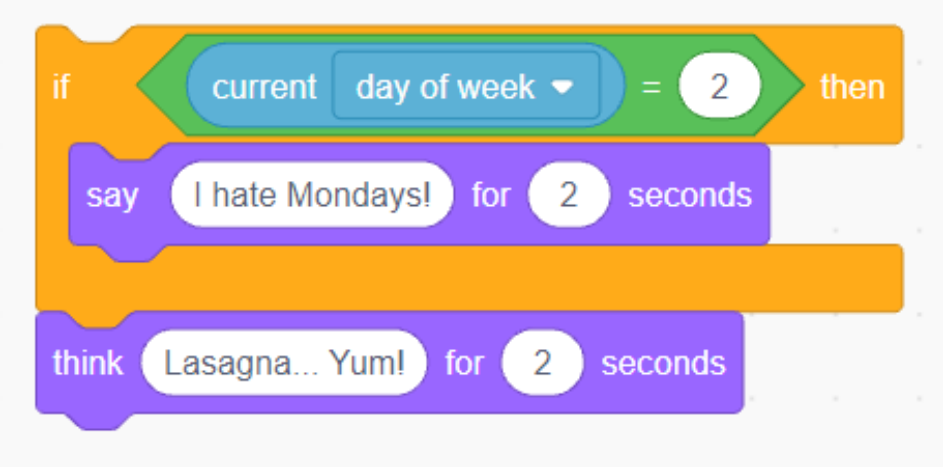
# Conditionals: Doing Stuff... Sometimes!

- Interesting programs are reactive - they make decisions based on the changing data
- Use conditionals to run code *sometimes* if a condition is met:
- Two basic kinds of conditionals are "if..." and "if... else..."



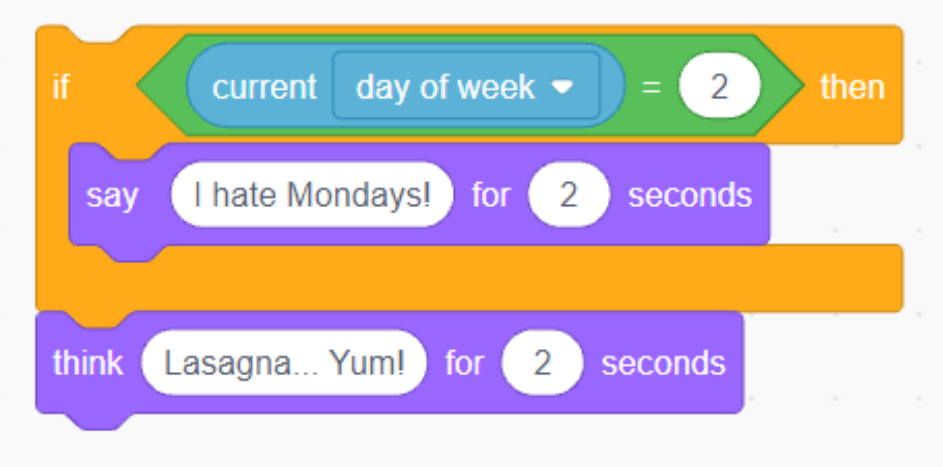
# Conditionals: Doing Stuff... Sometimes!

- Interesting programs are reactive - they make decisions based on the changing data
- Use conditionals to run code *sometimes* if a condition is met:
- Two basic kinds of conditionals are "if..." and "if... else..."
  - "if ..." blocks run the nested code if it's Boolean input evaluates to true, otherwise it moves to the next block



# Conditionals: Doing Stuff... Sometimes!

- Interesting programs are reactive - they make decisions based on the changing data
- Use conditionals to run code *sometimes* if a condition is met:
- Two basic kinds of conditionals are "if..." and "if... else..."
  - "if ..." blocks run the nested code if it's Boolean input evaluates to true, otherwise it moves to the next block




  - The program proceeds to the block of code, whether the condition is met or not

# Conditionals: Doing Stuff... Sometimes!

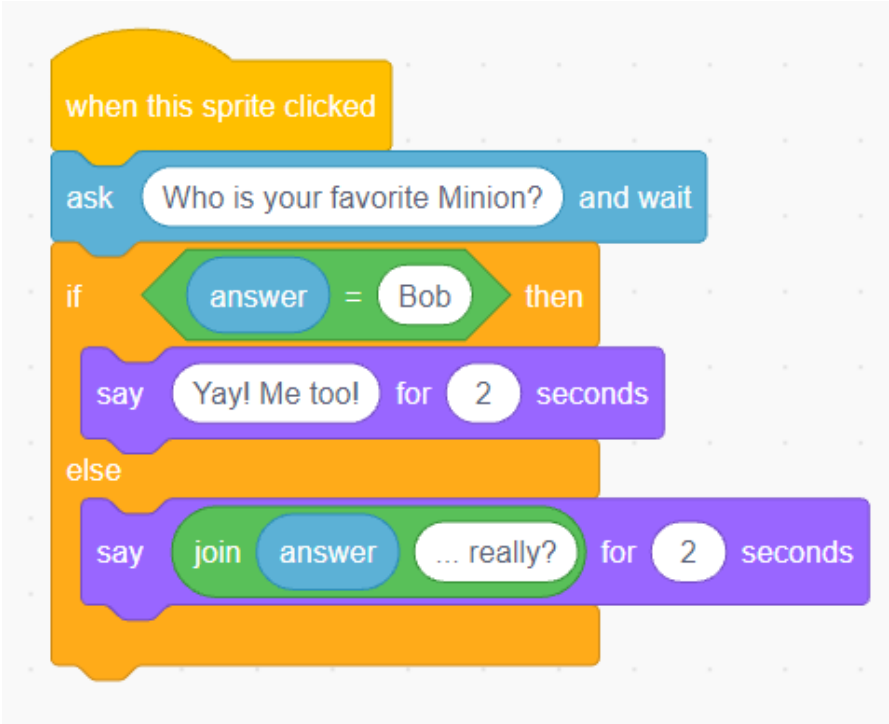
- "if ... else ..." blocks run the first nested code if the Boolean input is true and runs the second nested code if the input is false

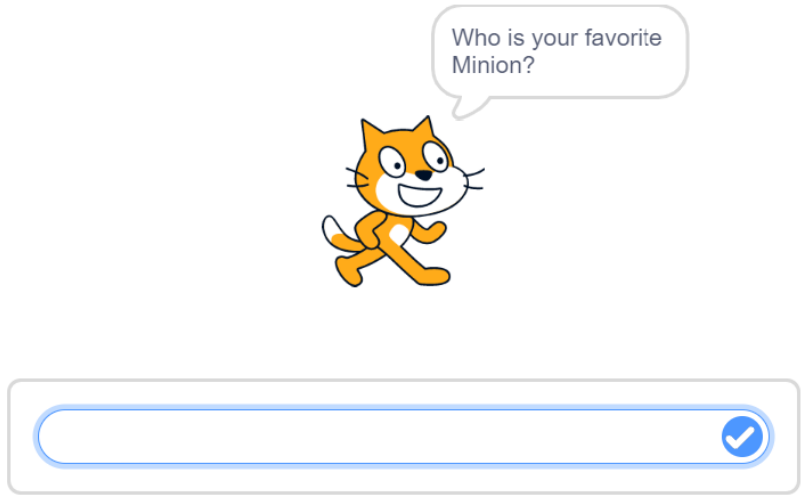
# Conditionals: Doing Stuff... Sometimes!

- "if ... else ..." blocks run the first nested code if the Boolean input is true and runs the second nested code if the input is false

# Conditionals: Doing Stuff... Sometimes!

- "if ... else ..." blocks run the first nested code if the Boolean input is true and runs the second nested code if the input is false



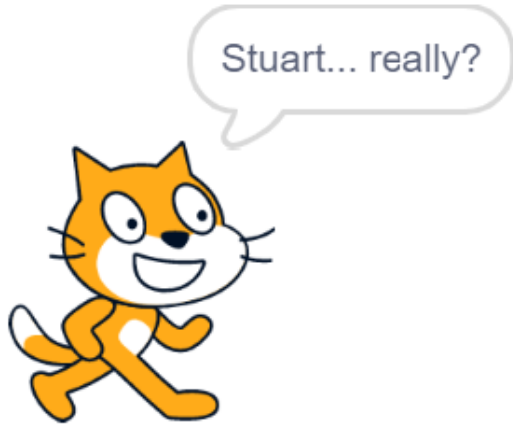



# Conditionals: Doing Stuff... Sometimes!

- "if ... else ..." blocks run the first nested code if the Boolean input is true and runs the second nested code if the input is false


# Conditionals: Doing Stuff... Sometimes!

- "if ... else ..." blocks run the first nested code if the Boolean input is true and runs the second nested code if the input is false

# Conditionals: Doing Stuff... Sometimes!

- "if ... else ..." blocks run the first nested code if the Boolean input is true and runs the second nested code if the input is false


- You can nest conditionals to get more complicated control flows

# Loops: Doing Stuff Many Times!

- Imagine you want your program to do the same thing multiple times... how can we achieve this?

# Loops: Doing Stuff Many Times!

- Imagine you want your program to do the same thing multiple times... how can we achieve this?
- If it's just two or three times, we could easily copy and paste our code

# Loops: Doing Stuff Many Times!

- Imagine you want your program to do the same thing multiple times... how can we achieve this?
- If it's just two or three times, we could easily copy and paste our code
- But what if we wanted to do something 100 times? What if we copied it a bunch and then decided we wanted to make a change?

# Loops: Doing Stuff Many Times!

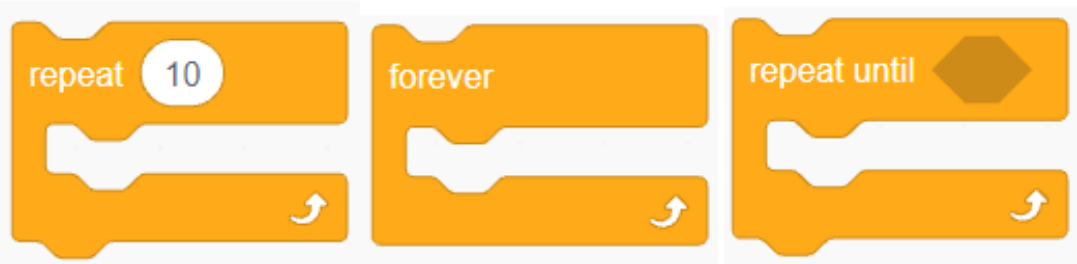
- Imagine you want your program to do the same thing multiple times... how can we achieve this?
- If it's just two or three times, we could easily copy and paste our code
- But what if we wanted to do something 100 times? What if we copied it a bunch and then decided we wanted to make a change?
- Hard duplicating code is *bad*
- Instead of copying and pasting, use loops!

# Loops: Doing Stuff Many Times!

- Imagine you want your program to do the same thing multiple times... how can we achieve this?
- If it's just two or three times, we could easily copy and paste our code
- But what if we wanted to do something 100 times? What if we copied it a bunch and then decided we wanted to make a change?
- Hard duplicating code is *bad*
- Instead of copying and pasting, use loops!
- There are three basic kinds of loops in Scratch: "repeat x times", "forever..." and "repeat until ..."

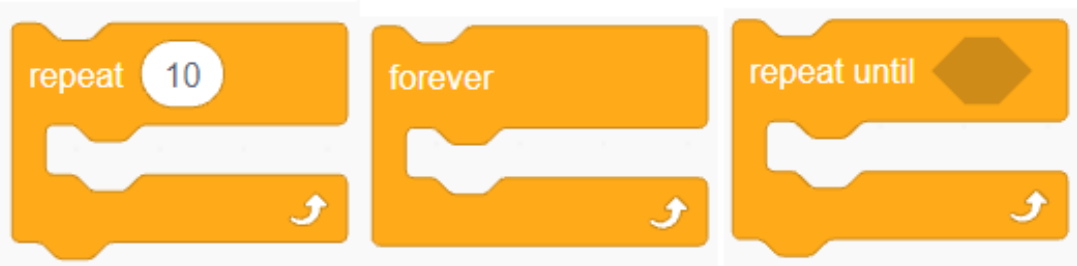
# Loops: Doing Stuff Many Times!

- Imagine you want your program to do the same thing multiple times... how can we achieve this?
- If it's just two or three times, we could easily copy and paste our code
- But what if we wanted to do something 100 times? What if we copied it a bunch and then decided we wanted to make a change?
- Hard duplicating code is *bad*
- Instead of copying and pasting, use loops!
- There are three basic kinds of loops in Scratch: "repeat x times", "forever..." and "repeat until ..."



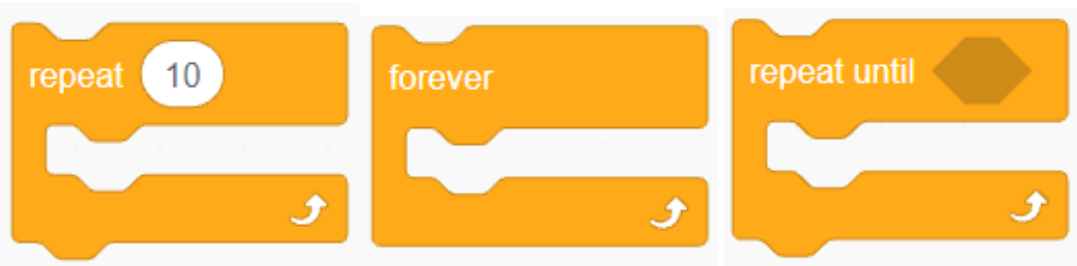
# Loops: Doing Stuff Many Times!

- There are three basic kinds of loops in Scratch: "repeat x times", "forever..." and "repeat until ..."



# Loops: Doing Stuff Many Times!

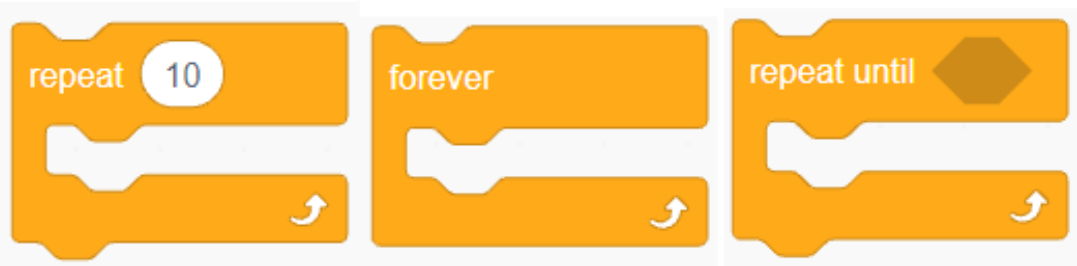
- There are three basic kinds of loops in Scratch: "repeat x times", "forever..." and "repeat until ..."




- "repeat x times" loops will run the nested code x times

# Loops: Doing Stuff Many Times!

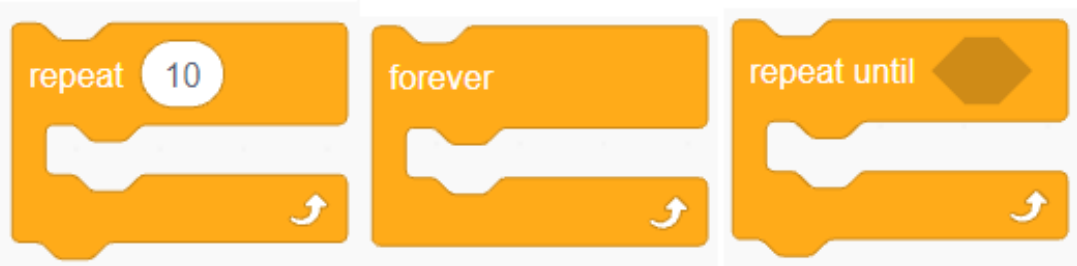
- There are three basic kinds of loops in Scratch: "repeat x times", "forever..." and "repeat until ..."




- "repeat x times" loops will run the nested code x times
- "forever" loops will repeatedly run the nested code forever

# Loops: Doing Stuff Many Times!

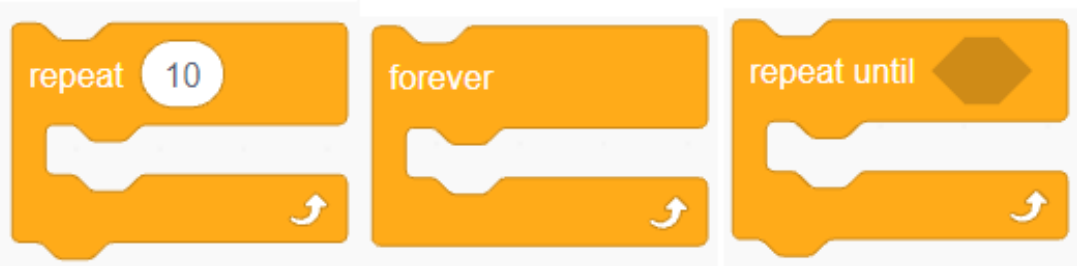
- There are three basic kinds of loops in Scratch: "repeat x times", "forever..." and "repeat until ..."




- "repeat x times" loops will run the nested code x times
- "forever" loops will repeatedly run the nested code forever
- "repeat until ..." loops will run the nested code until the Boolean input evaluates to true

# Loops: Doing Stuff Many Times!

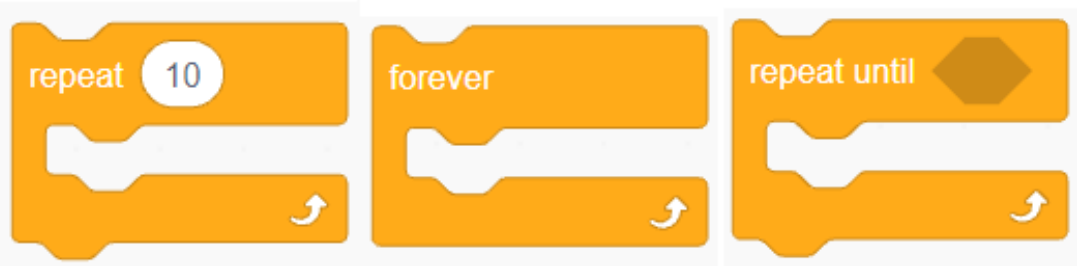
- There are three basic kinds of loops in Scratch: "repeat x times", "forever..." and "repeat until ..."




- "repeat x times" loops will run the nested code x times
- "forever" loops will repeatedly run the nested code forever
- "repeat until ..." loops will run the nested code until the Boolean input evaluates to true
  - If the condition is initially true then the nested code is *never* run

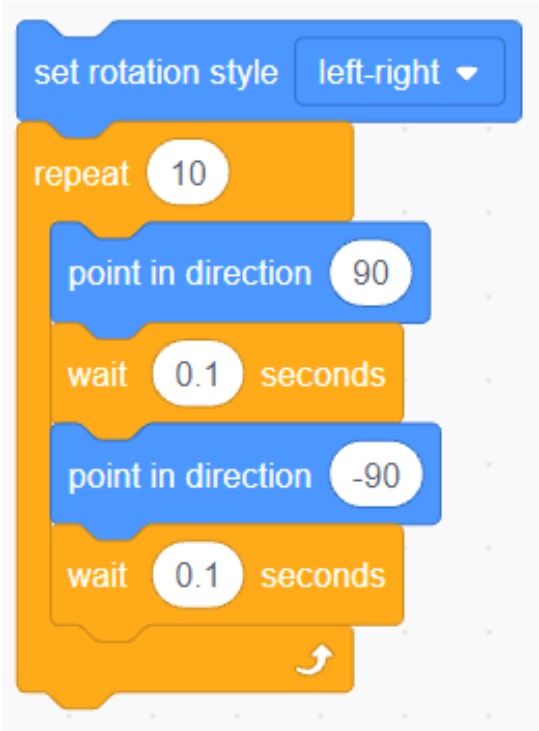
# Loops: Doing Stuff Many Times!

- There are three basic kinds of loops in Scratch: "repeat x times", "forever..." and "repeat until ..."

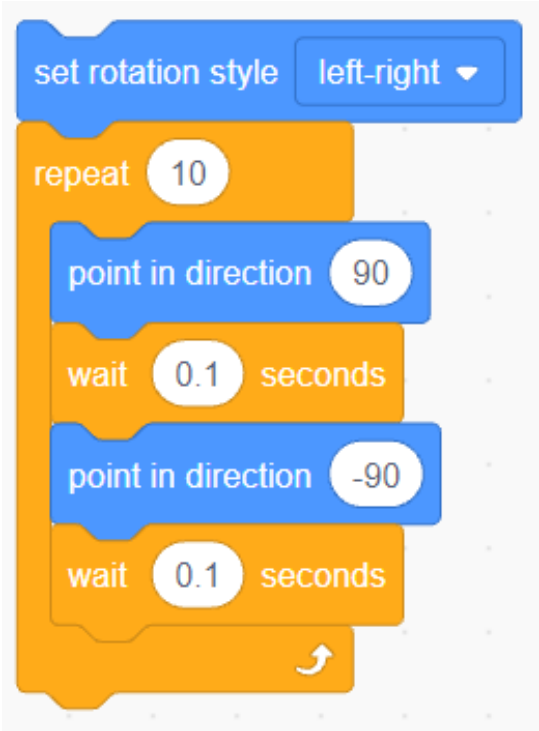



- "repeat x times" loops will run the nested code x times
- "forever" loops will repeatedly run the nested code forever
- "repeat until ..." loops will run the nested code until the Boolean input evaluates to true
  - If the condition is initially true then the nested code is *never* run
  - Ideally, the condition depends on variables that get updated during each iteration (run of the nested code)

# Loops: Doing Stuff Many Times!



# Loops: Doing Stuff Many Times!




- This code will make the sprite "shimmy"

# Functions: Reusing and Organizing Code

- What if I want to run the same code multiple times, but not in a row?

# Functions: Reusing and Organizing Code

- What if I want to run the same code multiple times, but not in a row?
- As programs gets more complicated, reading the code becomes harder if its all in one place

# Functions: Reusing and Organizing Code

- What if I want to run the same code multiple times, but not in a row?
- As programs gets more complicated, reading the code becomes harder if its all in one place
- Functions allow us to reuse and organize our code sensibly!

# Functions: Reusing and Organizing Code

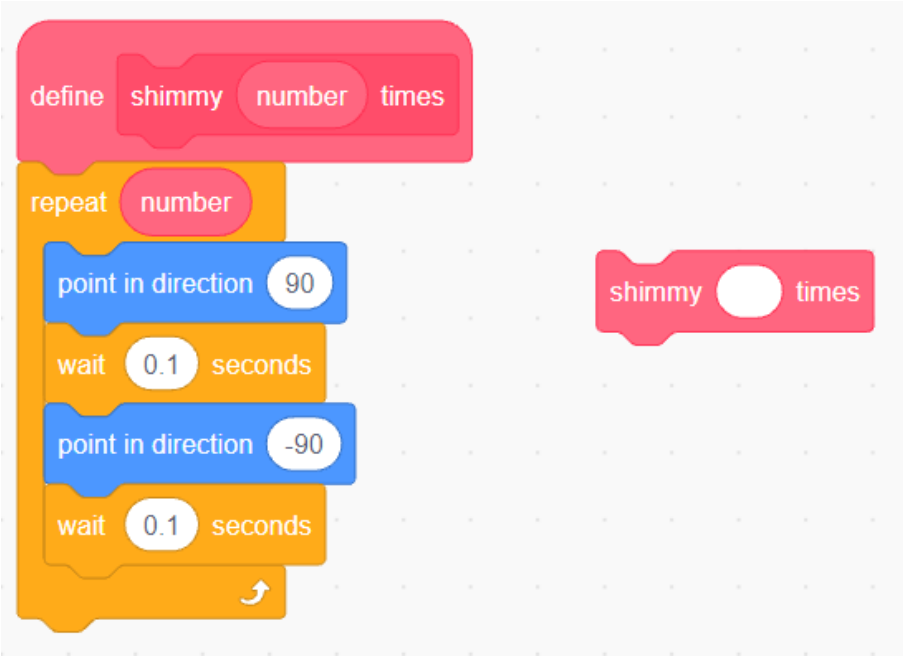
- What if I want to run the same code multiple times, but not in a row?
- As programs gets more complicated, reading the code becomes harder if its all in one place
- Functions allow us to reuse and organize our code sensibly!
  - In Scratch, user created functions are called “My Blocks”

# Functions: Reusing and Organizing Code

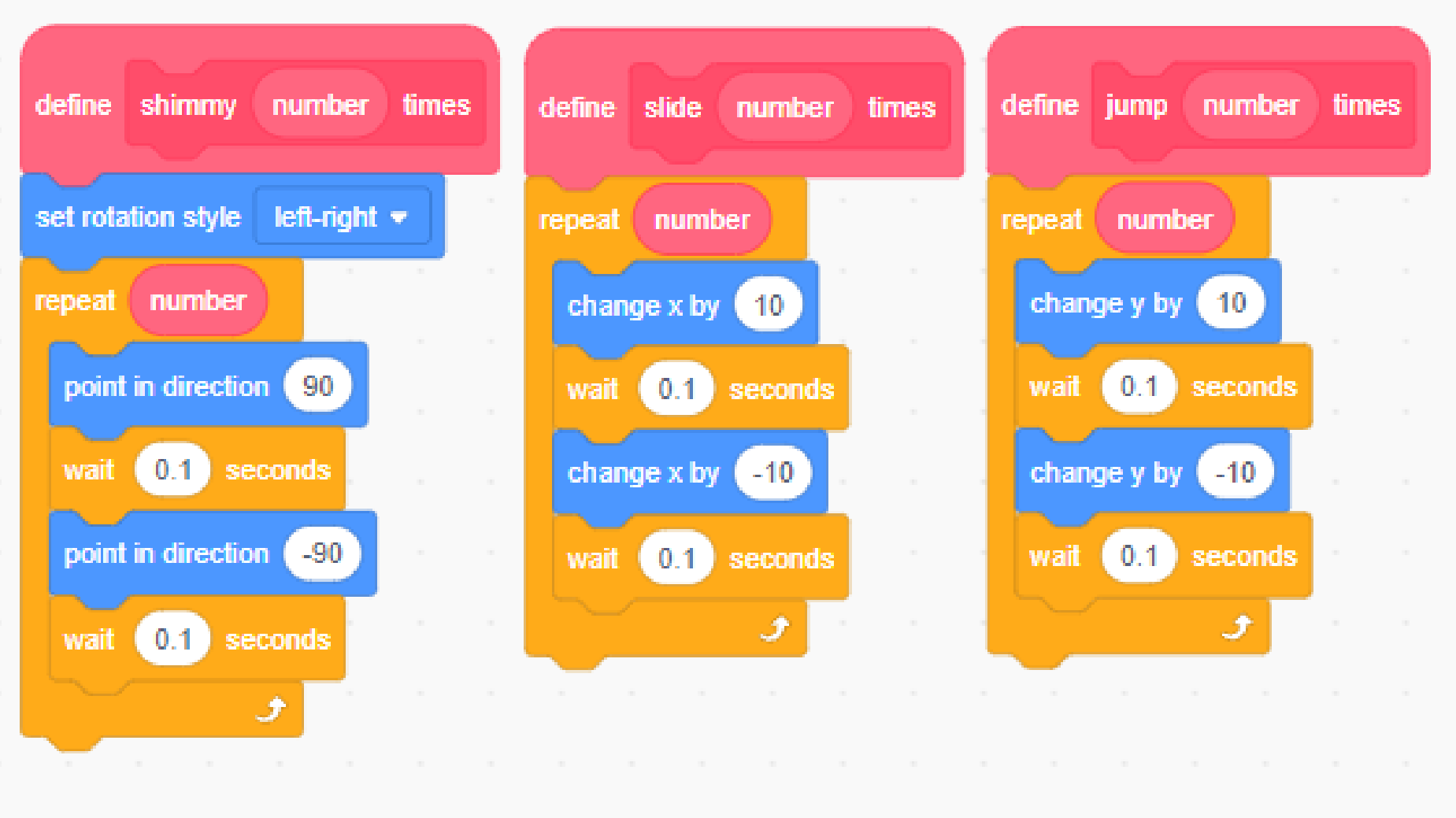
- What if I want to run the same code multiple times, but not in a row?
- As programs gets more complicated, reading the code becomes harder if its all in one place
- Functions allow us to reuse and organize our code sensibly!
  - In Scratch, user created functions are called "My Blocks"
  - Custom Blocks can take inputs which can be called in the code

# Functions: Reusing and Organizing Code

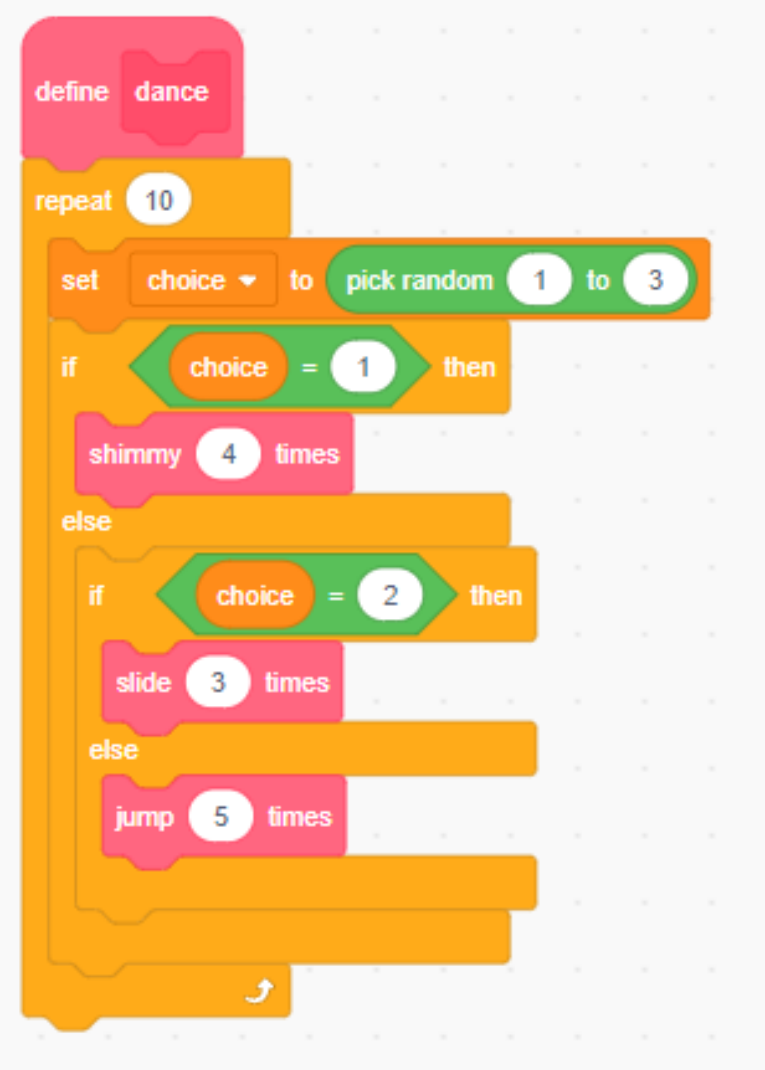
- What if I want to run the same code multiple times, but not in a row?
- As programs gets more complicated, reading the code becomes harder if its all in one place
- Functions allow us to reuse and organize our code sensibly!
  - In Scratch, user created functions are called "My Blocks"
  - Custom Blocks can take inputs which can be called in the code



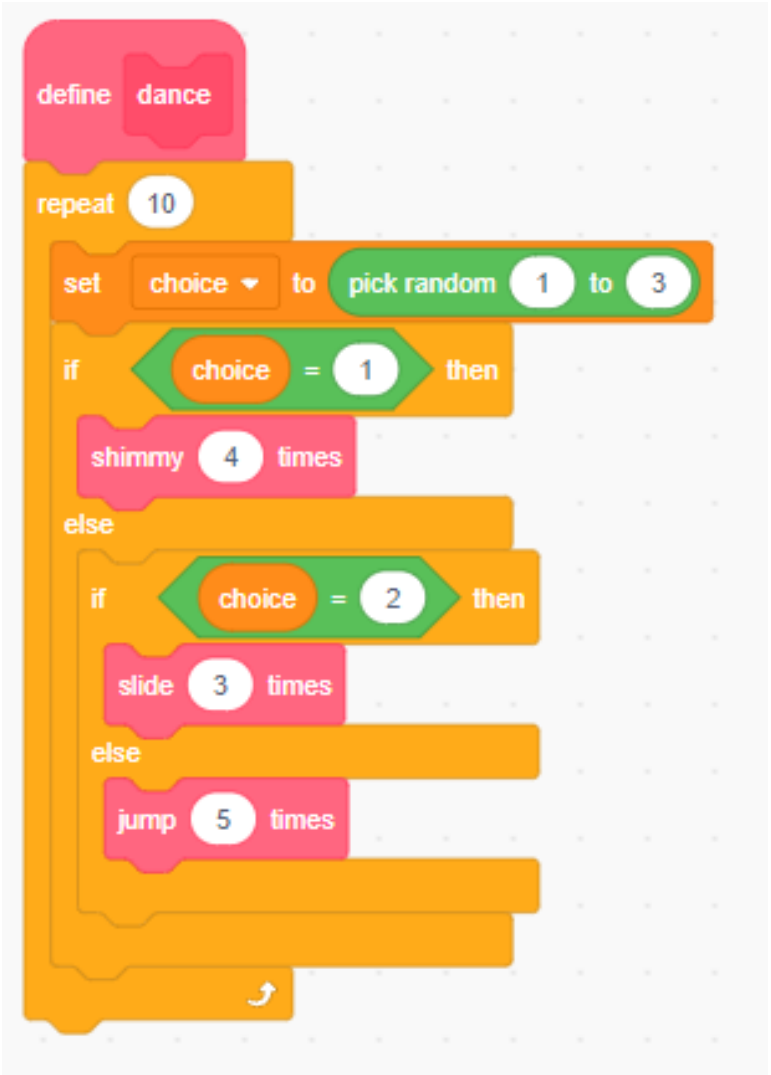
# Functions Example: Dancing

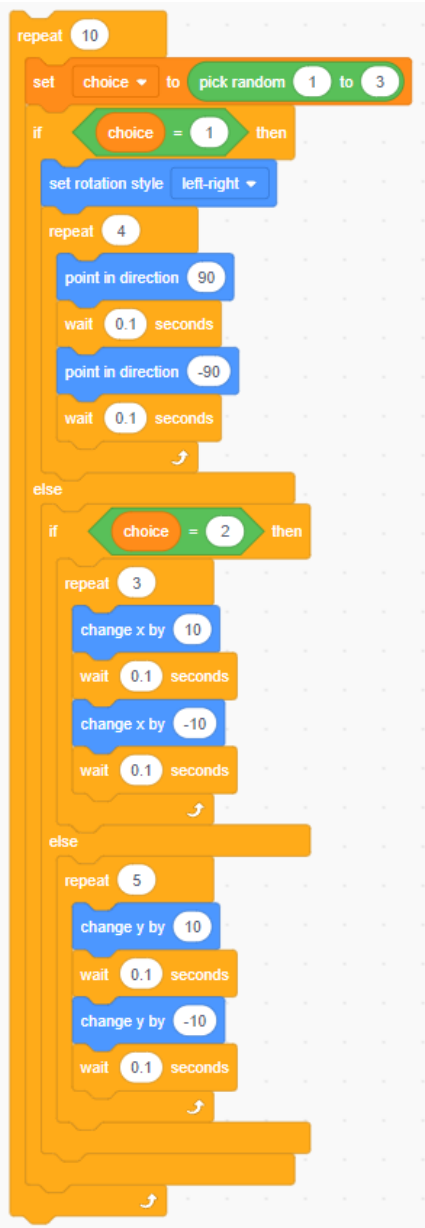


# Functions Example: Dancing



# Functions Example: Dancing



# Functions As Black Boxes

- One powerful thing about functions is we can use them as *black boxes*

# Functions As Black Boxes

- One powerful thing about functions is we can use them as *black boxes*
- A *black box* is a system that is "opaque" (black): we feed inputs into it and it outputs stuff without us knowing *how* it does it

# Functions As Black Boxes

- One powerful thing about functions is we can use them as *black boxes*
- A *black box* is a system that is "opaque" (black): we feed inputs into it and it outputs stuff without us knowing *how* it does it
  - And for most applications, we don't care how it works, we just care that it works.

# Functions As Black Boxes

- One powerful thing about functions is we can use them as *black boxes*
- A *black box* is a system that is "opaque" (black): we feed inputs into it and it outputs stuff without us knowing *how* it does it
  - And for most applications, we don't care how it works, we just care that it works.
- Once we make a function that correctly completes a task, we can reuse it without ever needing to think about its internal workings ever again!

# Functions As Black Boxes

- One powerful thing about functions is we can use them as *black boxes*
- A *black box* is a system that is "opaque" (black): we feed inputs into it and it outputs stuff without us knowing *how* it does it
  - And for most applications, we don't care how it works, we just care that it works.
- Once we make a function that correctly completes a task, we can reuse it without ever needing to think about its internal workings ever again!
- If we're working in a team, other people can use our functions without needing to know how they work!

# Functions As Black Boxes

- One powerful thing about functions is we can use them as *black boxes*
- A *black box* is a system that is "opaque" (black): we feed inputs into it and it outputs stuff without us knowing *how* it does it
  - And for most applications, we don't care how it works, we just care that it works.
- Once we make a function that correctly completes a task, we can reuse it without ever needing to think about its internal workings ever again!
- If we're working in a team, other people can use our functions without needing to know how they work!
- The key to problem solving is to break your problem into sub-problems, solve those subproblems, and then put those solutions together.

# Challenge

Create a program that has a sprite say the numbers 1 to 100, replacing any multiples of 3 with the word Fizz, replacing any multiples of 5 with the word Buzz, and any multiples of both with FizzBuzz.

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz, 16, . . .

(This is an actual "common" job interview question to weed out people who don't know the basics, you can now pass it with what we've seen!)