

Programming 1 / Object Oriented Programming

Thinking Like a Programmer:
Lecture #3 - Boolean logic and
more on functions

What on earth is a "boolean"?

- ⦿ Think back to Carol, and when you were making her make decisions...
- ⦿ ...you used a variety of blocks in conjunction with IF blocks, WHILE blocks and REPEAT/UNTIL blocks...

Carol is blocked?

Carol is not blocked?

Carol is at the goal?

Carol is not at the goal?

Carol can see an item to pick up?

What on earth is a "boolean"?

- ⦿ Think back to Carol, and when you were making her make decisions...
- ⦿ ...you used a variety of blocks in conjunction with IF blocks, WHILE blocks and REPEAT/UNTIL blocks...

Carol is blocked?

Carol is not blocked?

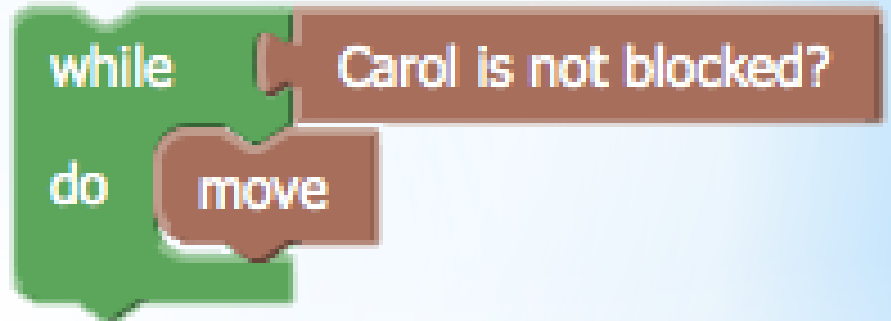
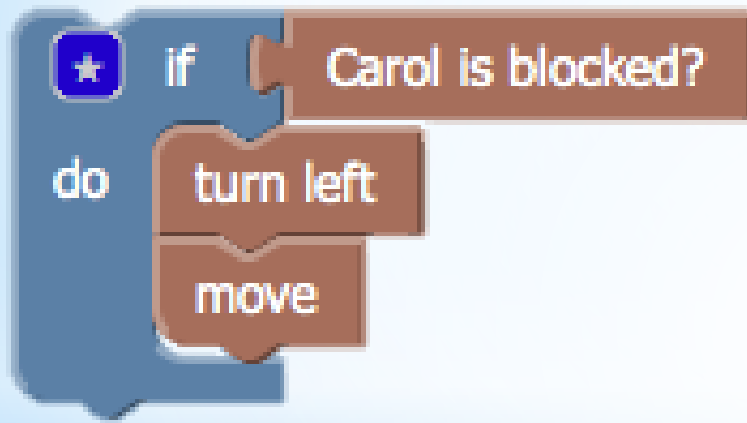
Carol is at the goal?

Carol is not at the goal?

Carol can see an item to pick up?

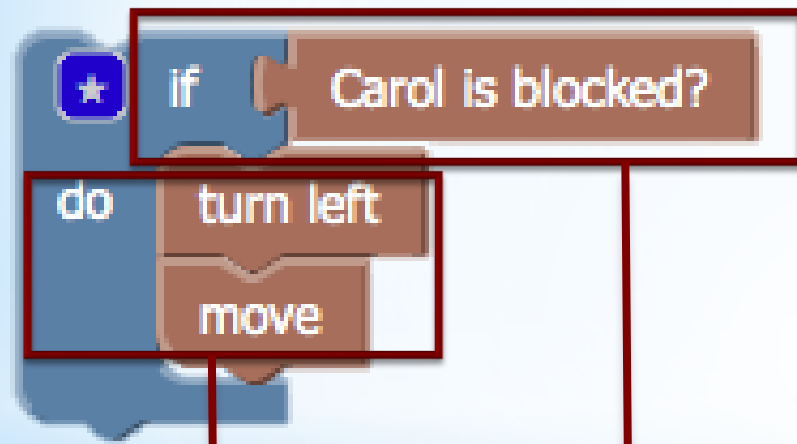
What on earth is a "boolean"?

- Think about how you might have used one of these blocks in the past...



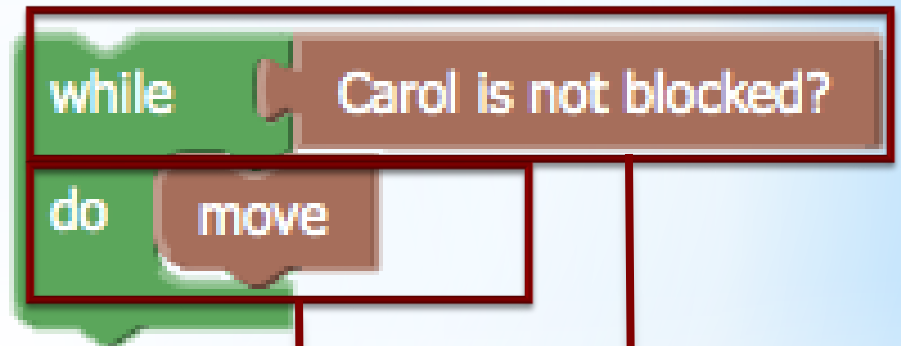
What on earth is a "boolean"?

- Think about how you might have used one of these blocks in the past...



If the answer to this is YES...

...do this



While the answer to this is YES...

...do this

What on earth is a "boolean"?

- ⦿ The blocks you used in conjunction with your IF, WHILE and REPEAT/UNTIL loops were essentially questions with a YES or NO answer
- ⦿ More accurately, they were statements that could be TRUE or FALSE
- ⦿ Anything that results in a true or false answer can be said to have a *boolean* value

Carol is blocked?

Carol is not blocked?

Carol is at the goal?

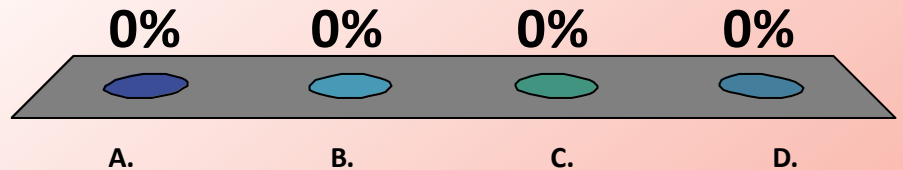
Carol is not at the goal?

Carol can see an item to pick up?

Which of these statements is a Boolean?

- A. The sky is purple with orange polka dots
- B. $2 + 2$
- C. Move forward six squares

D.



Relational operators

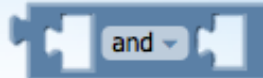
- ⊙ Our Carol boolean functions were pretty simple - e.g. slap them on an IF statement, job done!
- ⊙ In "real world" programming, however, you will often need to compare things and use the result of the comparison for the basis of your decision
- ⊙ For example
 - ⊙ is the player's score more than the current high score?
 - ⊙ is the current temperature less than the thermostat setting?
 - ⊙ is the password that's been typed equal to the user's actual password?

The relational operator block

- ◉ To compare things with each other, you use the relational operator block

- ◉ Look in the **Comparisons** menu for the block



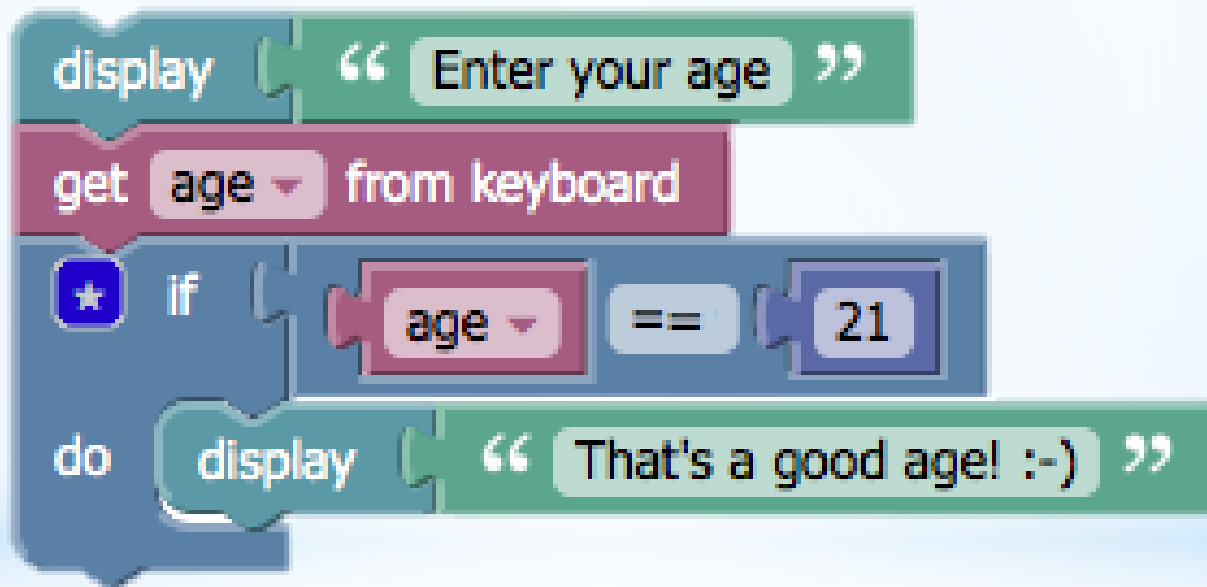
- ◉ The relational operator block looks very similar to the calculation block
- ◉ This is not by accident! The relational operator DOES perform a calculation - one which will result in a TRUE or FALSE (i.e. a boolean) result
- ◉ (It also looks similar to the logical operator block  which is also in the **Comparisons** menu and which we'll discuss later - but in the meantime, do not confuse the two!)

The relational operator block

- ⊙ As with the maths block, you can compare any two things that you like and that will fit in the two "holes"
 - ⊙ as always... if the block fits, you're good to go!
- ⊙ You could compare
 - ⊙ Two numbers
 - ⊙ Two variables
 - ⊙ A number and a variable
 - ⊙ A variable and the result of a calculation block
 - ⊙ note that the calculation block would need its own "holes" filled in!

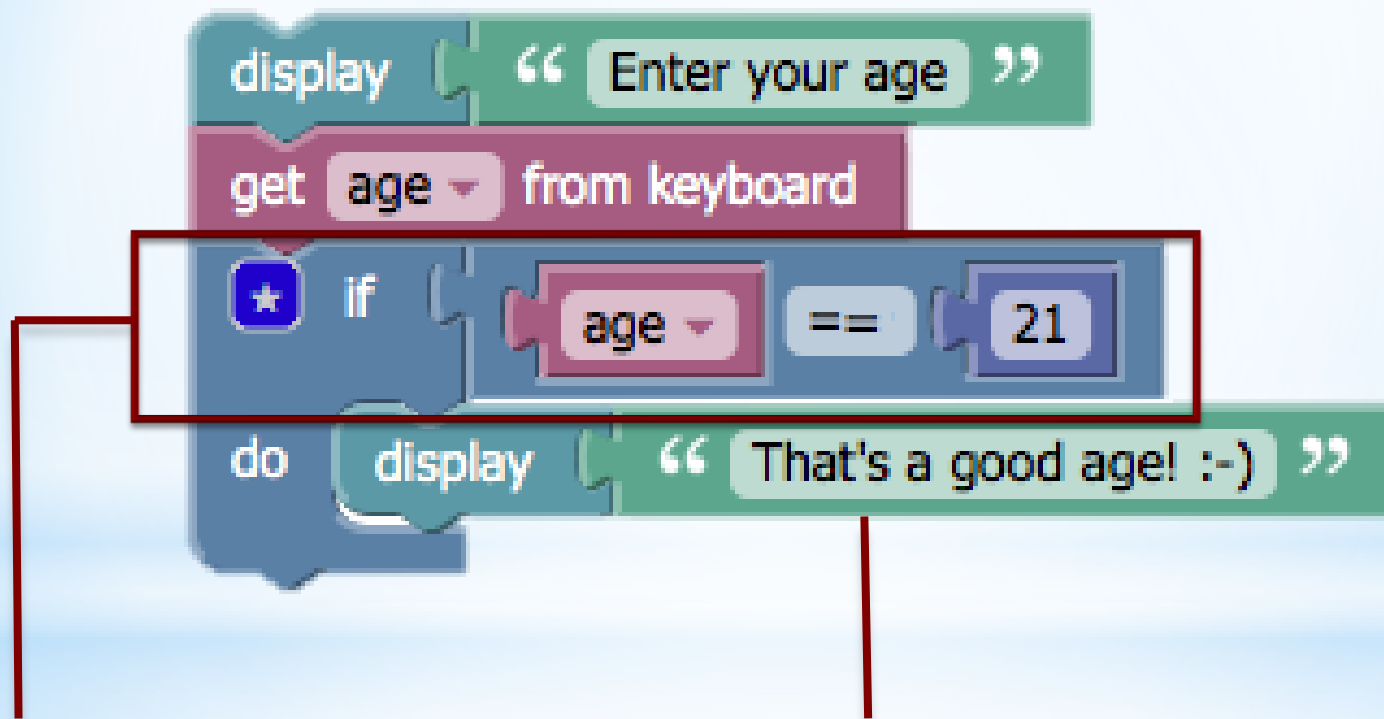
Using the relational operator block

- ◉ We'll explain it with an example:



Using the relational operator block

- ◉ We'll explain it with an example:



If the answer to this is
YES...

or, another way of putting
it: if this is TRUE...

...then do this

Using the relational operator block

- ⦿ A more complex example:



The relational operators

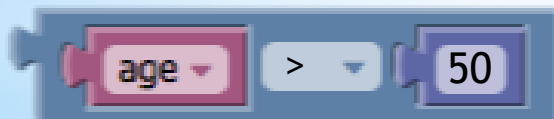
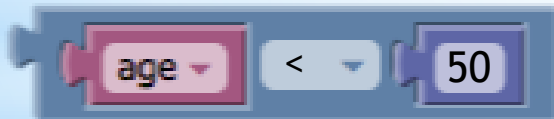
- ⊙ == equal to (note DOUBLE equals sign..!)
- ⊙ != not equal to
- ⊙ > greater than
- ⊙ < less than
- ⊙ >= greater than or equal to
- ⊙ <= less than or equal to

Relational operators result in boolean values

- ⦿ You are the judge! Assuming that we've created a variable and set its value thus:



- ⦿ What will the boolean value (true or false) be for these expressions?

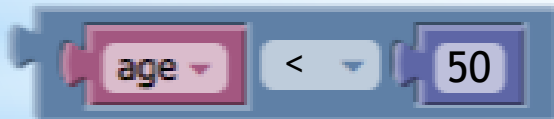


Relational operators result in boolean values

- ⦿ You are the judge! Assuming that we've created a variable and set its value thus:



- ⦿ What will the boolean value (true or false) be for these expressions?



TRUE

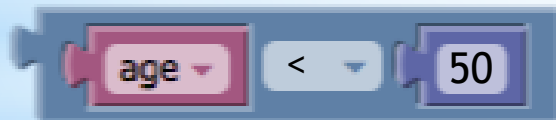


Relational operators result in boolean values

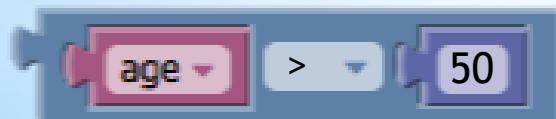
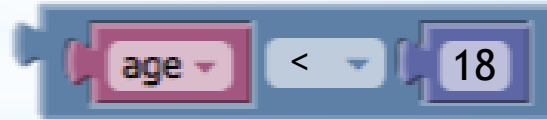
- ⦿ You are the judge! Assuming that we've created a variable and set its value thus:



- ⦿ What will the boolean value (true or false) be for these expressions?



TRUE



FALSE

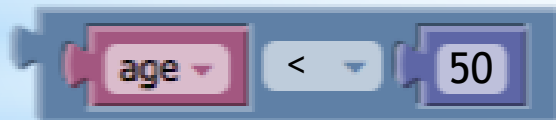


Relational operators result in boolean values

- ⦿ You are the judge! Assuming that we've created a variable and set its value thus:



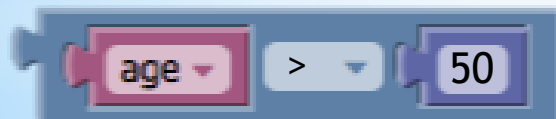
- ⦿ What will the boolean value (true or false) be for these expressions?



TRUE



FALSE



FALSE



Relational operators result in boolean values

- ⦿ You are the judge! Assuming that we've created a variable and set its value thus:



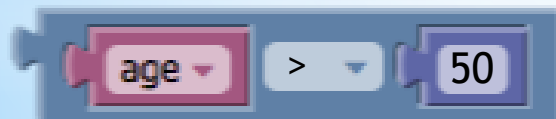
- ⦿ What will the boolean value (true or false) be for these expressions?



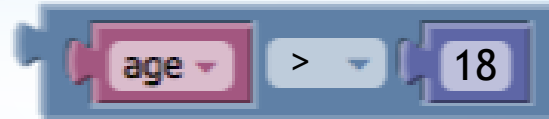
TRUE



FALSE



FALSE



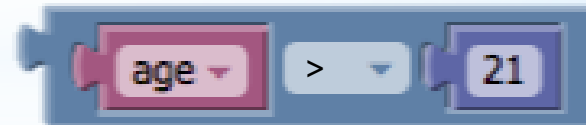
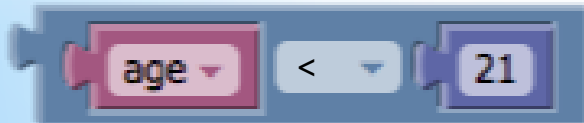
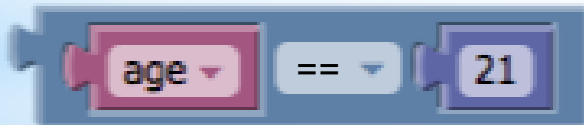
TRUE

Relational operators result in boolean values

- ⦿ You are the judge! Assuming that we've created a variable and set its value thus:



- ⦿ What will the boolean value (true or false) be for these expressions?



Relational operators result in boolean values

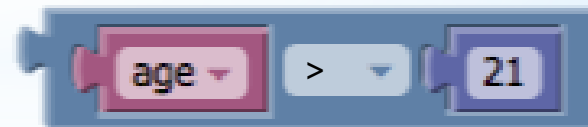
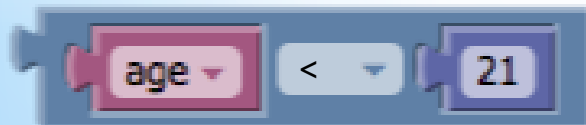
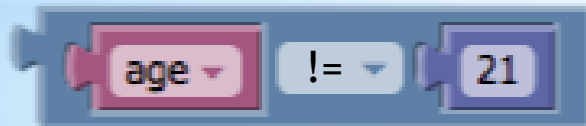
- ⦿ You are the judge! Assuming that we've created a variable and set its value thus:



- ⦿ What will the boolean value (true or false) be for these expressions?



TRUE

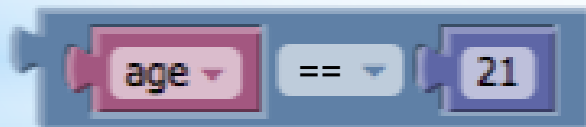


Relational operators result in boolean values

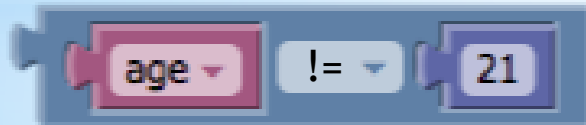
- ⦿ You are the judge! Assuming that we've created a variable and set its value thus:



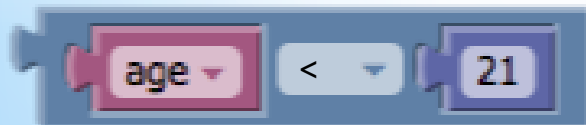
- ⦿ What will the boolean value (true or false) be for these expressions?



TRUE



FALSE

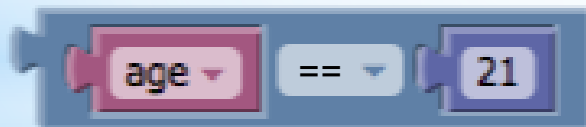


Relational operators result in boolean values

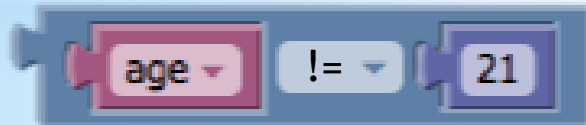
- ⦿ You are the judge! Assuming that we've created a variable and set its value thus:



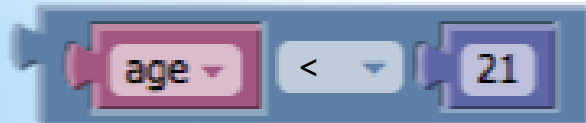
- ⦿ What will the boolean value (true or false) be for these expressions?



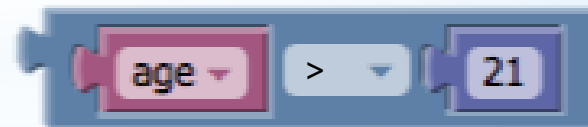
TRUE



FALSE



FALSE



Relational operators result in boolean values

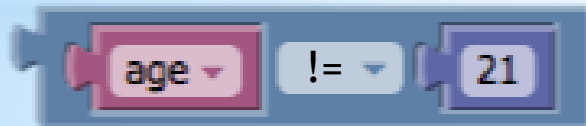
- ⦿ You are the judge! Assuming that we've created a variable and set its value thus:



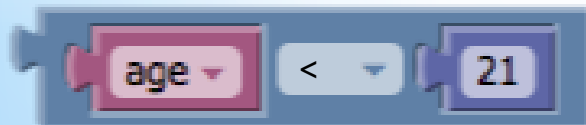
- ⦿ What will the boolean value (true or false) be for these expressions?



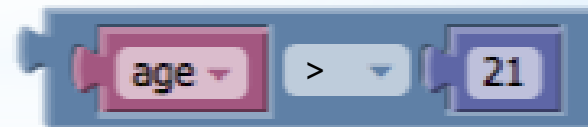
TRUE



FALSE



FALSE



FALSE

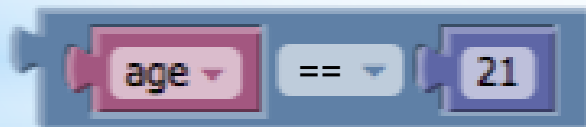


Relational operators result in boolean values

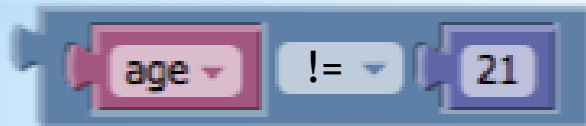
- ⦿ You are the judge! Assuming that we've created a variable and set its value thus:



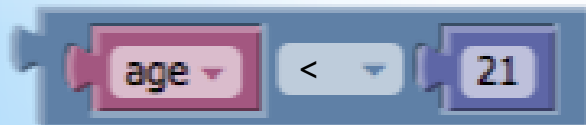
- ⦿ What will the boolean value (true or false) be for these expressions?



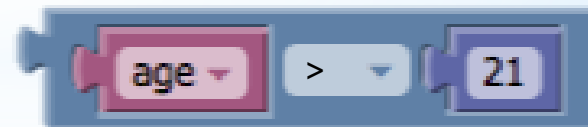
TRUE



FALSE



FALSE



FALSE



TRUE

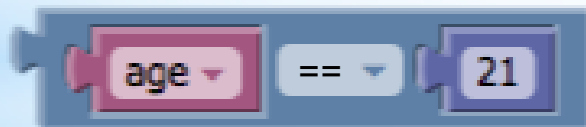


Relational operators result in boolean values

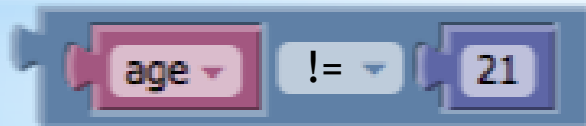
- ⦿ You are the judge! Assuming that we've created a variable and set its value thus:



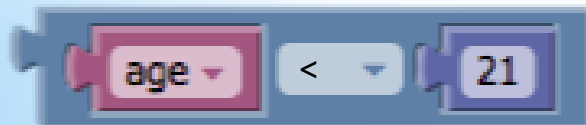
- ⦿ What will the boolean value (true or false) be for these expressions?



TRUE



FALSE



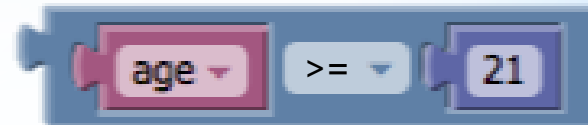
FALSE



FALSE



TRUE



TRUE

Why the double equals for equality?

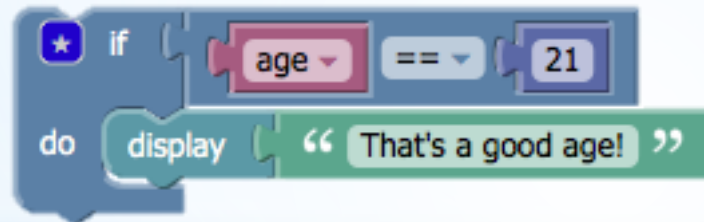
- Consider the code for this block



- The code equivalent would be

```
set age = 21
```

- Now consider the code for this IF block



- The code equivalent would be

```
if (age == 21)
```

```
    display "That's a good age!"
```

```
endif
```

- Why the single equals in the `set` and the double in the `if`?

Why the double equals for equality

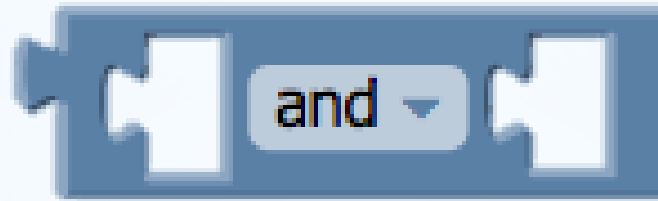
- ⦿ There is a difference between single and double equals
- ⦿ Single equals means *make the thing on the left equal to the thing on the right*
- ⦿ Double equals means *is the thing on the left equal to the thing on the right?*
- ⦿ So, a rule of thumb is
 - ⦿ if you're setting something, use single equals
 - ⦿ if you're making a decision, use double equals
- ⦿ This will be important when we turn off the blocks and you have to write text-based code!
- ⦿ **"S for Setting and Single equals,**
- ⦿ **D for Decisions and Double equals"**

Logical operators

- ⦿ Sometimes, you may need to combine several conditions together
 - ⦿ If the temperature is more than the thermostat setting OR the person has turned the central heating off, turn off the radiator
 - ⦿ If the username typed in AND the password typed in matches the user's registered details, allow access to the bank account
 - ⦿ How about, if the age is between 16 and 18 (e.g. for a college student travelcard?)
 - ⦿ i.e. if the age is more than or equal to 16 AND less than or equal to 18
 - ⦿ Or more correctly
 - ⦿ If the age is more than or equal to 16 AND the age is less than or equal to 18
- ⦿ This is where logical operators come in

The logical operator block

- Look in the **Comparisons** menu



- Each hole in the block can be filled with any block or combination of linked blocks that result in a boolean value (i.e. any combination that results in either TRUE or FALSE)
- NB. Try not to mix up this block with the logical operator block !

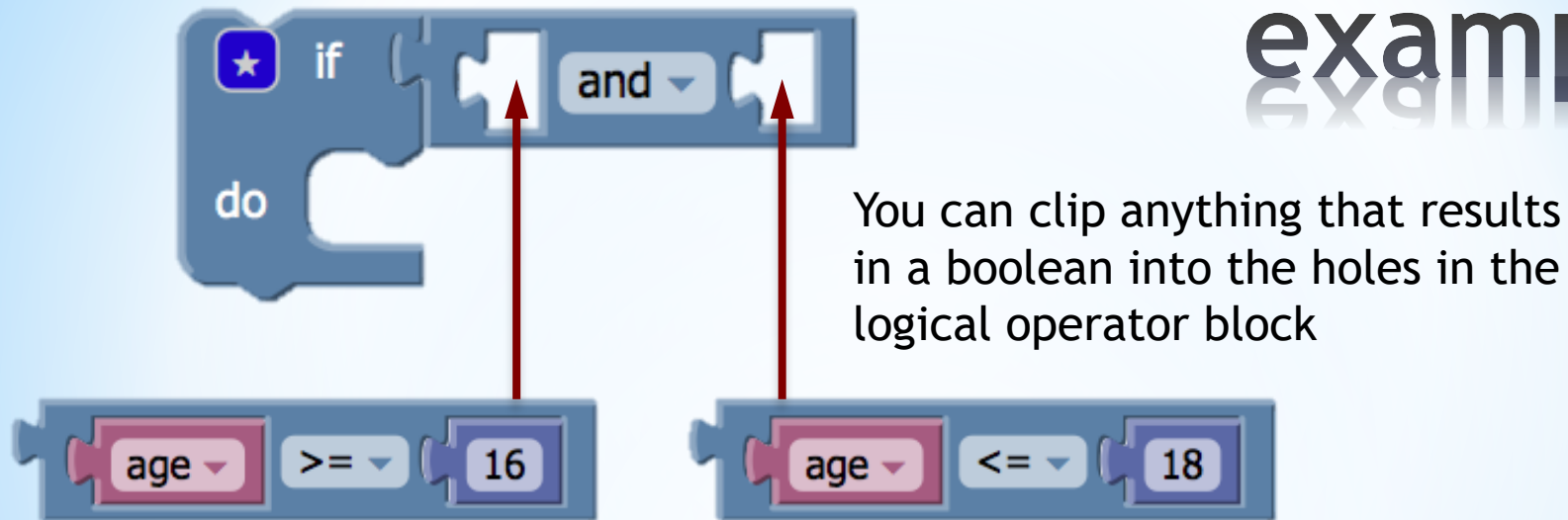
Logical operators - an example

- ◉ Sticking with the example of the college student travelcard...
- ◉ if the age is between 16 and 18 (e.g. for a college student travelcard?)
 - ◉ i.e. if the age is more than or equal to 16 AND less than or equal to 18
- ◉ So, you might create two relational blocks to check for the two conditions:

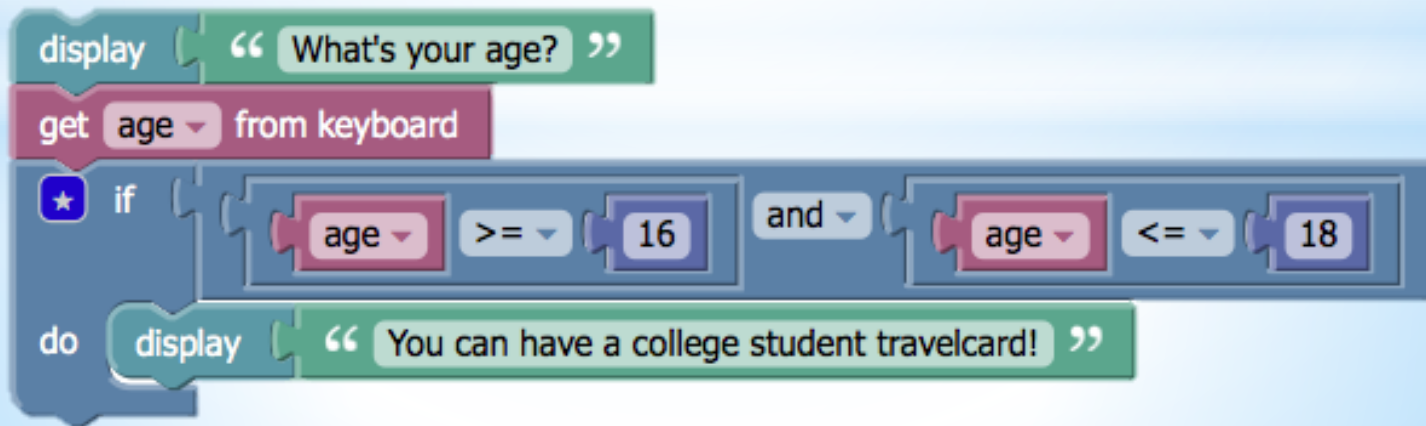


- ◉ Both of these conditions result in a boolean value - either TRUE or FALSE
- ◉ Therefore, these conditions will fit in the "holes" of the logical operator block

Logical operators - an example



So for our college travelcard example, we might end up with something like



AND and OR

- ⦿ If you use AND, both sides of the logical operator must be true
- ⦿ If you use OR, one side of the logical operator must be true
- ⦿ Example scenario 1:
 - ⦿ If the gender is female AND the age is more than 18...
- ⦿ Example scenario 2:
 - ⦿ If the gender is female OR the age is more than 18...
- ⦿ Test data:

⦿ Kevin, male, age 27	Basil, male, age 12
⦿ Susan, female, age 21	Robert, male, age 35
⦿ Jill, female, age 14	Alison, female, age 37

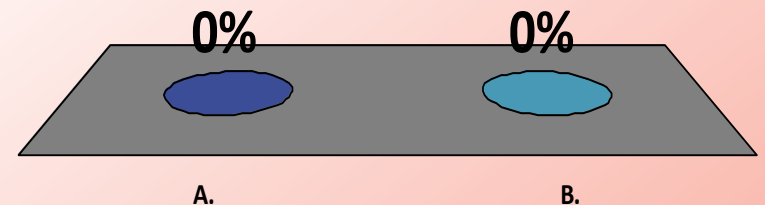
AND and OR

- ⦿ The moral of the story:
 - ⦿ Don't mix up your ANDs and your ORs - you could get yourself into *serious* trouble 😊
- ⦿ Note the text symbols for AND and OR
 - ⦿ && AND
 - ⦿ || OR
 - ⦿ | is called the *pipe* symbol - on British keyboards, it's usually on the key next to the left shift key with the backslash symbol, and you'll need to press shift to get it

What would be printed for a 62 year old man, given the program below?

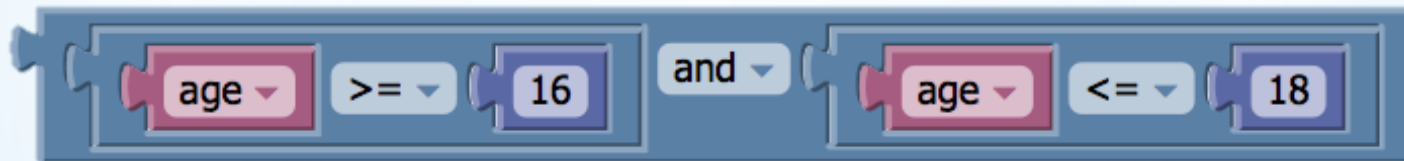
```
display "How old are you?"
get age from keyboard
display "Are you male or female?"
display "Type 'm' for male and 'f' for female."
get gender from keyboard
if (age >= 60 and gender == "m")
do
display "Go and collect your pension!"
else if (age >= 65 and gender == "f")
do
display "Go and collect your pension!"
else
display "Keep on being a wage slave!"
```

- A. Keep on being a wage slave
- B. Go and collect your pension



Nesting logical operators

- ⦿ The logical operator block itself results in a boolean TRUE or FALSE
- ⦿ Let's say AGE was 17...



TRUE

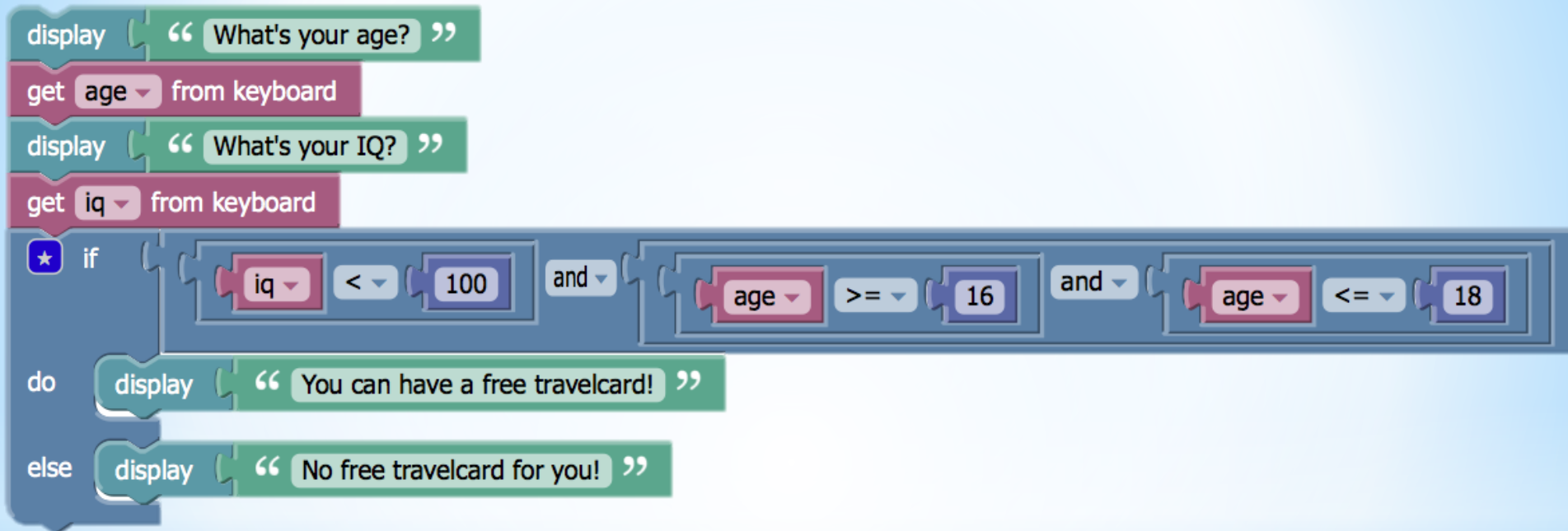
and

TRUE

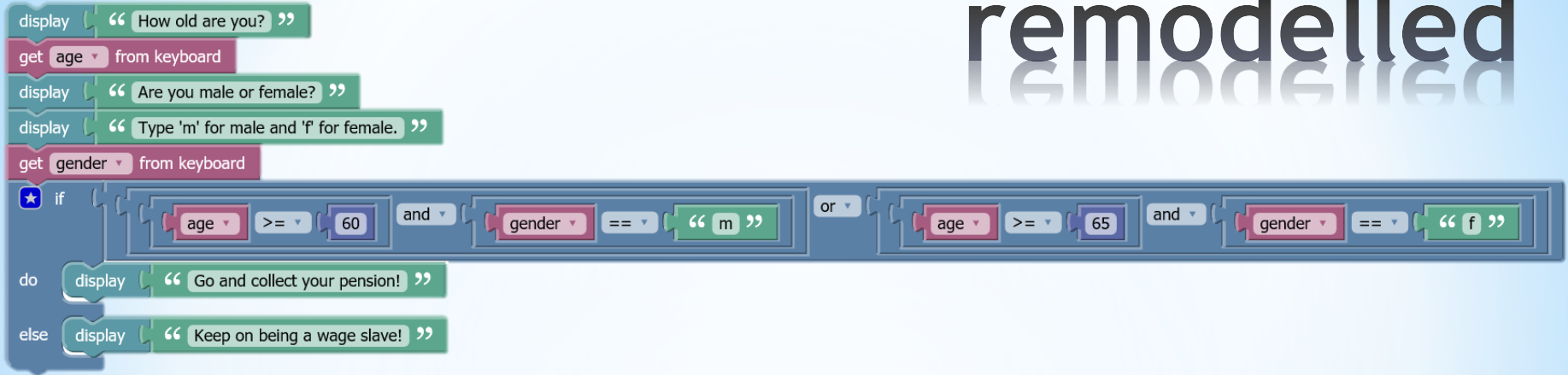
= TRUE

- ⦿ So, you can place a logical operator into one of the holes of another logical operator to make "monster" ones just as we did with the calculation operator previously

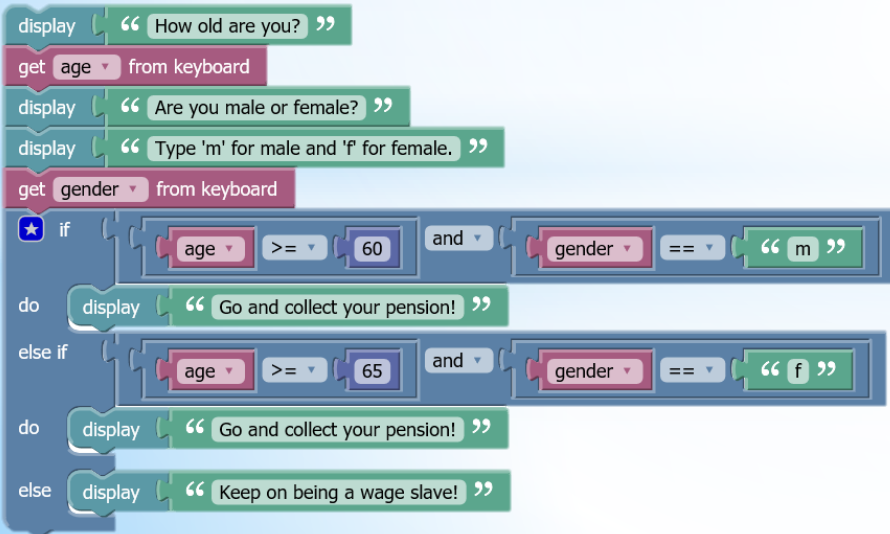
Nesting logical operators



Or, our earlier example remodelled

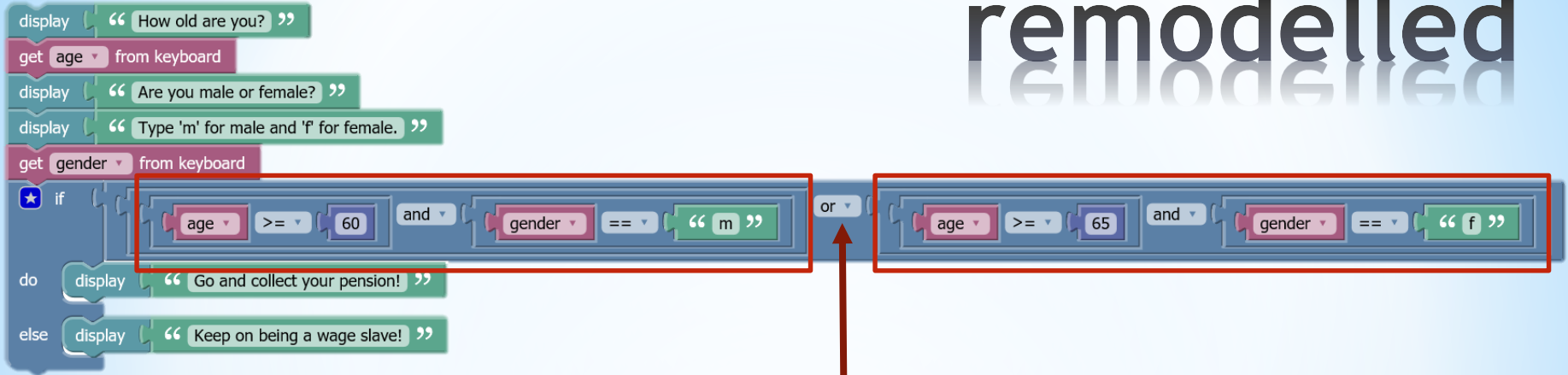


versus

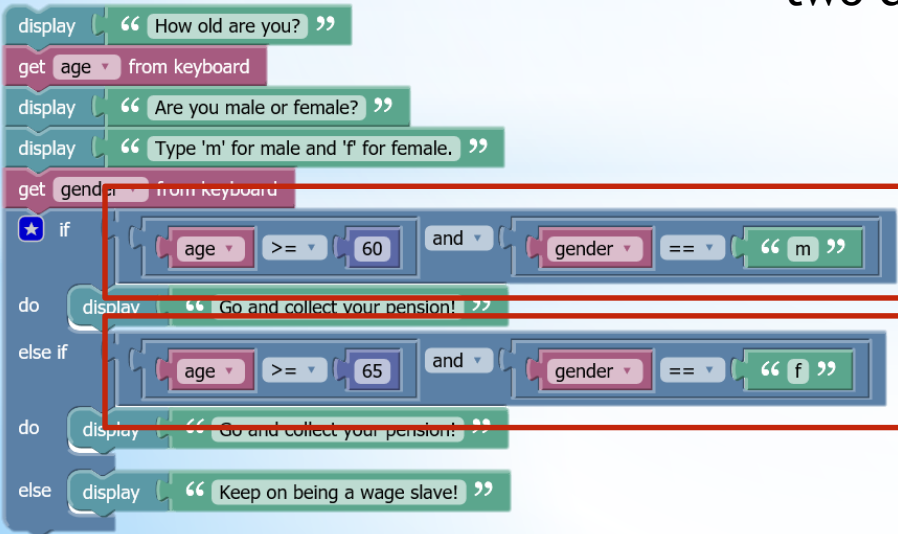


The top version is more efficient, although the bottom version might be preferable in some cases, because it might be easier to read/understand (even if there is some duplication)

Or, our earlier example remodelled



versus



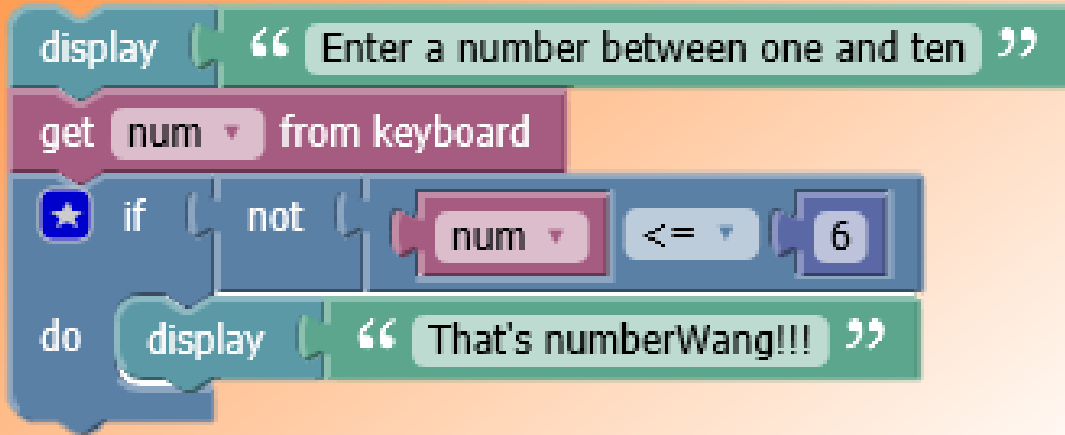
Here, the OR operator lets us combine the two discrete conditions

The top version is more efficient, although the bottom version might be preferable in some cases, because it might be easier to read/understand (even if there is some duplication)

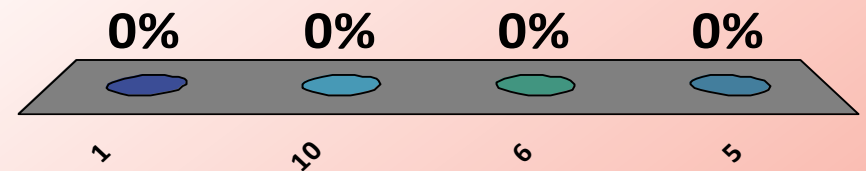
NOT

- ⦿ The NOT operator takes a boolean value and reverses it
 - ⦿ if it was originally TRUE, it makes it FALSE
 - ⦿ if it was originally FALSE, it makes it TRUE

What number will result in the message “That’s numberWang!!!”



- A. 1
- B. 10
- C. 6
- D. 5



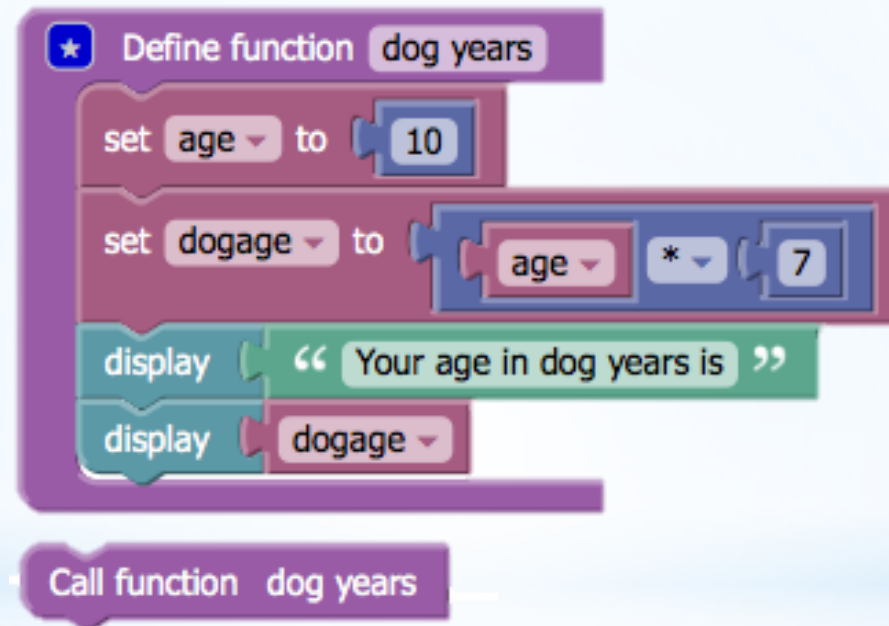
AND NOW
FOR SOMETHING
COMPLETELY
DIFFERENT

More on functions

- ⦿ When we worked with Carol we wrote functions for things that had to be done often within a given context
 - ⦿ e.g. "turn right", "move until blocked", "move 4 spaces" etc
- ⦿ Programs for the real world are no different
- ⦿ You will often need functions in real world programs too
- ⦿ Let's go back to the example from a while back where we calculated ages in dog years
 - ⦿ 7 dog years is 1 human year - so to get an age in dog years we multiply it by 7


More on functions

- So consider the following program, which makes use of a function to calculate and display the dog years:

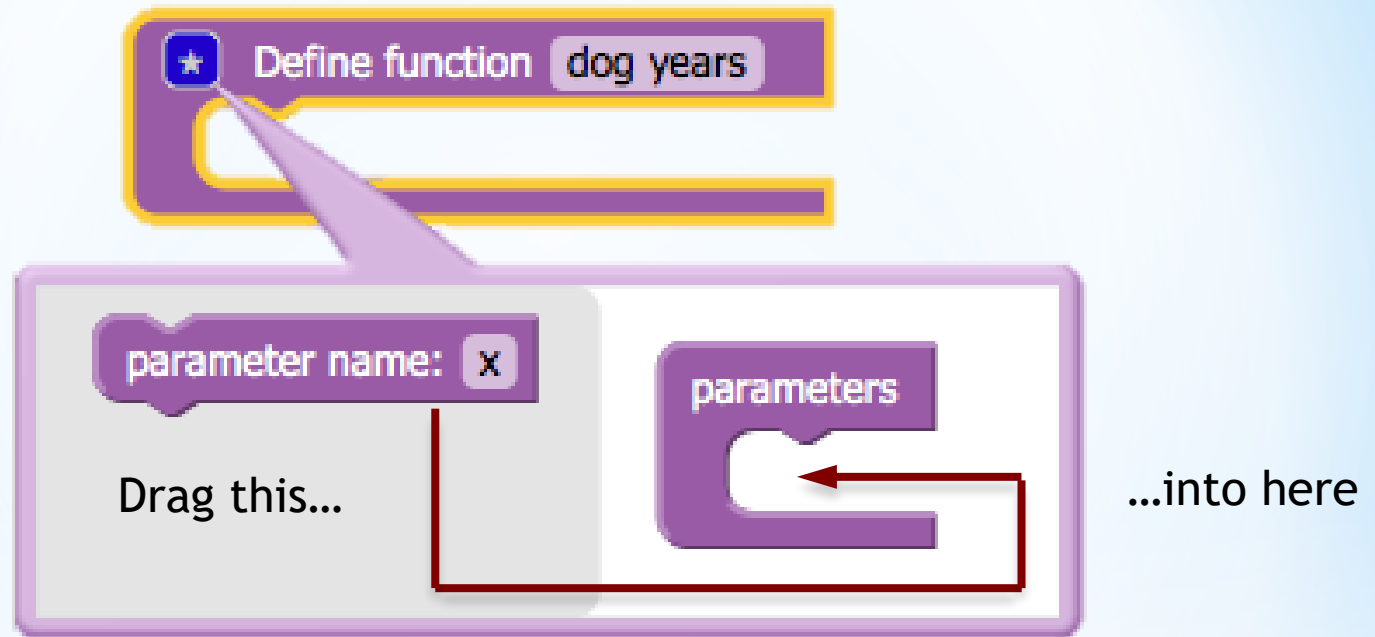


- What about if we want to change the age?
- What about if we want to give a different age every time we call the function?

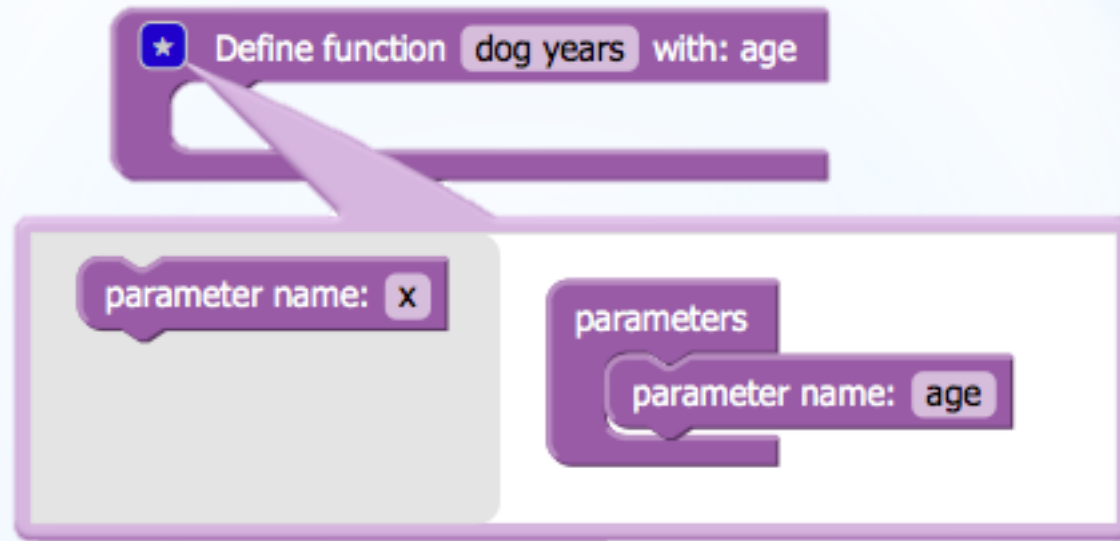
Function parameters

- ⦿ *Parameters* let us send information into the function from outside
- ⦿ So, in our example, we could give the age as a parameter
- ⦿ To specify that a function accepts parameter(s), click the star  at the top of the function block

Function parameters



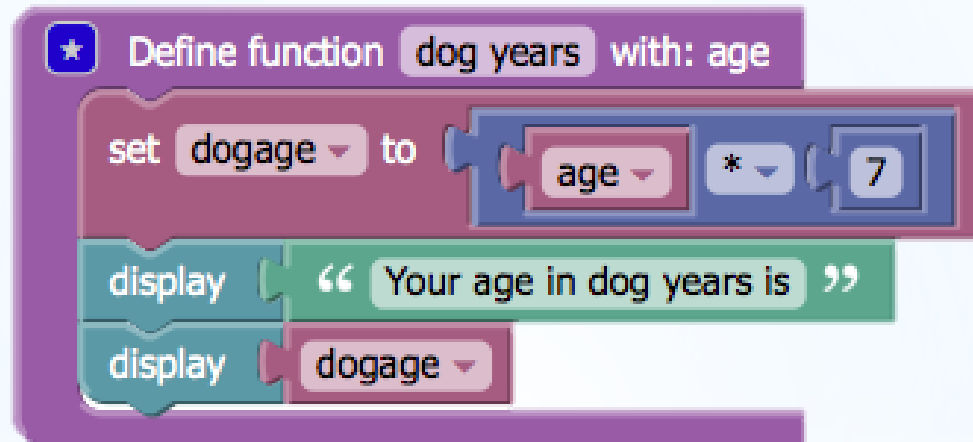
Function parameters



- Click on the **x** on the right hand side and type to give your parameter a sensible name
- In our case, we are going to "transmit" the age of the person into the function - so *age* seems like a good idea for a name!
- Note how the top of the function block changes to indicate that there's a parameter on our function

Function parameters

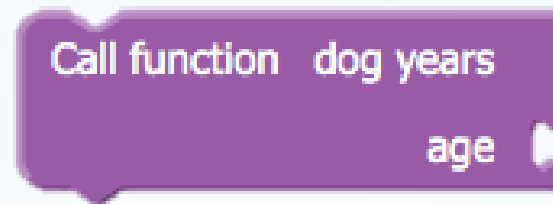
- Here's a full version of our previous function using a parameter for the age



- Note that we don't have to declare **age** as a variable - we can use parameters in the same way as we do variables for the duration of the function

Calling a function with a parameter

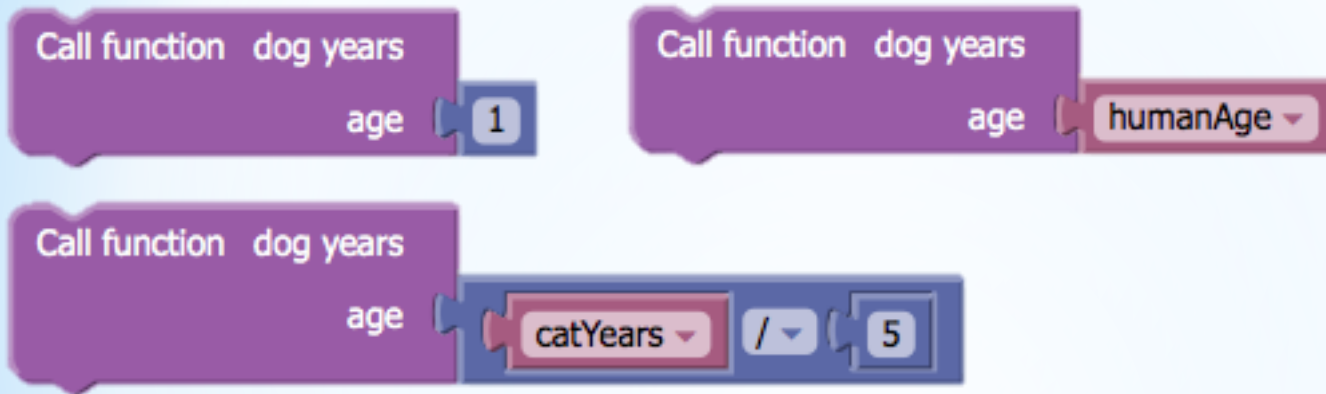
- ◉ When a function has a parameter, you'll notice that the block for calling it changes shape slightly:



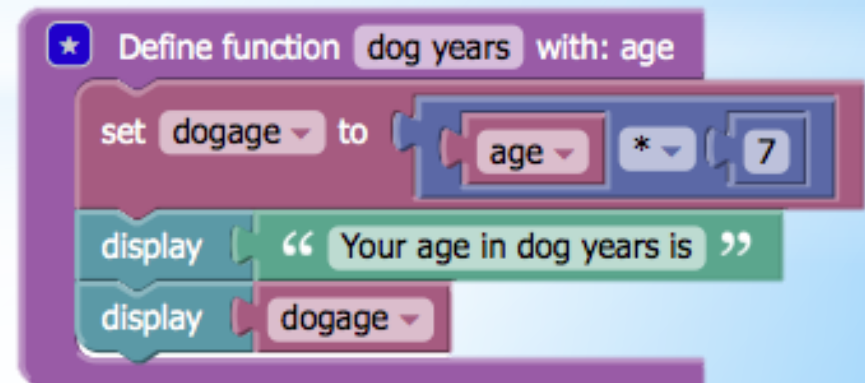
- ◉ Each parameter will be listed with a slot to clip on another block
- ◉ You must clip something to each parameter slot
- ◉ Whatever you clip against a parameter, when the function is run the parameter will contain that value

Calling a function from a parameter

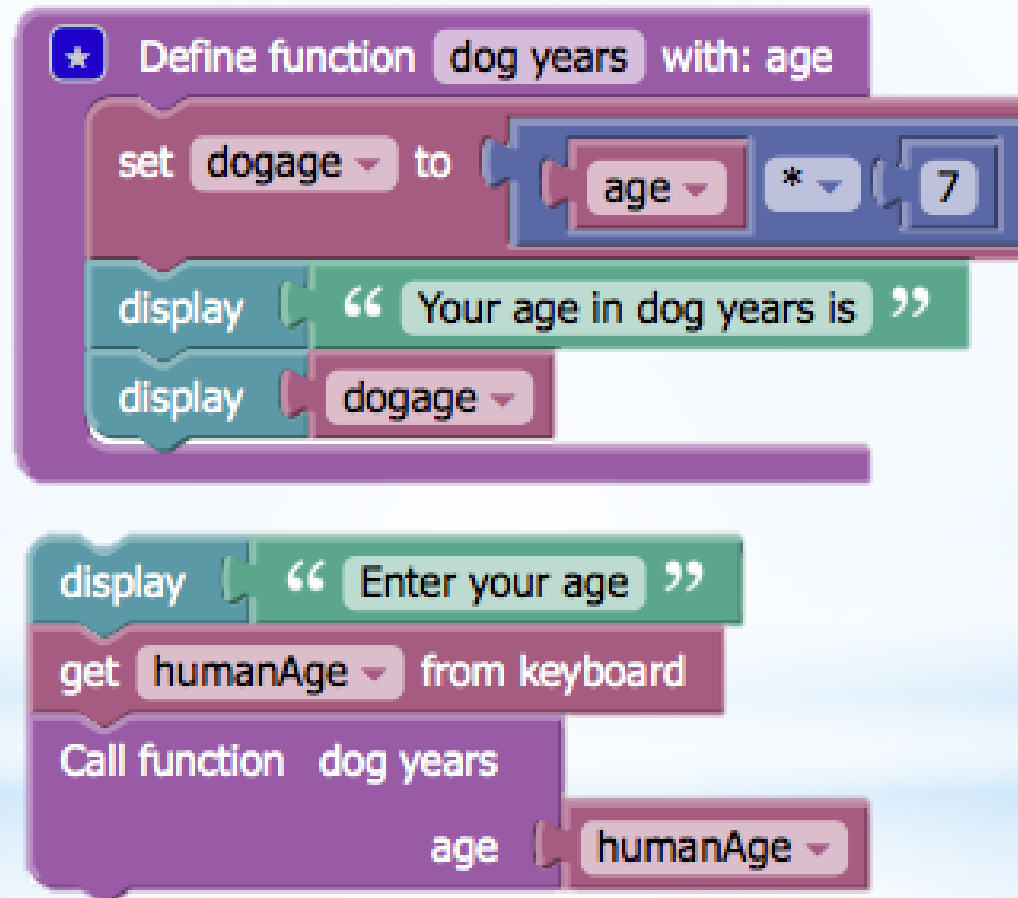
- For example:



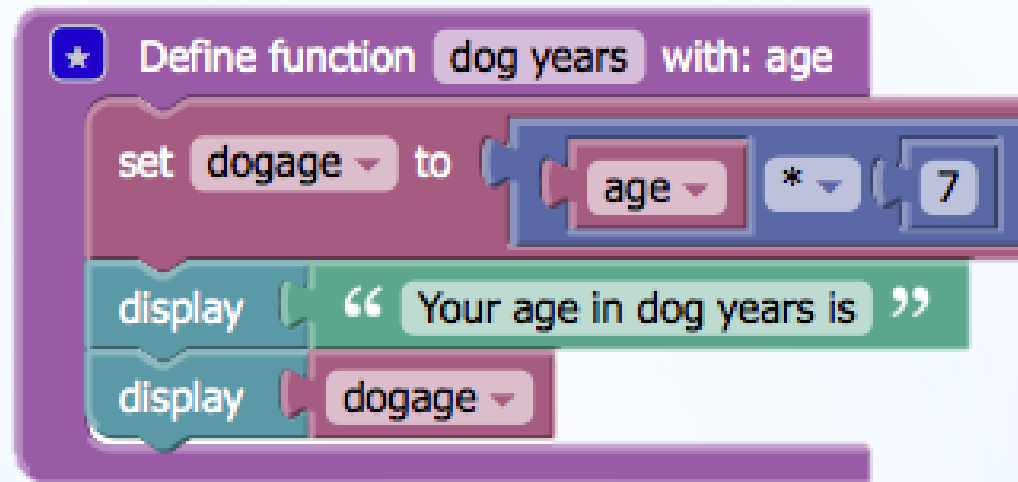
- Remember: if the block fits, you can use it! 😊



A full example

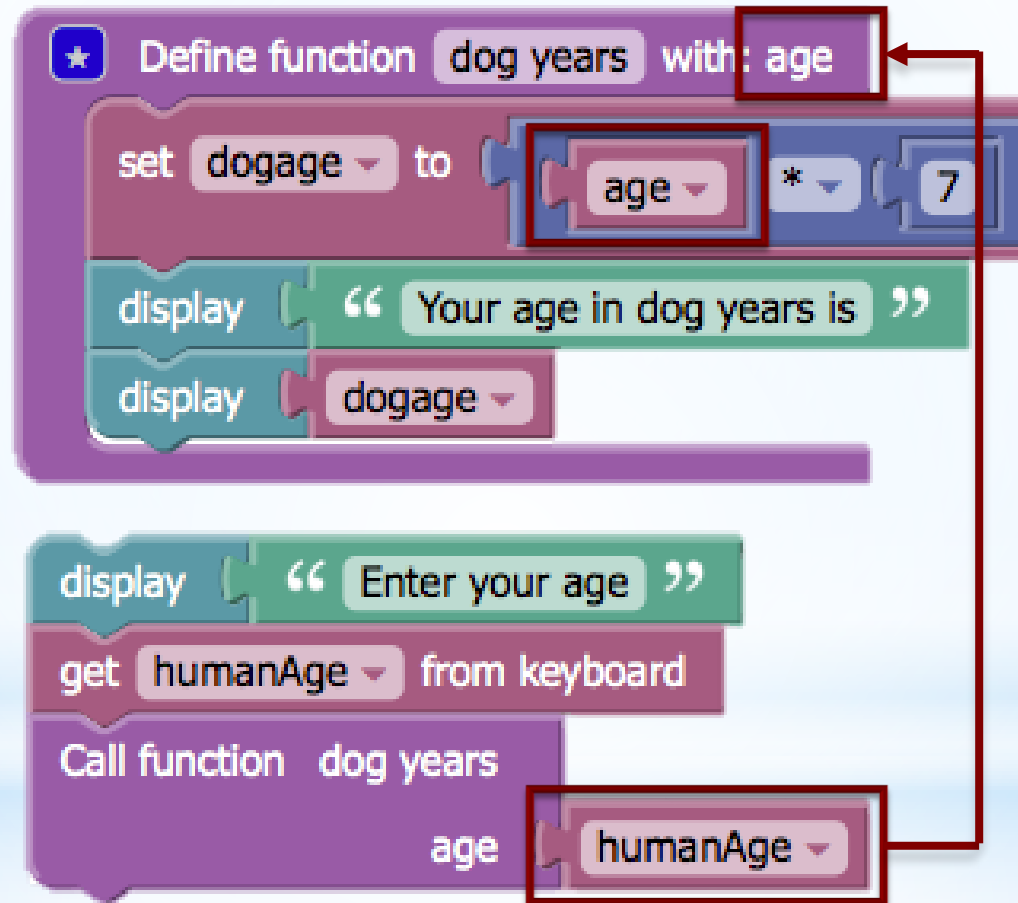


A full example



whatever is
in here

A full example



...gets sent
to here

Function parameters: just remember

- ⦿ A function parameter allows the bit of code that calls that function to also send some addition data into the function

⦿ This has nothing to do with getting input from the keyboard!

Function parameters: just remember

- ⦿ A function parameter allows the bit of code that calls that function to also send some additional data into the function

⦿ This has nothing to do with getting input from the keyboard!

- ⦿ If you have an exercise that asks you to write a function that takes two parameters, and you are using the GET block in it, you probably have misunderstood the exercise...

Return values - sending things back from functions

- ⦿ Return values are the "opposite" of parameters
 - ⦿ Just as a parameter lets you send a value *into* a function, a return value lets you send a value back from the function
- ⦿ **THIS IS VERY VERY DIFFERENT FROM PRINTING THE RESULT TO THE SCREEN!**

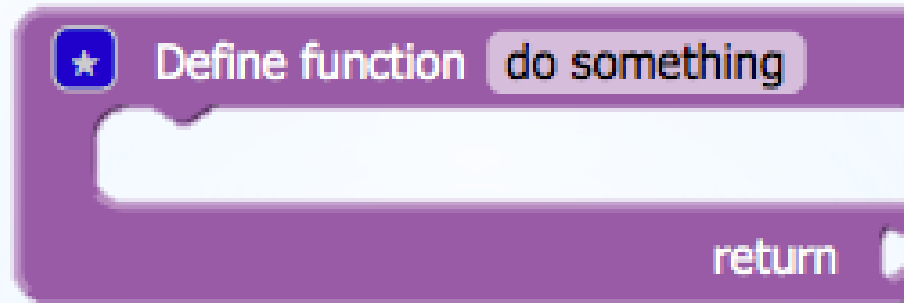
Return values - sending things back from functions

- ⦿ Return values are the "opposite" of parameters
 - ⦿ Just as a parameter lets you send a value *into* a function, a return value lets you send a value back from the function

**⦿ THIS IS VERY VERY
DIFFERENT FROM
PRINTING THE RESULT TO
THE SCREEN!**

Return value example

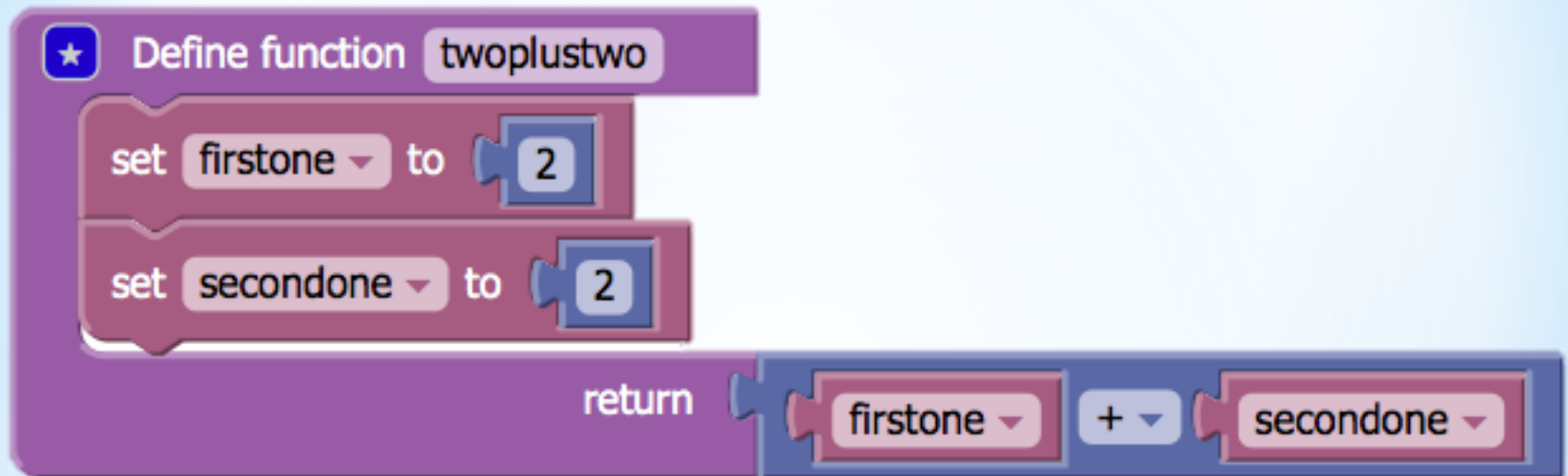
- From **Functions**, choose this block:



- This function block works exactly like the one we've been using previously:
 - Click the **do something** text and type to give the function a name
 - Clip the blocks you want to happen when the function is called inside the function block
 - BUT - you also need to clip something onto the **return** slot

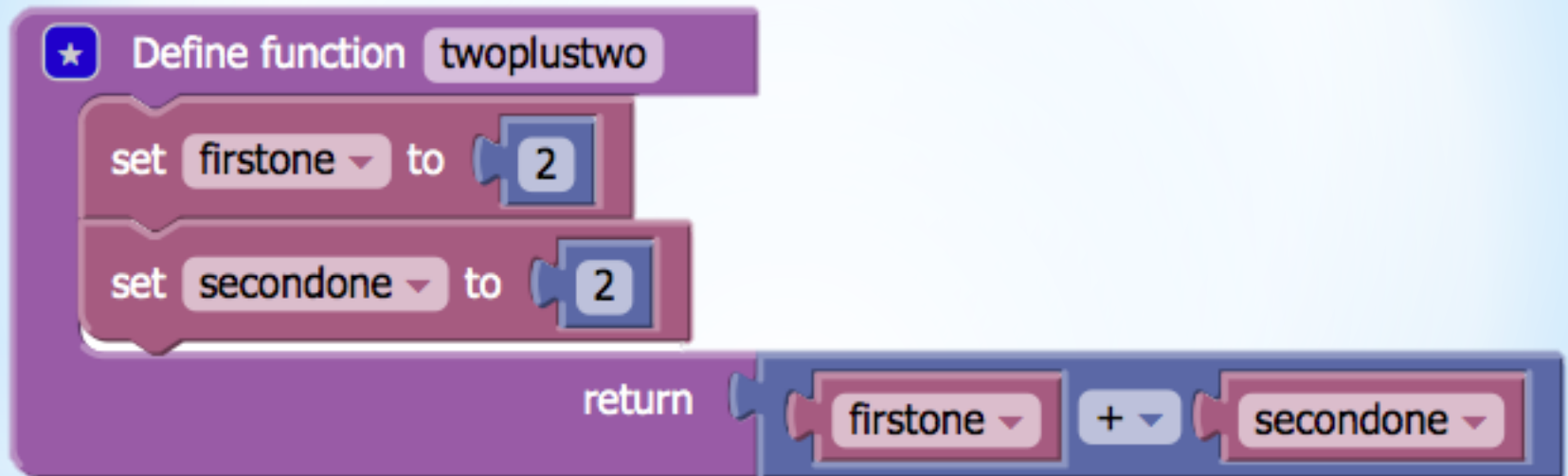
A simple example

- Assuming this is a complete program, what would be printed to the screen? Why?



A simple example

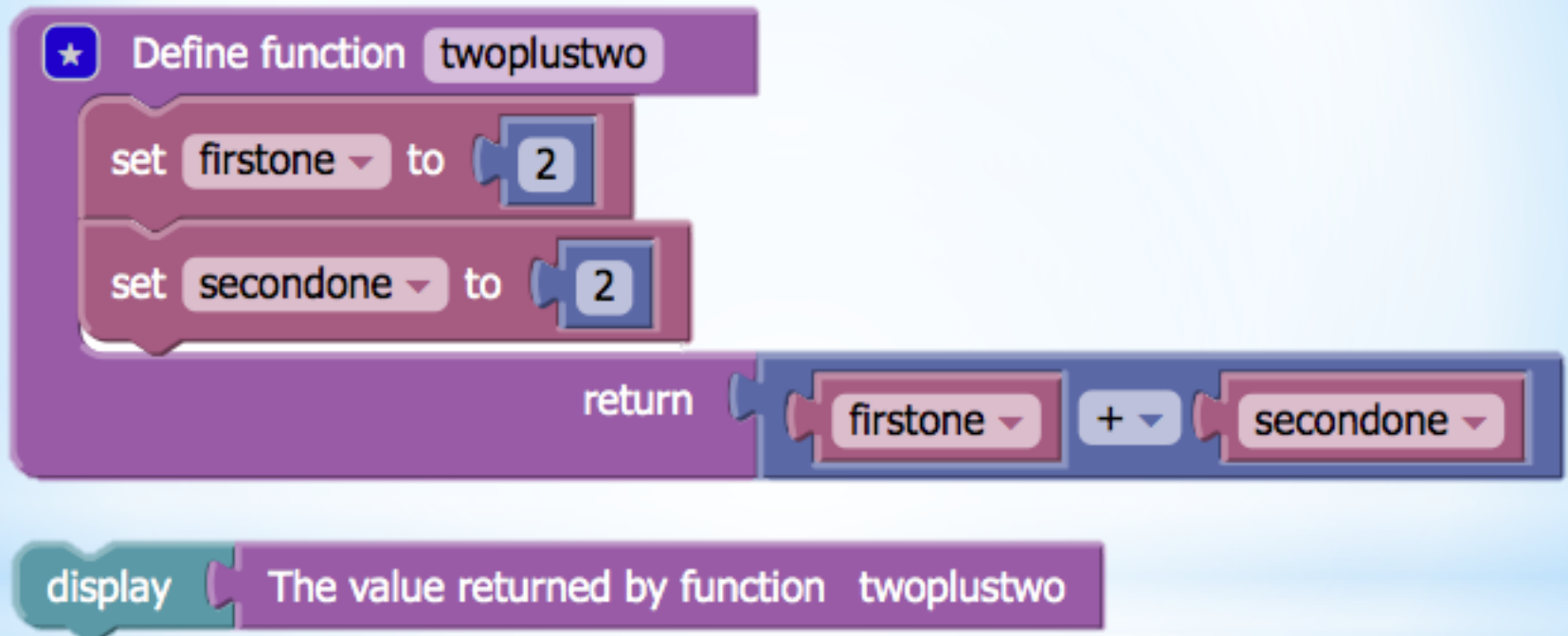
- Assuming this is a complete program, what would be printed to the screen? Why?



- NOTHING IS PRINTED!
 - The function is never called!
 - Also - do you see a **display** block anywhere in that code?!

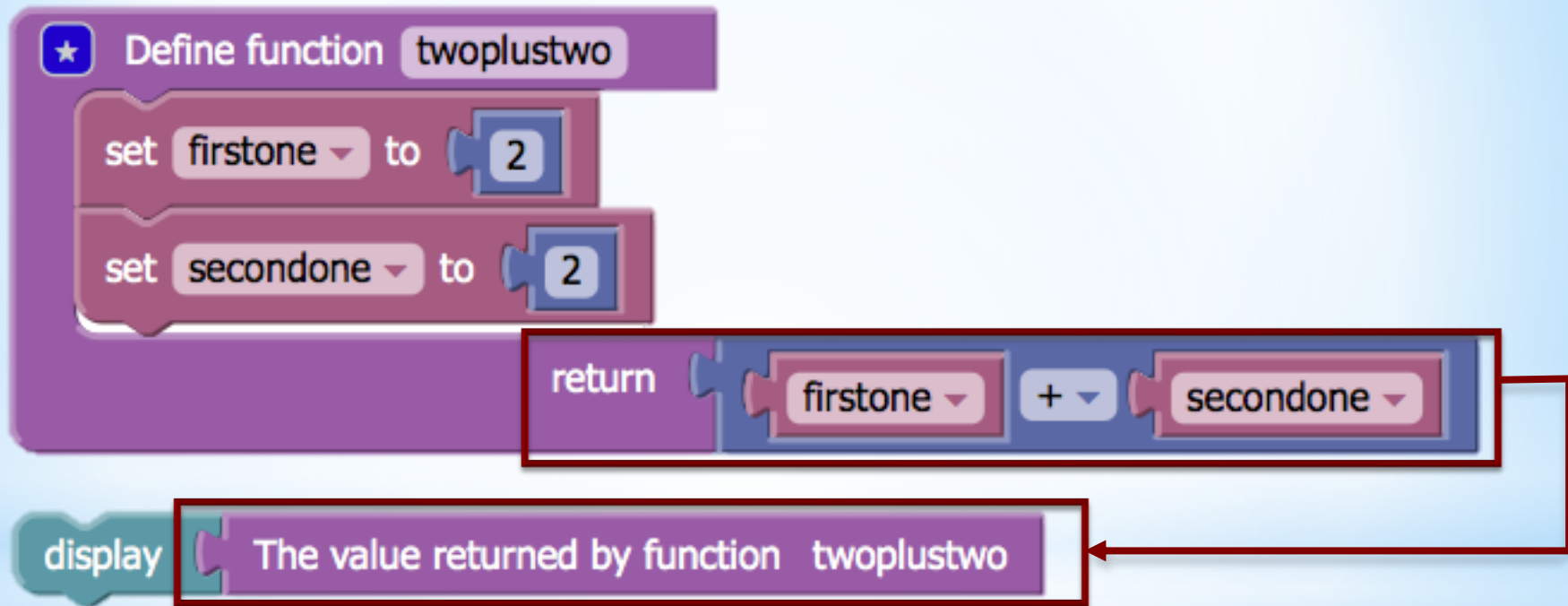
A simple example

- What would be printed to the screen? Why?



A simple example

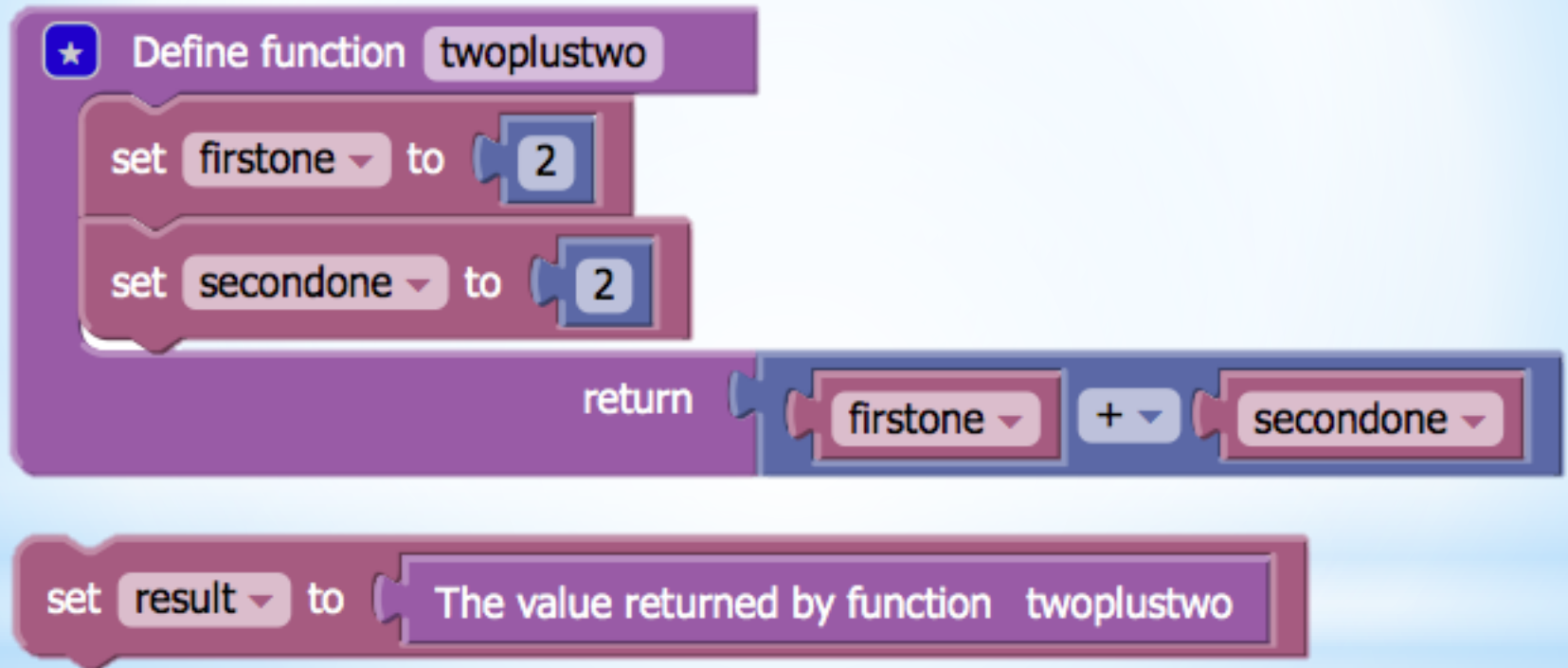
- What would be printed to the screen? Why?



- The value that gets *returned* by the function gets used by the display block in the main body
- Again: if the block fits...!

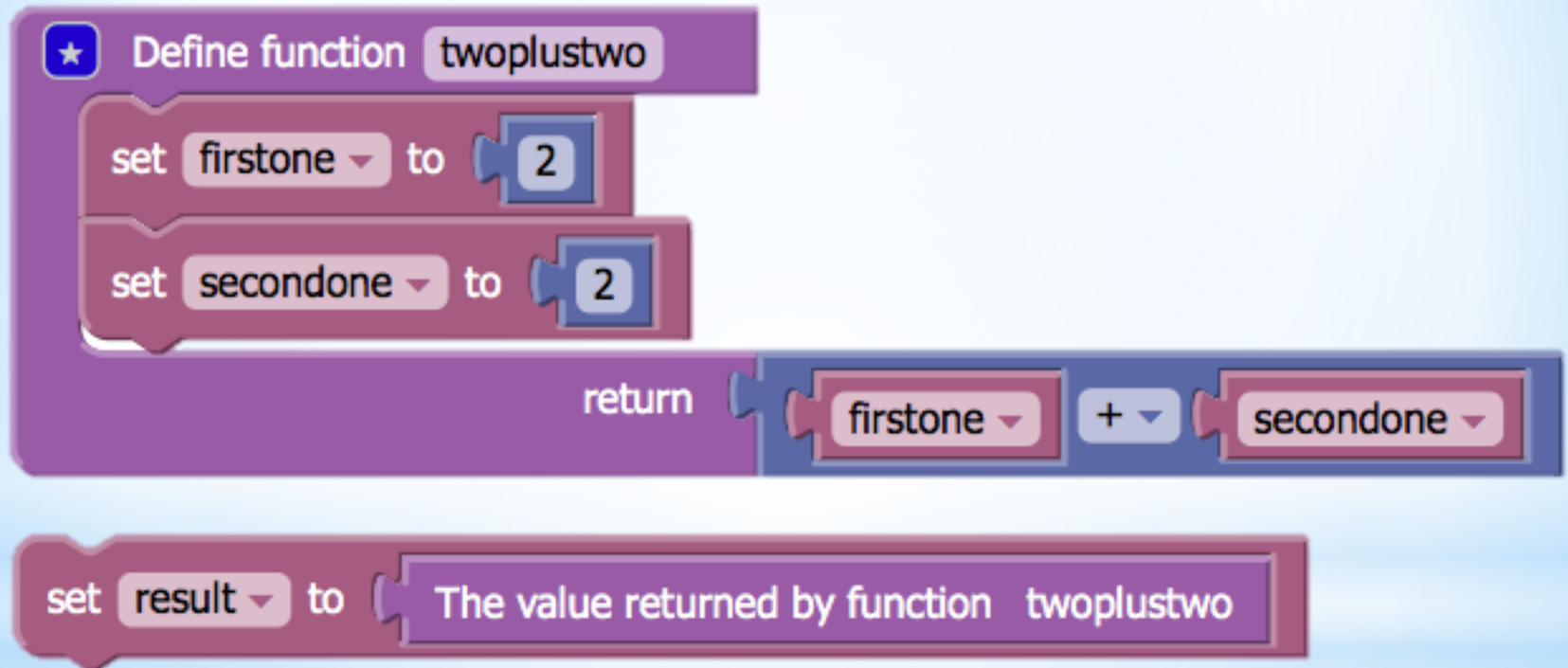
A simple example

- What would be printed to the screen? Why?
- What would be in the **result** variable?



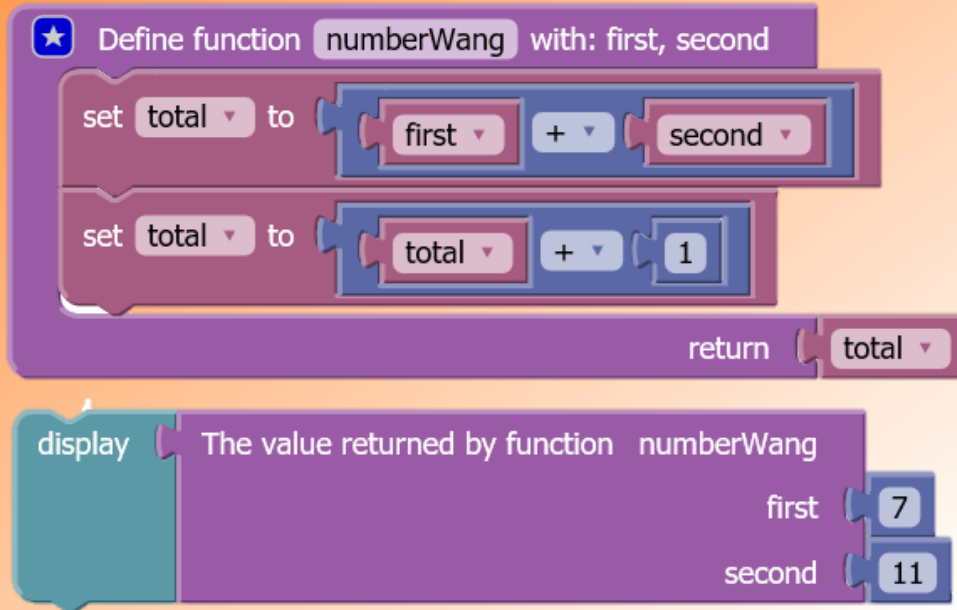
A simple example

- What would be printed to the screen? Why?
- What would be in the **result** variable?

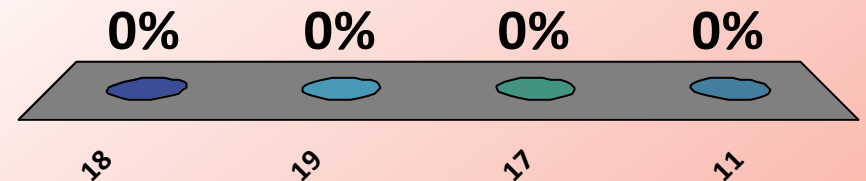


- NOTHING would be printed - again, do you see a **display** block anywhere?
- result** would contain 4

Using both return values and parameters; what will be printed?



- A. 18
- B. 19
- C. 17
- D. 11



Using both return values and parameters

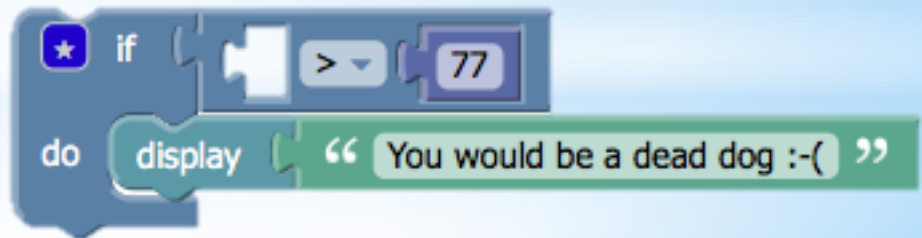
- ⦿ Say we wanted to examine the number of dog years the user was in our previous example...
- ⦿ ...say we wanted to check whether they'd still be alive or whether they'd have popped their clogs by now if they were a dog!

It's a dog's life...

- ◉ We might have some code like this...

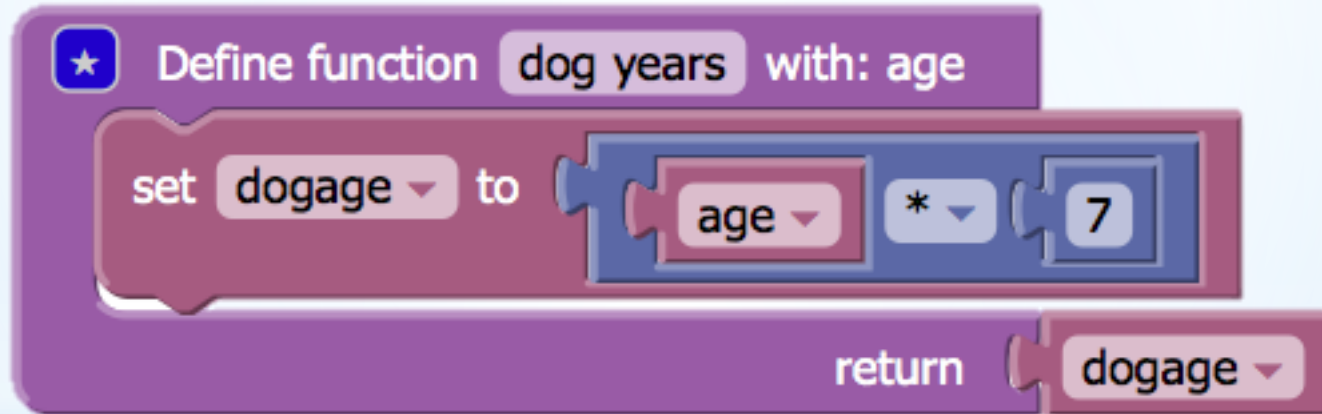


- ◉ But what would we put in the blank here? How would we get the result out of the function?



It's a dog's life

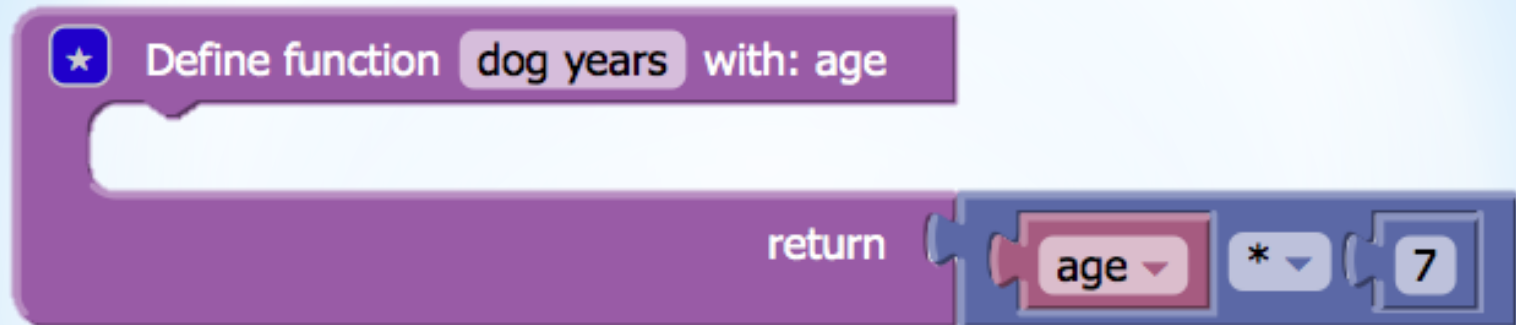
- ⦿ We would need to not only use a parameter in our function, but also a return value
- ⦿ So, our function might look like this



- ⦿ Note that our function doesn't print anything - all it does is calculate the result and send it back
- ⦿ The responsibility for doing something with that result lies with whatever code *calls* the function

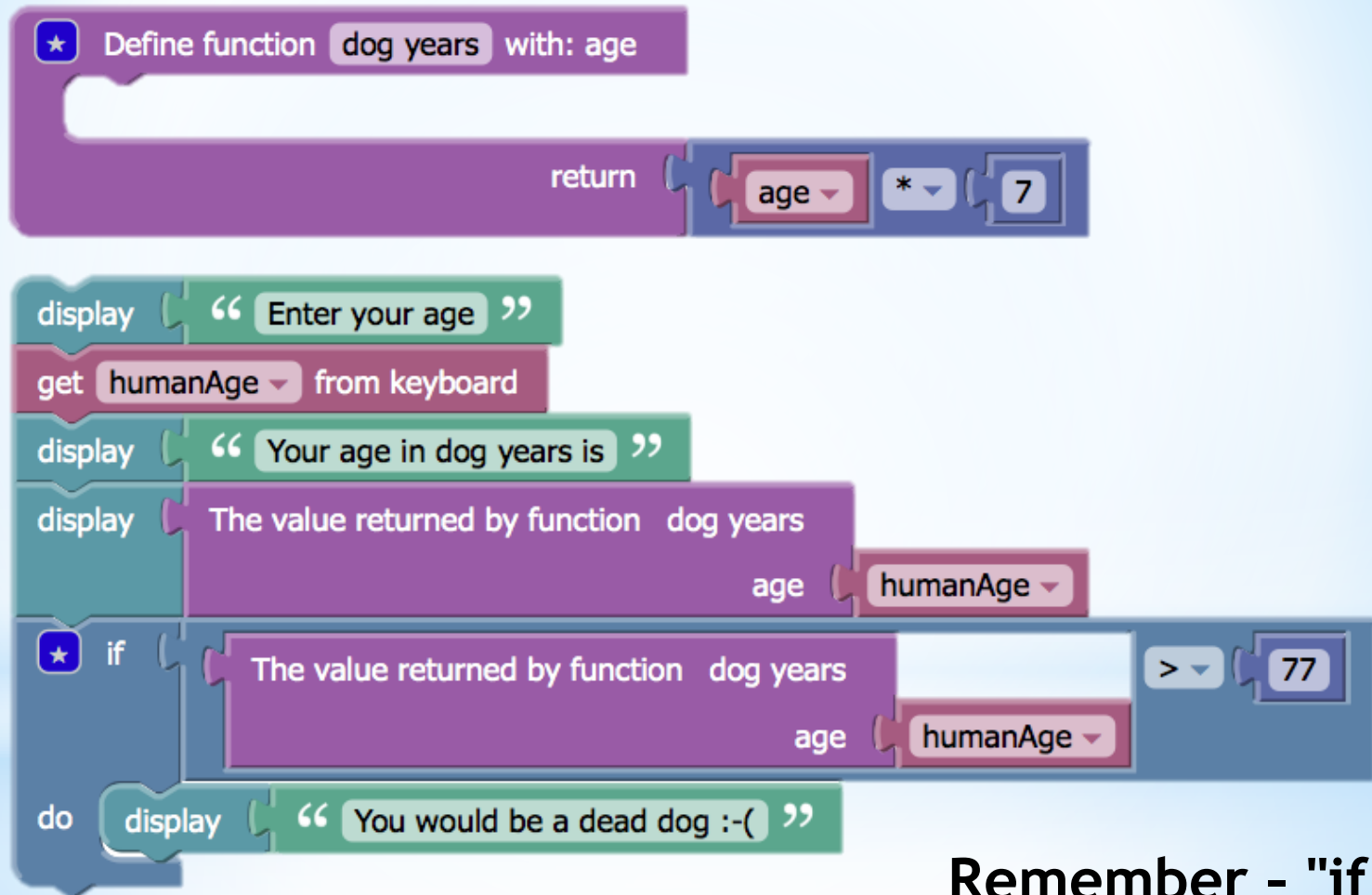
It's a dog's life

- ◉ In fact we could even write our function like this



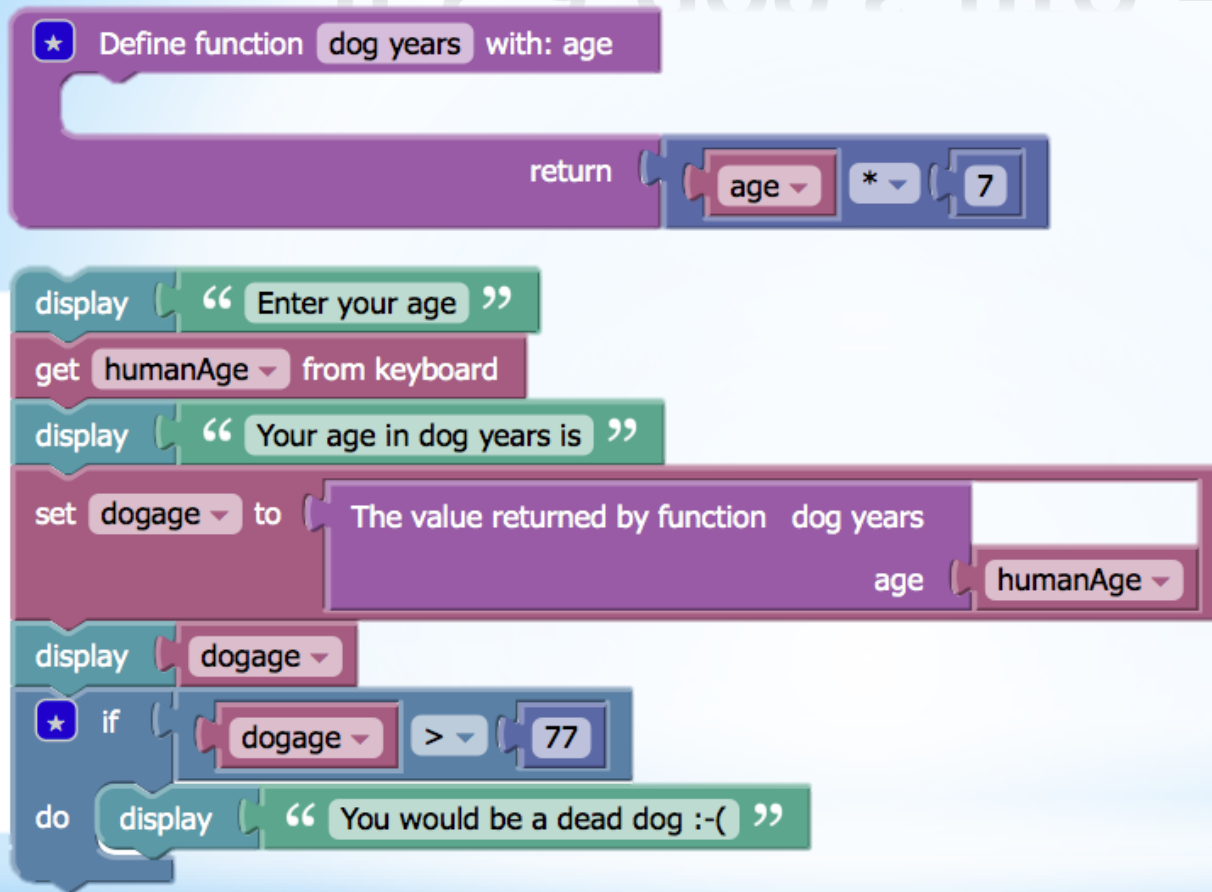
- ◉ Remember - "if the block fits..."
- ◉ All we want our function to do is send back the age in dog years - i.e. multiply it by 7
- ◉ The calculation block results in a single number, we want to return a single number - why bother creating another variable? Just clip the calculation block onto the return slot!

It's a dog's life - one solution



Remember - "if
the block fits"...

It's a dog's life - another solution



- ⦿ We can debate which is more "efficient"...
- ⦿ The solution on the previous page doesn't use an additional variable
- ⦿ The solution on this page is perhaps easier to read

It's a dog's life - the text based representation

```
function dog_years(age)
    return age * 7
endfunction

display "Enter your age"
get humanAge
display "Your age in dog years is"
set dogage = resultof dog_years(humanAge)
display dogage
if (dogage > 77)
    display "You would be a dead dog :-("
endif
```

It's a dog's life - the text based representation

```
function dog_years(age)
```

```
    return age * 7
```

```
endfunction
```

← Note the format of the `return` statement

```
display "Enter your age"
```

```
get humanAge
```

```
display "Your age in dog years is"
```

```
set dogage = resultof dog_years(humanAge)
```

```
display dogage
```

```
if (dogage > 77)
```


```
    display "You would be a dead dog :-("
```

```
endif
```

It's a dog's life - the text based representation

```
function dog_years(age)
    return age * 7;
endfunction
```

```
display "Enter your age"
get humanAge
display "Your age in dog years is"
set dogage = resultof dog_years(humanAge)
display dogage
if (dogage > 77)
    display "You would be a dead dog :-( "
endif
```



Note how we use the **resultof** statement here. This both calls the function and specifies that the result that is returned will be used as the value that (in this case) gets put into the variable

Summary

- ⦿ A boolean value is a value that can be either true or false
 - ⦿ (Our Carol functions like "Carol is blocked?" produced boolean values...)
- ⦿ Relational operators let us compare things
 - ⦿ e.g. are they equal? Is one greater than other?
- ⦿ A relational operator also gives back a boolean TRUE or FALSE
- ⦿ When we use the conditional programming blocks like IF, WHILE or REPEAT/UNTIL, the block expects a boolean value
- ⦿ Thus we can use relational operators together with conditional blocks to make decisions in our programs

Summary

- ⦿ Logical operators let us chain two boolean blocks together (such as the relational operator block)
- ⦿ The AND logical operator is TRUE if both sides of the expression are true
- ⦿ The OR logical operator is TRUE if one side or both sides of the expression are true
- ⦿ NOT reverses a boolean value
- ⦿ The logical operator block itself yields a boolean value - so you can nest them together to make "monster" blocks

Summary

- ⦿ Functions can take parameters and return a value
- ⦿ Parameters let you send data into the function
- ⦿ A return value lets you send data out of the function back to whatever called it
- ⦿ DON'T GET CONFUSED ABOUT RETURN VALUES AND JUST PRINTING SOMETHING TO THE SCREEN!
- ⦿ Remember - if the block fits, use it
 - ⦿ A call to a function with a return value might be used...
 - ⦿ ...clipped to a "set" variable block - the variable will be set to whatever the function returns
 - ⦿ ...clipped to a "display" block - whatever the function returns will be printed
 - ⦿ ...in fact, it could be used *anywhere* - the surrounding block will treat it like whatever value the function returns!