



DT Challenge Blockly

Chatbot

1. Getting started with code
2. Data types: numbers and strings
3. Strings: working with words
4. Project 1
5. Making decisions
6. Investigating strings
7. Project 2
8. Repeating things
9. Project 3
10. Project 4: Putting it all together



[\(https://creativecommons.org/licenses/by/4.0/\)](https://creativecommons.org/licenses/by/4.0/)

The Australian Digital Technologies Challenges is an initiative of, and funded by the
[Australian Government Department of Education and Training](https://www.education.gov.au/)
(<https://www.education.gov.au/>).

© Australian Government Department of Education and Training.

1

GETTING STARTED WITH CODE

1.1. Writing your first program

1.1.1. Hello, World!

Let's write your very first program in Blockly:

print "Hello, World! "

Hello, World!

You can run it by clicking the ► button (above).

When you run the program, you can see that it writes a message. That's what the **print** block does, it prints messages.

💡 Writing Blockly or Python?

Blockly uses *visual programming*, like [Scratch](https://scratch.mit.edu/) (<https://scratch.mit.edu/>). The blocks are code, just like any real-world language, such as [Python](https://python.org) (<https://python.org>).

You can see the Python version by clicking the  button.

If you want to learn Python, try the [Python](https://groklearning.com/course/aca-dt-78-py-chatbot/) (<https://groklearning.com/course/aca-dt-78-py-chatbot/>) version instead.

1.1.2. Is this just LEGO!?

What do the coloured blocks mean?

print

The **print** block is an *instruction* for the computer to follow. The hole in the block means it needs extra information to do its job.

You need to tell the computer what to print. Here is a message:

"Hello, World! "

This message block doesn't do anything by itself. It's just a message, not an *instruction*. So if you press play, nothing happens!

Put them together and you have an instruction that the computer can understand and run:



print "Hello, World!"



```
Hello, World!
```

LEGO® is a registered trademark of the LEGO Group.

1.1.3. Problem: Hello, World!



Time to solve your first problem! Write a program that prints out:

`Hello, World!`



What, again? **Yes, now it's your turn to write it from scratch.**

You need to put your blocks in the editor (the big area on the right).

The **print** block is in the **Output** menu. Drag it into the editor.

Add an empty **" "** string block from the **Strings** menu. Drag it into the hole in the **print** block, and change the message.

How do I submit?

1. Write your program in the editor (large panel on the right);
2. Run your program by clicking  in the top right-hand menu bar. The output will appear below your code. **Check the program works correctly!**
3. Mark your program by clicking  and we will automatically check if your program is correct, and if not, give you some hints to fix it up.

Testing

- ☐ Testing that the words are correct.
- ☐ Testing that the whitespace is correct.
- ☐ Testing that the punctuation is correct.
- ☐ Testing that the capitalisation is correct.
- ☐ Hurrah, you got *everything* right!

1.1.4. Problem: Therefore, I rock!



You've printed your first text - congratulate yourself! Write a program that prints out:

Therefore, I rock!

Don't forget, the **print** block is in the **Output** menu. Drag it into the editor.

Add an empty **" "** string block from the **Strings** menu. Drag it into the hole in the **print** block, and change the message. Remember that the marker is really picky about punctuation and spelling.

Testing

- ☐ Testing that the words are correct.
- ☐ Testing that the capitalisation is correct.
- ☐ Testing that the punctuation is correct.
- ☐ Testing that the whitespace is correct.
- ☐ Great work! You got *everything* right!

1.2. Strings and printing

1.2.1. A string of characters

Computers don't understand **Hello, World!**, or any other human language. To a computer, they are just a *string* of letters.

The green string block can contain any letters, numbers, punctuation and spaces that you want to use in a message:



```
print " abc ABC 123 @!?.# "
```

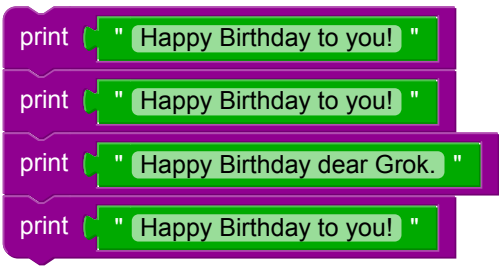


```
abc ABC 123 @!?.#
```

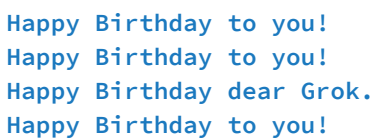
The individual letters, digits, symbols and spaces are called *characters* and the word string is short for *string of characters*.

1.2.2. Printing more messages

To print multiple messages, you just attach **print** blocks together:



```
print " Happy Birthday to you! "
print " Happy Birthday to you! "
print " Happy Birthday dear Grok. "
print " Happy Birthday to you! "
```



```
Happy Birthday to you!
Happy Birthday to you!
Happy Birthday dear Grok.
Happy Birthday to you!
```

Each message is printed on its own line. **Run it to check.**

1.2.3. Problem: Bashō Haiku



A [haiku](https://en.wikipedia.org/wiki/Haiku) (<https://en.wikipedia.org/wiki/Haiku>) is a three line Japanese poem. The first and last lines have five syllables, and the middle line has seven. Here's a famous example by [Bashō Matsuo](https://en.wikipedia.org/wiki/Matsuo_Bash%C5%8D) (https://en.wikipedia.org/wiki/Matsuo_Bash%C5%8D):

古池や
蛙飛びこむ
水の音

Write a program which prints an English translation of this haiku:

An old silent pond...
A frog jumps into the pond,
splash! Silence again.

You will need to use three **print** blocks – one for each line.

Hint:

Pay careful attention to capital letters and punctuation! You should print out *exactly* the same translation.

Testing

- ☐ Testing the words in the first line.
- ☐ Testing the capital letters in the first line.
- ☐ Testing the punctuation and spaces in the first line.
- ☐ Testing the words and spaces in the second line.
- ☐ Testing the capitals and punctuation in the second line.
- ☐ Testing the words and spaces in the third line.
- ☐ Testing the capitals and punctuation in the third line.

1.2.4. Problem: Ancient Riddle



A [riddle](https://en.wikipedia.org/wiki/Riddle) (<https://en.wikipedia.org/wiki/Riddle>) is a question with an answer that is a mystery or double meaning. They were very popular in ancient times but also in the writings of [J. R. R. Tolkien](https://en.wikipedia.org/wiki/J._R._R._Tolkien) ([https://en.wikipedia.org/wiki/J. R. R. Tolkien](https://en.wikipedia.org/wiki/J._R._R._Tolkien)) and [J. K. Rowling](https://en.wikipedia.org/wiki/J._K._Rowling) ([https://en.wikipedia.org/wiki/J. K. Rowling](https://en.wikipedia.org/wiki/J._K._Rowling)):

Write a program which prints an Ancient Riddle:

```
There is a house.
One enters it blind and comes out seeing.
What is it?
A school
```

You will need to use four **print** blocks – one for each line.

Hint:

Pay careful attention to capital letters and punctuation! You should print out *exactly* the same riddle.

Testing

- ☐ Testing the words in the first line.
- ☐ Testing the capital letters in the first line.
- ☐ Testing the punctuation in the first line.
- ☐ Testing that the spaces in the first line.
- ☐ Testing the words and spaces in the second line.
- ☐ Testing the capitals and punctuation in the second line.
- ☐ Testing the words and spaces in the third line.
- ☐ Testing the capitals and punctuation in the third line.
- ☐ Testing the words and spaces in the fourth line.
- ☐ Testing the capitals and punctuation in the fourth line.

1.3. Variables

1.3.1. Remember messages with variables

Writing a long message many times is a pain. It would be great if we could store the message somewhere and reuse it.

A *variable* is that place! Each variable has a *name* which we use to set and get the message:



```

For she's a jolly good fellow.
For she's a jolly good fellow.
For she's a jolly good fellow.
And so say all of us!
  
```

The `set line` block creates a new variable called `line`. It holds the message `"For she's a jolly good fellow. "`. We can then use the `line` block to print that message as often as we want.

💡 Creating a variable

To create a new variable, click the down arrow next to the variable name and select **New Variable...**

1.3.2. More `print` blocks

What if you want to print more than one message on a line?

Blockly has `print` blocks with more than one hole, e.g.:



You can use these with multiple strings, like this:



```

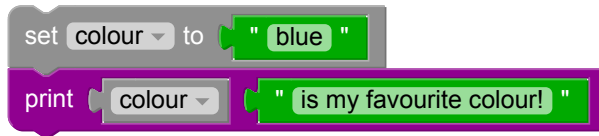
Harry Potter
  
```

When you run it, notice that the output is not `HarryPotter`.

The `print` automatically adds a space between the two strings.

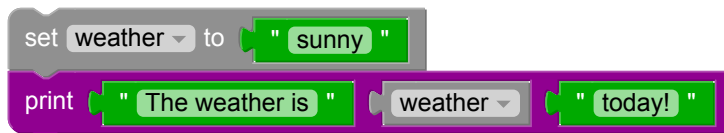
1.3.3. Mixing messages and variables

We can now use **print** to print a mix of strings and variables:



blue is my favourite colour!

This example uses a 3-hole block to print the variable in the middle:



The weather is sunny today!


The **print** block puts a space between *each* value.

1.3.4. Problem: Library Day



You need to remember to bring your library bag and book returns on Library day.

We've put a program in the editor that reminds you what day is Library day.

Click  to see what it does.

Run

The day you need to bring your book bag and returns is stored in the `library day` variable.

Disaster! Library day has changed from Friday to Wednesday so your reminder doesn't work.

Update this program so that it works for a different day, `Wednesday`. Your updated program should print the message:

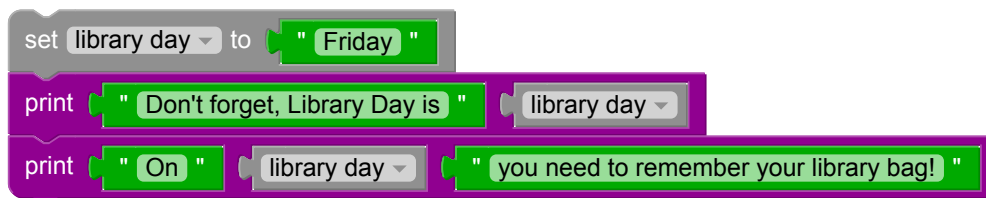
Don't forget, Library Day is Wednesday
On Wednesday you need to remember your library bag!

 **Only change the** `library day`

You just need to change the string `Friday` to `Wednesday`, run it to check it works, and then mark it.

You'll need

 [program.blockly](https://aca.edu.au/challenges/56-blockly.html)



Testing


- ☐ Testing that the words are correct.
- ☐ Testing that the whitespace is correct.
- ☐ Testing that the punctuation is correct.
- ☐ Testing that the capitalisation is correct.
- ☐ Great work, we love borrowing books!

1.3.5. Problem: I have no homework



Sometimes, if you repeat something often enough, it seems true. Repeating an opinion over and over again to make it seem true is called [argumentum ad nauseam](http://rationalwiki.org/wiki/Argumentum_ad nauseam) (http://rationalwiki.org/wiki/Argumentum_ad nauseam), or *argument by repetition*.

We've written a program in the editor that repeats something a statement.

Click  to see what it does.

The statement is stored in a variable called `suggestion`.

Update this program so that it works for a different suggestion: 'I have no homework.' Your updated program should print the message:

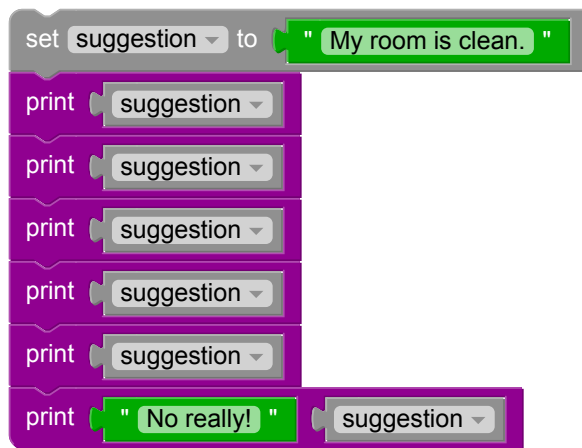
```
I have no homework.
I have no homework.
I have no homework.
I have no homework.
I have no homework.
No really! I have no homework.
```

 **Only change the string stored in the `suggestion` variable!**

You just need to change the value of `suggestion` to be `I have no homework.` instead of `My room is clean.`, run it to check it works, and then mark it.

You'll need

 [program.blockly](#)



Testing

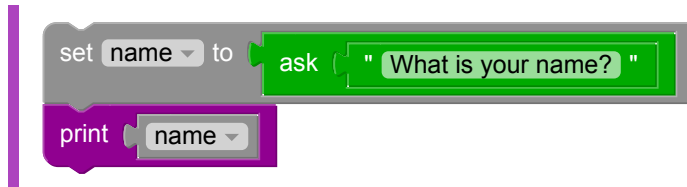
- ☐ Testing that the words are correct.
- ☐ Testing that the whitespace is correct.
- ☐ Testing that the punctuation is correct.
- ☐ Testing that the capitalisation is correct.

☐ Great work! Keep working on your Jedi mind tricks!

1.4. Reading user input

1.4.1. Asking the user for information

Let's write a program that asks the user for their name:



Run this program. Even if you haven't run any so far, run this one!

You will need to type a name and press Enter:

```

What is your name? Grok
Grok
  
```

The **ask** block needs a question string to *ask the user*. It returns the user's answer to our program as a new string. Our program stores the answer in the **name** variable so we can print it later.

Run it again with a different name. Then try changing the question.



1.4.2. Problem: Echo! Echo!

There is a game that little kids play when they're first learning to speak. It's called Echo!

It's a pretty easy game. You just need to repeat whatever was just said.

You are going to program this game. You need to get input from the user and print back exactly what the user input.

You are going to need a **variable** to store the words, an **ask** block to get the words from the user and a **print** block to make it appear on the screen.

What do you want to say? Echo!
Echo!

Your program should work with anything the user types:

What do you want to say? I am having a great day!
I am having a great day!

Hint

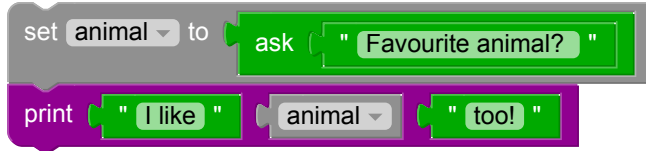
Make sure you give **ask** the same prompt question that is used in the example above.

Testing

- ☐ Testing that the words in the prompt are correct.
- ☐ Testing that the punctuation in the prompt is correct.
- ☐ Testing that the capitalisation in the prompt is correct.
- ☐ Testing that the white space in the prompt is correct.
- ☐ Testing with the word **Echo!**
- ☐ Testing with the word **I am having a great day!**
- ☐ Testing a hidden test case (to make sure your program works in general).

1.4.3. Variable variables!

Now we see why variables are called *variables*! When you run the program and ask the user a question, they could type anything:



```

Favourite animal? tigers
I like tigers too!
  
```

Here, the `animal` variable contains the message `" tigers "`, but if the user types in something else, it will contain something else:

```

Favourite animal? pineapple
I like pineapple too!
  
```

This time, the `animal` variable contains `" pineapple "`.

Variables are *variable* because you may not know their value when you write the program, it could be *anything*!

1.4.4. Problem: Meet the Puppy



Your friend has a new puppy! Write a program to tell the puppy to sit using its name, like this:

```
What is the puppy called? Fluffy
Sit Fluffy
```

Your program should work with any puppy name:

```
What is the puppy called? Shadow
Sit Shadow
```

When your program runs, it should wait for the user to type in the puppy's name, using the **ask** block. It should then use the name the user entered to tell the puppy to sit.

Hint

Make sure you give **ask** the same prompt question that is used in the example above.

Testing

- ☐ Testing that the words in the prompt are correct.
- ☐ Testing that the punctuation in the prompt is correct.
- ☐ Testing that the capitalisation of the prompt is correct.
- ☐ Testing that the whitespace in the prompt is correct.
- ☐ Testing that the words in the sit command are correct.
- ☐ Testing that the punctuation in the sit command is correct.
- ☐ Testing that the capitalisation in the sit command is correct.
- ☐ Testing that the whitespace in the sit command is correct.
- ☐ **Yay, you've got the first example right!**
- ☐ Testing the second example in the question (when the user enters **Shadow**).
- ☐ Testing with the name **Fido**.
- ☐ Testing a two word name (**Shaggy Dog**).
- ☐ Testing a hidden test case (to make sure your program works in general).
- ☐ Testing another hidden test case.

1.4.5. Problem: Cheer me on!



Several of your friends are competing in a bike race and you want to cheer them all on! Write a program to write a cheer for you to shout for any friend's name.

Friend: Jane
Go Jane Go!

Your program should work with any friend's name:

Friend: Isabella
Go Isabella Go!

When your program runs, it should wait for the user to type in their friend's name, using the **ask** block. It should then use the name the user entered to tell make the cheer.

Hint

Make sure you give **ask** the same prompt question that is used in the example above.

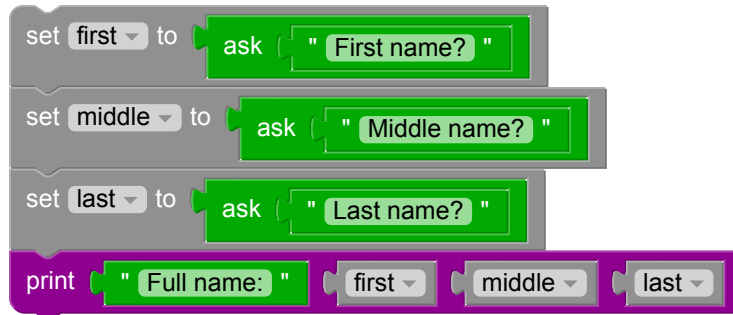
Testing

- ☐ Testing that the words in the prompt are correct.
- ☐ Testing that the punctuation in the prompt is correct.
- ☐ Testing that the capitalisation of the prompt is correct.
- ☐ Testing that the whitespace in the prompt is correct.
- ☐ Testing that the words in the cheer are correct.
- ☐ Testing that the punctuation in the cheer is correct.
- ☐ Testing that the capitalisation in the cheer is correct.
- ☐ Testing that the whitespace in the cheer is correct.
- ☐ **Yay, you've got the first example right!**
- ☐ Testing the second example in the question (when the user enters **Isabella**).
- ☐ Testing with the name **Vivek**.
- ☐ Testing a two word name (**Juan Vicente**).
- ☐ Testing a hidden test case (to make sure your program works in general).
- ☐ Testing another hidden test case.

1.5. Multiple variables

1.5.1. Using multiple variables

You can create as many variables as you need, as long as they have different names (otherwise, you're setting an existing variable).



The variable names make it very clear our program makes a full name out of a first, middle and last name:

```

First name? Hans
Middle name? Christian
Last name? Andersen
Full name: Hans Christian Andersen
  
```

Good variable names help explain what the code does!

1.5.2. Problem: Match of the Year!



It's the greatest match ever! Team 1 against Team 2... oh what's that? The team names are missing.

Write a program that asks for two team names, then prints out an announcement of the match.

Here is an example:

```
Who is team 1? Diamonds
Who is team 2? Silver Ferns
The match of the year: Diamonds vs. Silver Ferns
```

Here's another example:

```
Who is team 1? Liverpool
Who is team 2? Everton
The match of the year: Liverpool vs. Everton
```

Testing

- ☐ Testing the words in the first prompt message.
- ☐ Testing the words in the second prompt message.
- ☐ Testing the capitalisation of the prompts.
- ☐ Testing the punctuation in the prompts.
- ☐ Testing the spaces in the prompts.
- ☐ Testing the words in the match line.
- ☐ Testing the punctuation, spaces and capitals in the match line.
- ☐ **Testing the first example in the question.**
- ☐ Testing the second example in the question.
- ☐ Testing two rugby league teams.
- ☐ Testing two NBA basketball teams.
- ☐ Testing a hidden case.

1.5.3. Problem: Best New Ice Cream Combination



Everyone knows that two scoops of ice cream are better than one! But the best part is coming up with a new winning combination of flavours.

Write a program that asks for two ice cream flavours, then prints out an announcement of the new best ever flavour combination.

Here is an example:

```
Flavour 1: Chocolate
Flavour 2: Hazelnut
New best ever flavour combination...
Chocolate and Hazelnut
```

Here's another example:

```
Flavour 1: Sour Cherry
Flavour 2: Cookies 'n' Cream
New best ever flavour combination...
Sour Cherry and Cookies 'n' Cream
```

Testing

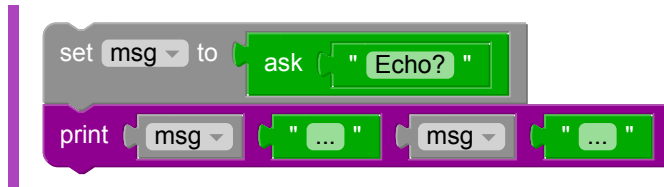
- ☐ Testing the words in the first prompt message.
- ☐ Testing the words in the second prompt message.
- ☐ Testing the capitalisation of the prompts.
- ☐ Testing the punctuation in the prompts.
- ☐ Testing the spaces in the prompts.
- ☐ Testing the words in the best ever flavour line.
- ☐ Testing the words in the ice cream combo line.
- ☐ Testing the punctuation, spaces and capitals in the ice cream combo line.
- ☐ **Testing the whole first example.**
- ☐ Testing the second example in the question.
- ☐ Testing with Mango and Chocolate.
- ☐ Testing with Green Tea and Black Sesame.
- ☐ Testing a hidden case.

1.6. Congratulations

1.6.1. Congratulations!

Congratulations, you have completed the first module!

You should now be able to write and explain code like this:



1. the **ask** block asks for a string with prompt **Echo?**;
2. the string is stored in **msg** (short for *message*);
3. the **print** block prints message and ... twice on one line.

2

DATA TYPES: NUMBERS AND STRINGS

2.1. Numbers

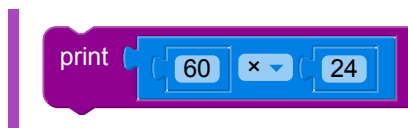
2.1.1. Blockly, the calculator

Blockly can do maths too!

Let's calculate how many minutes there are in a day:

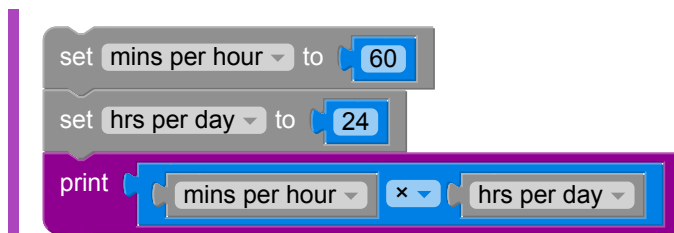
$$60 \text{ minutes per hour} \times 24 \text{ hours per day}$$

In Blockly, this calculation looks like:



1440

The same calculation can be done with variables:



1440

This code is longer, but a lot easier to understand.

2.1.2. Doing maths

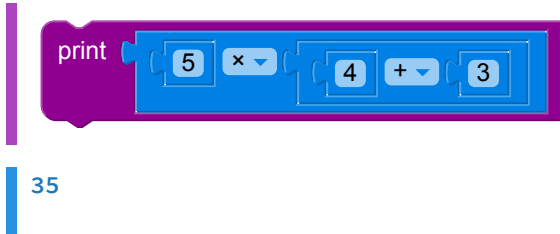
Blockly can do all the maths a calculator can do:

Sign	Name
+	add
-	subtract
x	multiply

Sign	Name
÷	divide

Click on the sign in the maths block to change what it does.

Blockly calculates the inner maths blocks first, so:



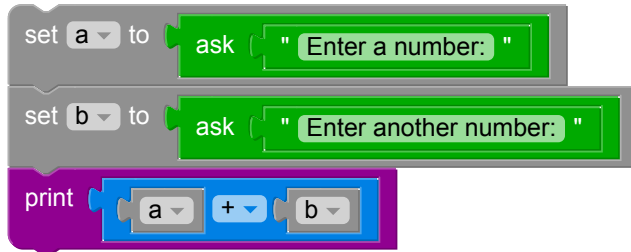
calculates the $4 + 3$ first (to give 7) and then does 5×7 .

In maths, we use brackets like this: $5 \times (4 + 3)$ to say *do the add before the multiply*.

2.2. Data types

2.2.1. Trying to add numbers from the user

Let's ask the user for two numbers and then add them together:



Run this, you'll be surprised by the result! Try entering 5 and 6:

```

Enter a number: 5
Enter another number: 6
56
  
```

That's unexpected! The answer should be 11, but we got 56.

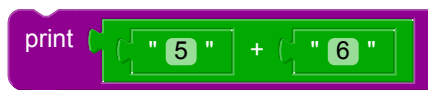
Strings and numbers are different types of data.

2.2.2. Different types of data

The computer uses "5" and 5 in different ways, even though they look the same to us, because **they are different data types**.

Blockly uses colour to show the data type: the green blocks are strings and the blue blocks are numbers.

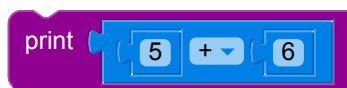
If you add two strings, the computer doesn't try to understand what's in them. It just joins the strings together:



```

56
  
```

If you add two numbers, the computer knows how to sum them:



```

11
  
```

Strings and numbers are two ways of **representing** the value 5.

💡 **Operations have types too!**

Adding strings creates a new string, and so the block that adds strings is **green**. It appears in the **Strings** menu in the editor.

Adding numbers creates a new number, so its block is **blue**.

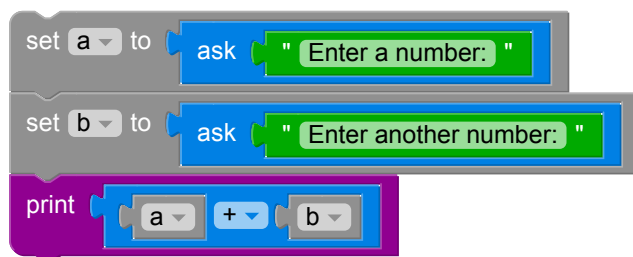
2.2.3. Reading in numbers

The green **ask** block always returns a string. If you want to read in a number, use the blue **ask** block instead:



This block tries to interpret what the user types as a number.

Let's have another go at adding two numbers:



This gives us the answer we expect:

```
Enter a number: 5
Enter another number: 6
11
```

2.2.4. Problem: Next Olympics



The summer [Olympic Games](https://en.wikipedia.org/wiki/Olympic_Games) (https://en.wikipedia.org/wiki/Olympic_Games) happen every 4 years.

Write a program which asks for the year of the previous summer Olympics, then prints the year of the next Olympics (by adding 4).

Here's how it should work:

```
When is the Olympics? 2016
The next Olympics is in 2020
```

Here's another example:

```
When is the Olympics? 2020
The next Olympics is in 2024
```

Your program should still add 4, even if the year entered is wrong:

```
When is the Olympics? 2011
The next Olympics is in 2015
```

 **Use the `ask` block!**

Make sure you read the year from the user with the blue `ask` block, so that it is a number, not a string.

Testing

- ☐ Testing the words in the first example in the question.
- ☐ Testing the punctuation, spaces and capitals in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing an Olympic year.
- ☐ Testing another Olympic year.
- ☐ Testing a hidden case.
- ☐ **Great work. Bring on Rio!**

2.2.5. Problem: Letter from the Queen



For nearly 100 years, there has been a tradition that citizens of the United Kingdom and Commonwealth countries receive a letter from the King or Queen on their [100th birthday](https://en.wikipedia.org/wiki/Centenarian#British_and_Commonwealth_traditions). ([https://en.wikipedia.org/wiki/Centenarian#British and Commonwealth traditions](https://en.wikipedia.org/wiki/Centenarian#British_and_Commonwealth_traditions)).

Write a program that works out how long until your letter arrives!

Your program should ask the user how old they are then calculate 100 minus their age. For example:

```
How old are you? 14
You must wait 86 years.
```

Here is another example:

```
How old are you? 52
You must wait 48 years.
```

If you're already over 100, the answer is a negative number:

```
How old are you? 102
You must wait -2 years.
```

Testing

- ☐ Testing the first example from the question.
- ☐ Testing the second example from the question.
- ☐ Testing the third example from the question.
- ☐ Testing a very young person (3 years old).
- ☐ Testing someone who is almost there! (99 years old).
- ☐ Testing someone who is halfway there (50 years old).
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

2.3. Multiplying strings

2.3.1. Joining strings together

As well as printing multiple strings on one line, we can also join or add them together to create larger strings:

To print the word **Harry** followed by the word **Potter**, we can use:



HarryPotter

This doesn't add any extra spaces, so you need to add them yourself if you want some:

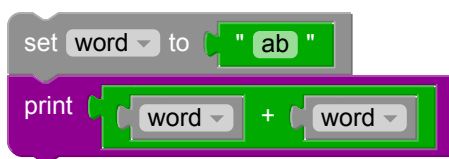


Harry Potter

Programmers call adding strings together *string concatenation*, which is often abbreviated to *concat* or just *cat*.

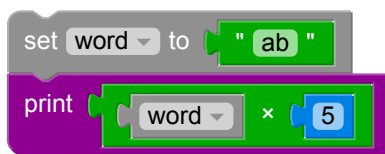
2.3.2. Multiplying strings

We can already add two strings (*concatenation*):



abab

Since multiplication is like repeated addition, it turns out we can also multiply a string by a number, say 10. It is like adding the string to itself until you have 10 copies of it:



ababababab

Note that subtraction and division don't work with strings (and integers) because it isn't obvious what they would do.

2.3.3. Problem: Noooooooooo!



When writing a movie script, sometimes you want a simple “No” and sometimes a longer dramatic “Noooooooooo”.

Write a program which asks how long the No should be, then uses that number of o's. Here is an example:

```
How long? 1
No
```

Here is another example with more o's:

```
How long? 10
Nooooooooooo
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing a length of 5.
- ☐ Testing a length of 4.
- ☐ Testing a length of 15.
- ☐ Testing a length of 0.
- ☐ Testing a hidden case.

2.3.4. Problem: Pull a happy face



Now that you're getting the hang of programming, you want to show everyone how happy you are! Write a program to help you generate [Japanese-style emoticons](https://en.wikipedia.org/wiki/Japanese_style_emoticons) (https://en.wikipedia.org/wiki/Japanese_style_emoticons) to express your joy. ^_^

Write a program which asks how happy you are, and prints out a face to match!

How happy? 1

^_^

Here is another example, where you're rather happy:

How happy? 3

^___^

Here is another example, where you're really happy:

How happy? 10

^-----^

Testing

- ☐ Testing the words in the input prompt.
- ☐ Testing the punctuation in the input prompt.
- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a case when you are at happiness level 2.
- ☐ Testing a case when you are at happiness level 6.
- ☐ Testing a case when you are at happiness level 0.
- ☐ Testing a case when you are at happiness level 11.
- ☐ Testing a hidden case.

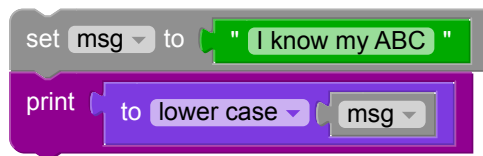
3

STRINGS: WORKING WITH WORDS

3.1. Uppercase and lowercase

3.1.1. Converting to lower and UPPER case

We often want to change a string to uppercase (capital letters) or lowercase. Blockly has a block for this (under **Strings**):



i know my abc

The **lowercase** block converts any capitals in **msg** to lowercase. The other characters are unchanged.
Try it yourself!

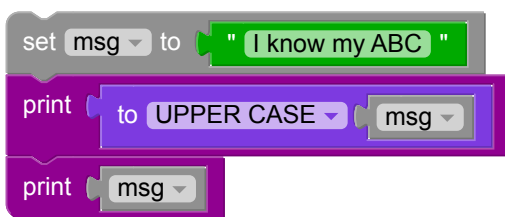
💡 lowercase, UPPERCASE and Titlecase

This block allows you to convert to **lowercase**, **UPPERCASE** or **Titlecase**. Click **lowercase** to choose from a list.

Titlecase is what we use for most names in English — uppercase first letter in each word, and lowercase for every other letter.

3.1.2. Saving the modified string

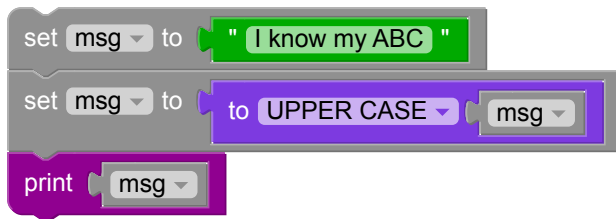
Changing the case of a string never modifies the original string:



I KNOW MY ABC
I know my ABC

Instead, it creates a new converted string.

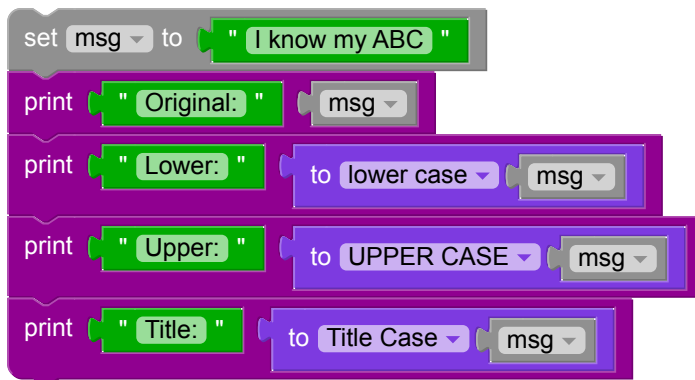
To modify the original, you need to store it back in the old variable:



I KNOW MY ABC

3.1.3. All the cases!

Let's try all the capitalisation options:



Original: I know my ABC
Lower: i know my abc
Upper: I KNOW MY ABC
Title: I Know My Abc

3.1.4. Problem: Shout it from the rooftops



Have you ever felt something so strongly you want to shout it from a rooftop?

Let's write a SHOUTER program that asks the user for some text and then SHOUTS it in upper case for us.

Here is an example interaction with the program:

```
Enter text: I love programming!
I LOVE PROGRAMMING!
```

Here is another example:

```
Enter text: It's lunchtime!
IT'S LUNCHTIME!
```

Testing

- ☐ Testing that the words in the prompt are correct.
- ☐ Testing that the punctuation in the prompt is correct.
- ☐ Testing that the capitalisation of the prompt is correct.
- ☐ Testing that the whitespace in the prompt is correct.
- ☐ Testing the words in the first example in the question.
- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing a short message.
- ☐ Testing a long message.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

3.1.5. Problem: Breaking the fourth wall



An [aside](https://en.wikipedia.org/wiki/Aside) (<https://en.wikipedia.org/wiki/Aside>) is a dramatic device used to let a character speak directly to the audience, without the other characters hearing. It's a way of breaking the [fourth wall](https://en.wikipedia.org/wiki/Fourth_wall) ([https://en.wikipedia.org/wiki/Fourth wall](https://en.wikipedia.org/wiki/Fourth_wall)). It is used in theatre, TV and film, and needs to be specifically marked as an *aside* in a script.

You're writing a play, and want to make sure your asides stand out. Write a program to help. Each aside should start with **Aside:** and be entirely in **lower case**.

Here is an example from [Hamlet](https://en.wikipedia.org/wiki/Hamlet) (<https://en.wikipedia.org/wiki/Hamlet>), by Shakespeare:

Line: A little more than kin, and less than kind.
Aside: a little more than kin, and less than kind.

Here is another example from [The Emperor's New Groove](https://en.wikipedia.org/wiki/The_Emperor%27s_New_Groove) ([https://en.wikipedia.org/wiki/The Emperor%27s New Groove](https://en.wikipedia.org/wiki/The_Emperor%27s_New_Groove)):

Line: This is his story. WELL, ACTUALLY my story.
Aside: this is his story. well, actually my story.

Here's one more example from [Ferris Bueller's Day Off](https://en.wikipedia.org/wiki/Ferris_Bueller%27s_Day_Off) ([https://en.wikipedia.org/wiki/Ferris Bueller%27s Day Off](https://en.wikipedia.org/wiki/Ferris_Bueller%27s_Day_Off)):

Line: How could I possibly be expected to handle school on a day like this?
Aside: how could i possibly be expected to handle school on a day like this?

Testing

- ☐ Testing the words in the first example.
- ☐ Testing the spaces in the first example.
- ☐ Testing the punctuation in the first example.
- ☐ Testing the capitalisation in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a short aside.
- ☐ Testing a long aside.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

3.2. String length

3.2.1. How long is a string?

The **length** block counts the characters in a string:



12

It **counts all the characters**, including the 5 letters in **Hello**, the space, the 5 letters in **World**, and the exclamation mark.

We can use it to count the number of letters in the **alphabet** :



26

3.2.2. Problem: Twittier: Can I Tweet that?



[Twitter \(https://en.wikipedia.org/wiki/Twitter\)](https://en.wikipedia.org/wiki/Twitter) is a social network where users post short "tweets" that are 140 characters or fewer. You've made your own version, "Twittier", that limits posts to only 44 characters!

Write a program to read in the post you'd like to make, and tells you how many characters you have spare.

Message: First post!
You have 33 character(s) left.

Here is another example:

Message: Can't wait until the weekend! #sleepin
You have 6 character(s) left.

Here's an example with a message of exactly 44 characters:

Message: This msg is exactly 44 characters - no more!
You have 0 character(s) left.

If the post is longer than 44 characters, it should print out the negative number of characters:

Message: Fitbits are like Tamagochis, except you're trying to keep yourself alive.
You have -29 character(s) left.

Testing

- ☐ Testing the words in the first example.
- ☐ Testing the spaces in the first example.
- ☐ Testing the punctuation in the first example.
- ☐ Testing the capitalisation in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the fourth example in the question.
- ☐ Testing another long post.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

3.2.3. Problem: How good is your vocabulary?



If you're a fan of hangman you need to know some long words. [English has a lot of very long words \(https://en.wikipedia.org/wiki/Longest_word_in_English\)](https://en.wikipedia.org/wiki/Longest_word_in_English). The longest non-technical word in the English language is [antidisestablishmentarianism \(https://en.wikipedia.org/wiki/Antidisestablishmentarianism_\(word\)\)](https://en.wikipedia.org/wiki/Antidisestablishmentarianism_(word)). At 28 letters it is a **very** long word.

You want to test your friends' vocabulary by seeing how close they can get to the longest word.

Write a program to read in the longest word your friends can think of and tell them how close they are to the length of "antidisestablishmentarianism":

Long word: incomprehensibilities
It is 7 letter(s) shorter than the longest word.

Here is another example:

Long word: unimaginatively
It is 13 letter(s) shorter than the longest word.

If the long word is longer than 28 characters, it should print out the negative number of letters

Long word: supercalifragilisticexpialidocious
It is -6 letter(s) shorter than the longest word.

But we all know that's not a *real* word.

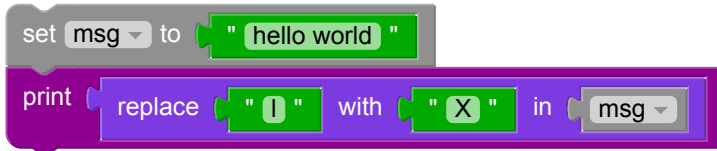
Testing

- ☐ Testing the words in the first example.
- ☐ Testing the spaces in the first example.
- ☐ Testing the punctuation in the first example.
- ☐ Testing the capitalisation in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing another long post.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

3.3. Parts of strings

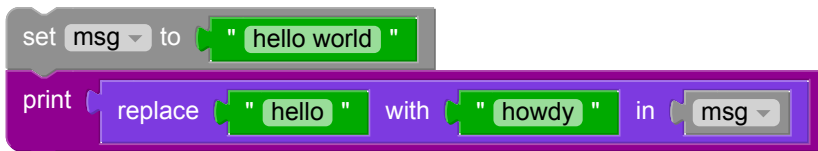
3.3.1. Replacing parts of a string

You can replace *part* of a string (*substring*) with another string using the **replace** block. This works for single characters:



heXXo worXd

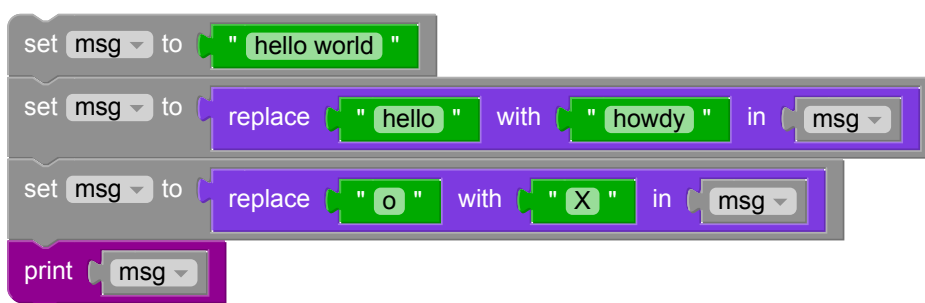
Every **l** is replaced by an **x**. It also works for longer substrings:



howdy world

3.3.2. Replacing multiple substrings

Just like case conversion, **replace** doesn't change the original string. If you store the new version back in the variable, you can replace multiple substrings:



Here, **hello** is replaced with **howdy**, and then **o** is replaced with **x**. This means the **o** in **howdy** also gets replaced, giving:

hXwdy wXrld

3.3.3. Problem: TELEGRAM STOP



Before telephones were invented, the only way to communicate quickly over long distances was by [telegraph](https://en.wikipedia.org/wiki/Telegraphy) (<https://en.wikipedia.org/wiki/Telegraphy>). Short messages known as [telegrams](https://en.wikipedia.org/wiki/The_Telegram) (https://en.wikipedia.org/wiki/The_Telegram) were sent over wires in [Morse Code](https://en.wikipedia.org/wiki/Morse_code) (https://en.wikipedia.org/wiki/Morse_code).

Telegrams were usually written all in uppercase letters, and instead of a full stop (.) they would write **STOP**.

Write a program which converts a message into a telegram by changing it to uppercase and replacing the full stops with the word **STOP** including a space before it. Here is an example:

Message: I will visit in April.
I WILL VISIT IN APRIL STOP

Telegrams would cost money per word, so they are usually short:

Message: French trip is awesome. See you soon.
FRENCH TRIP IS AWESOME STOP SEE YOU SOON STOP

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing a message with no stops in it.
- ☐ Testing a message with many stops.
- ☐ Testing a message with extra stops.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

3.3.4. Problem: Trackwork



Every so often, train tracks need maintenance or repairs. When that happens trains are replaced by buses. Write a program to help people know how to get where they're going.

Write a program which asks the user for their plans and replaces the word **train** with the word **bus**. Here is an example:

What are your plans? Catching the train to the zoo.
Catching the bus to the zoo.

Here is another example:

What are your plans? Take a train to Central then train to Redfern.
Take a bus to Central then bus to Redfern.

Sometimes there might be a funny outcome:

What are your plans? I am going to soccer training.
I am going to soccer busing.

Testing

- ☐ Testing the words in the first example.
- ☐ Testing the punctuation in the first example in the question.
- ☐ Testing the spaces and capitalisation of the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a message with many trains.
- ☐ Testing a message with no trains.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

4

PROJECT 1

4.1. Making simple games

4.1.1. Making simple games

Let's put what we've learnt so far into practice! In each of the Project modules in this course, we're going to write word games.

In this project, we'll start off with [Mad Libs](https://en.wikipedia.org/wiki/Mad_Libs) (https://en.wikipedia.org/wiki/Mad_Libs). Mad Libs is a game where one player asks the other for answers that fit a specific category, and then those answers are substituted in for blanks in a story.

We'll work up to more and more complex games in each project, and each project will start off small and build to something bigger.

4.1.2. Problem: Blank is the new blank



A common saying in fashion is to describe a new fashion colour or trend as "the new black". You can sometimes even see phrases like "[x is the new y](https://snowclones.org/2007/07/01/x-is-the-new-y/)" (<https://snowclones.org/2007/07/01/x-is-the-new-y/>): [Quiet is the new Loud](https://en.wikipedia.org/wiki/Quiet_Is_the_New_Loud) (https://en.wikipedia.org/wiki/Quiet_Is_the_New_Loud), [Bacon is the new Chocolate](https://www.theatlantic.com/magazine/archive/2005/11/better-bacon/304326/) (<https://www.theatlantic.com/magazine/archive/2005/11/better-bacon/304326/>) or even [Knitting is the new yoga](http://www.telegraph.co.uk/men/thinking-man/10552983/Mens-knitting-is-it-the-new-yoga.html) (<http://www.telegraph.co.uk/men/thinking-man/10552983/Mens-knitting-is-it-the-new-yoga.html>).

Write a program that reads in two things, and prints out the resulting *new* phrase.

Here is an example:

```
x: Orange
y: Black
Orange is the new Black
```

```
x: Quiet
y: Loud
Quiet is the new Loud
```

Here's another example:

```
x: knitting
y: yoga
knitting is the new yoga
```

Testing

- ☐ Testing the words in the first prompt message.
- ☐ Testing the words in the second prompt message.
- ☐ Testing the capitalisation of the prompts.
- ☐ Testing the punctuation in the prompts.
- ☐ Testing the spaces in the prompts.
- ☐ Testing the words in the match line.
- ☐ Testing the punctuation, spaces and capitals in the match line.
- ☐ **Testing the first example in the question.**
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing another new comparison.
- ☐ Testing a hidden case.

4.1.3. Problem: Mad Libs 1: Letters from camp!



Mad Libs (https://en.wikipedia.org/wiki/Mad_Libs) are funny, often nonsensical stories built by asking someone for a series of words and using those words to fill in the blanks in a story. There are often crazy results!

Every time kids go away to camp, people expect them to send postcards. But there's more fun outside than being stuck inside writing! Write a program that will help your friends tell fun stories about what they're doing at camp. Your program should ask the user for some things you might find at camp and turn them into a story. For example:

Here is an example:

```
Name: Jane
Relation: Aunty
Noun: stick
Animal (plural): sloths
Sport: tennis
Adjective: bouncy
Another Adjective: crunchy
Verb: hop
Dear Aunty,
Camp stick has been bouncy so far!
Tomorrow we will play tennis if the weather is ok.
Today it has been raining cats and sloths all day!
If we cannot play tennis, maybe we will just hop.
I am sure it will be crunchy either way!
See you soon! Jane
```

Hint

Don't forget you can copy and paste the text for long passages to save on typing. You will need to use string addition for bits that don't have any spaces between the variable and the punctuation



Testing

- ☐ Testing the words in the prompt messages.
- ☐ Testing the capitalisation in the prompt messages.
- ☐ Testing the punctuation in the prompts.
- ☐ Testing the spaces in the prompts.
- ☐ Testing the words and spaces in the first line of the letter.
- ☐ Testing the capitalisation and punctuation in the first line of the letter.
- ☐ Testing the words and spaces in the second line of the letter.
- ☐ Testing the capitalisation and punctuation in the second line of the letter.
- ☐ Testing the words and spaces in the third line of the letter.
- ☐ Testing the capitalisation and punctuation in the third line of the letter.

- ☐ Testing the words and spaces in the fourth line of the letter.
- ☐ Testing the capitalisation and punctuation in the fourth line of the letter.
- ☐ Testing the words and spaces in the fifth line of the letter.
- ☐ Testing the capitalisation and punctuation in the fifth line of the letter.
- ☐ Testing the words and spaces in the sixth line of the letter.
- ☐ Testing the capitalisation and punctuation in the sixth line of the letter.
- ☐ Testing the words and spaces in the seventh line of the letter.
- ☐ Testing the capitalisation and punctuation in the seventh line of the letter.
- ☐ **Testing the whole first example in the question.**
- ☐ Testing another example.
- ☐ Testing another example.
- ☐ Testing a hidden case.

4.1.4. Problem: Mad Libs 2: Bork Bork Bork!



Let's keep building up our [Mad Libs](https://en.wikipedia.org/wiki/Mad_Libs) (https://en.wikipedia.org/wiki/Mad_Libs)! The [Swedish Chef](https://en.wikipedia.org/wiki/Swedish_Chef) (https://en.wikipedia.org/wiki/Swedish_Chef) is a character from the Muppets who does hilarious things to food. His accent is quite thick, and he is often almost unintelligible.

Write a program to write out a three ingredients recipe in *Swedish Chef speak*. You should read in three ingredients from the user and replace all occurrences of the letters 'th' with the letter 'z'.

For this question, Swedish Chef's recipes are always the same (with different ingredients), and they always ends in 'Bork! Bork! Bork!'

Here is an example:

```
Ingredient 1: pastry
Ingredient 2: chicken thighs
Ingredient 3: beans
First cut ze pastry into triangles.
Zen smash ze chicken zighs wiz a hammer.
Fry a mix of pastry and chicken zighs stirring gently.
Add in ze beans one ladle at a time.
Sprinkle ze remaining pastry over ze top.
Bork! Bork! Bork!
```

Here is another example:

```
Ingredient 1: onion
Ingredient 2: wheat thins
Ingredient 3: emmenthaler cheese
First cut ze onion into triangles.
Zen smash ze wheat zins wiz a hammer.
Fry a mix of onion and wheat zins stirring gently.
Add in ze emmenzaler cheese one ladle at a time.
Sprinkle ze remaining onion over ze top.
Bork! Bork! Bork!
```

Hint

Don't forget you can copy and paste the text for long passages to save on typing.

Testing

- ☐ Testing the words in the three prompt messages.
- ☐ Testing the capitalisation in the prompts.
- ☐ Testing the punctuation in the prompts.
- ☐ Testing the spaces in the prompts.
- ☐ Testing the words in the first line, **cutting into triangles**.
- ☐ Testing the words in the second line, **smashing with a hammer**.
- ☐ Testing the words in the third line, **frying the mix**.
- ☐ Testing the words in the fourth line, **adding ladles**.
- ☐ Testing the words in the fifth line, **sprinkling the garnish**.

- ☐ Testing the capitalisation, spaces and punctuation in the first line, **cutting into triangles**.
- ☐ Testing the capitalisation, spaces and punctuation in the second line, **smashing with a hammer**.
- ☐ Testing the capitalisation, spaces and punctuation in the third line, **frying the mix**.
- ☐ Testing the capitalisation, spaces and punctuation in the fourth line, **adding ladles**.
- ☐ Testing the capitalisation, spaces and punctuation in the fifth line, **sprinkling the garnish**.
- ☐ Testing the last line, **Bork! Bork! Bork!**
- ☐ **Testing the whole first example in the question.**
- ☐ Testing the second example in the question.
- ☐ Testing another example.
- ☐ Testing another example.
- ☐ Testing a hidden case.

5

MAKING DECISIONS

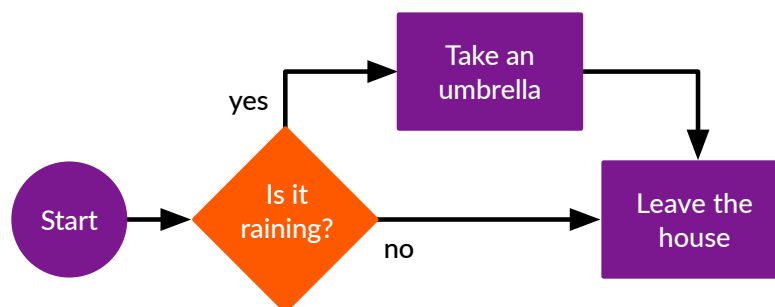
5.1. Making decisions

5.1.1. Why do we need decisions?

So far our programs have been just a sequence of steps that run from top to bottom. The programs *run the same way every time*.

In the real world, we **decide** to **take different steps** based on our situation. For example, if it's raining, we do an *extra step* of taking an umbrella before leaving the house.

This *flowchart* describes this process (or *algorithm*):



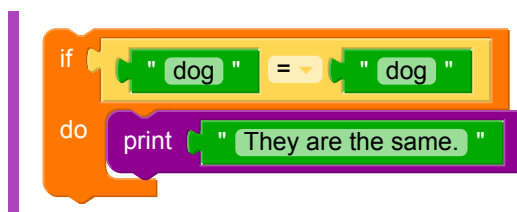
The diamond requires a **yes** or **no** decision. The answer determines which line we follow. If the answer is **yes**, we do the extra step of taking an umbrella. If the answer is **no**, we skip it.

We can write this in Blockly using an **if** block.

5.1.2. Are two strings the same?

The computer decides what steps to run with an **if** block.

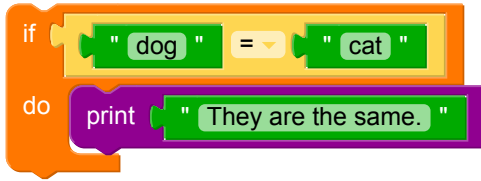
If the two strings **are the same** then the message is printed:



They are the same.

"dog" is equal to (the same as) "dog", so the **print** is run.

If the strings are **not equal**, then the message is not printed:



The **print** is not run because **"dog"** is not *equal* to **"cat"**.

💡 **Huh, these run the same way each time!?**

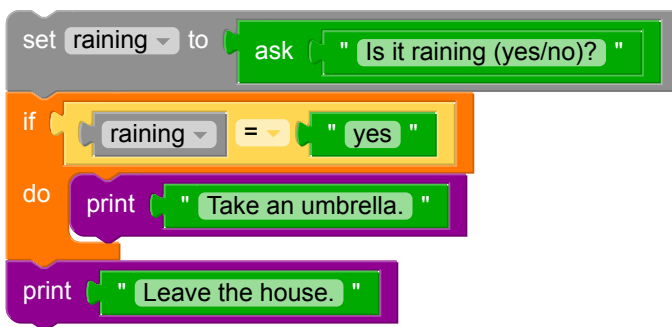
Yes, but we don't normally compare two string blocks. These examples just show you how **if** blocks work.

We normally use **if** blocks with variables. The **if** runs the **do** part only if the value in the variable matches the string.

5.1.3. What if it is raining?

Now we can write a program for our *flowchart*!

We can check if the contents of **raining** is equal to **"yes"**:



Try it! What happens when you say yes, no, or any other answer?

If the user enters **yes**, the **raining** variable will be equal to **"yes"** and so **Take an umbrella.** is printed. If the user enters **no**, or anything else, it isn't printed.

The second **print** always runs, because it isn't inside the **if** block.

💡 **An if block is a control structure**

The **if** block *controls* the program by deciding which blocks run.

5.1.4. Controlling more blocks

You can put as many blocks inside the **if** block as you want. The code *controlled* by the **if** (in the **do** part) is called its *body*:



Here, we ask the user what food they like. If they answer **cake**, the value in **food** will be equal to **"cake"**, and *both* **print** blocks in the body will run.

If they answer anything else, both the **print** blocks will be skipped.

Move a **print** outside of the **if** body to see the difference.

5.1.5. Problem: Cheap Tuesdays



Your local cinema has a special on Tuesdays, so tickets are cheap! But you want to hang out with your friends, so you are happy to go on any day.

Write a program to organise a day. If it's **Tuesday**, the program should print **Great! Tuesdays are cheap.** For example:

```
When can we see a movie? Tuesday
Great! Tuesdays are cheap.
I want to see Finding Dory.
```

If it's not **Tuesday**, you should still go to the movies:

```
When can we see a movie? Wednesday
I want to see Finding Dory.
```

Any answer other than **Tuesday** should work the same way:

```
When can we see a movie? Saturday
I want to see Finding Dory.
```

Testing

- ☐ Testing the words in the input prompt.
- ☐ Testing the capital, punctuation and spaces in the input prompt.
- ☐ Testing the first example in the question.
- ☐ Testing the punctuation, spaces and capital letters in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a different day of the week (**Monday**).
- ☐ Testing a non-day name (**any day**).
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

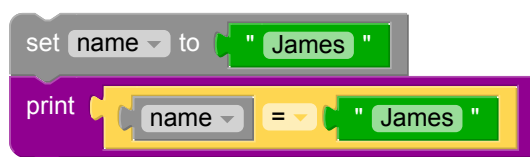
5.2. Decisions with two options

5.2.1. True or False?

An **if** block allows your program to make *yes or no* decisions. In programming, yes is called **True**, and no is called **False**.

The body of the **if** block runs when the comparison (the **=** block) is **True**. It doesn't run if the comparison is **False**.

Calculations that result in **True** or **False** values are called *Boolean expressions*. We can print their value directly:



True

If we change **name**, the conditional expression will be **False**:

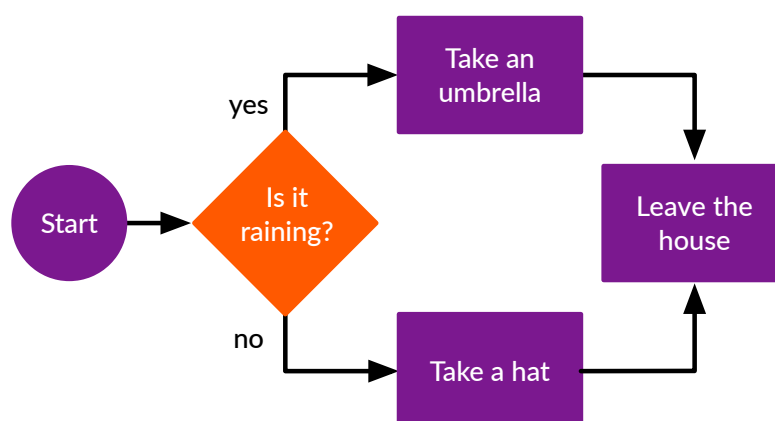


False

5.2.2. Decisions with two options

Our examples so far have done something when the Boolean expression is **True**, but nothing when the expression is **False**.

In the real world, we often want to do one thing for the **True** case, and something different for the **False** case.



If the user says **yes**, it is raining, then the program should respond **Take an umbrella**. but otherwise it should respond **Take a hat**.

What we want is an extra part to the **if** block which only run when the Boolean expression is **False**.

5.2.3. If it isn't raining...

Let's implement the two options in the new flowchart using a **if** block with an **else** part:

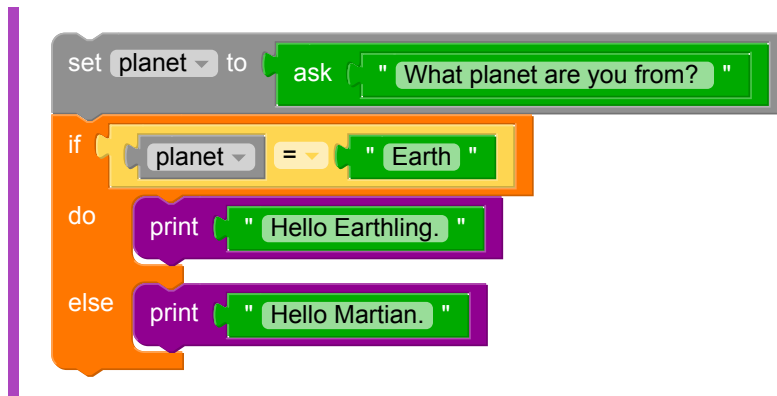
Here, either the first or second **print** block will be run, depending on the answer, **but not both**.

The third **print** always runs, since it is outside of the **if** body, and so is not controlled by the **if** block.

5.2.4. When options are limited

The **if** / **else** block is great for making decisions with only two options (friend or foe, wet or dry, day or night). When there are more than two possibilities, it can get confusing.

This program tries to decide if you are an **Earthling** or **Martian**:



If the user enters **Earth** or **Mars** the response is correct. However, if the user enters **Jupiter** the response is wrong (try it)!

💡 **else catches everything else!**

Remember: the **else** runs whenever the Boolean expression is **False**. Here, we were expecting **Earth** or **Mars**, but anything that is not **Earth** will also return **False**.

5.2.5. Problem: Pikachu, I choose you!



[Pokémon Go](https://en.wikipedia.org/wiki/Pok%C3%A9mon_Go) (https://en.wikipedia.org/wiki/Pok%C3%A9mon_Go) was downloaded over 100 million times in a month. We all have friends obsessed with Pokémon – and one of the favourites is Pikachu!

Write a program to talk to your [Pikachu](https://en.wikipedia.org/wiki/Pikachu) (<https://en.wikipedia.org/wiki/Pikachu>)-obsessed friend.

Your program should ask the user for their favourite Pokémon. If they say **Pikachu**, your program should print **Me too!** like this:

```
Which is your favourite? Pikachu
Me too!
```

If they say anything else, it should print **I like Pikachu.** instead:

```
Which is your favourite? Squirtle
I like Pikachu.
```

Testing

- ☐ Testing the words in the input prompt.
- ☐ Testing the capital, punctuation and spaces in the input prompt.
- ☐ Testing the words in first example in the question.
- ☐ Testing the punctuation, spaces and capital letters in the first example.
- ☐ Testing the words in the second example in the question.
- ☐ Testing the punctuation, spaces and capital letters in the second example.
- ☐ Testing with **Doduo**.
- ☐ Testing **Raichu**.
- ☐ Testing a typo.
- ☐ Testing a hidden case.

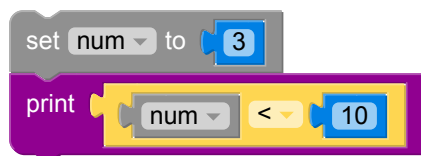
5.3. Decisions about numbers

5.3.1. How do we compare things?

So far we have only checked whether two strings are equal. However, there are other *comparison operators*, including:

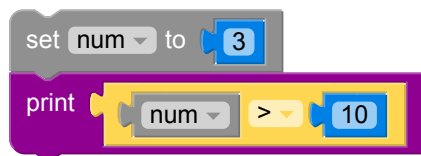
Operation	Operator
equal to	=
not equal to	≠
less than	<
less than or equal to	≤
greater than	>
greater than or equal to	≥

We can use these to test whether a number is bigger or smaller than other number:



True

This prints **True** because 3 is less than 10.

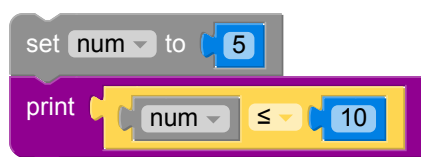


False

This prints **False** because 3 is not greater than 10.

5.3.2. Experimenting with comparison

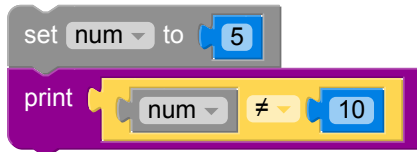
Let's try some more examples to see how conditional operators work. Firstly, we have less than or equal to (\leq):



True

Setting **num** up to *and including* 10 will give **True**. Setting **num** greater than 10 will give **False**. Greater than or equal to (\geq) works in reverse. Change the example and see for yourself!

Another important operator is not equal to (\neq):

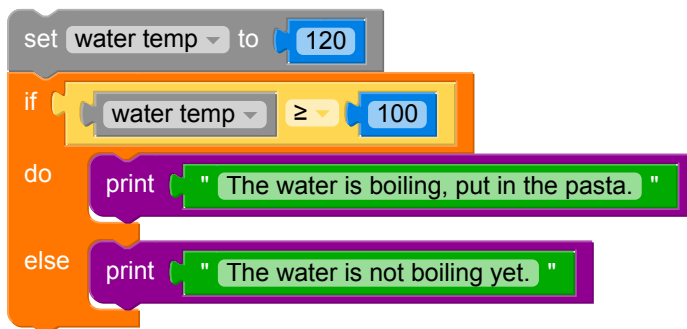


True

Notice this program prints **True** because 5 is not equal to 10. This can be a bit confusing — see what happens if you change the value of **num** to 10.

5.3.3. Making decisions with numbers

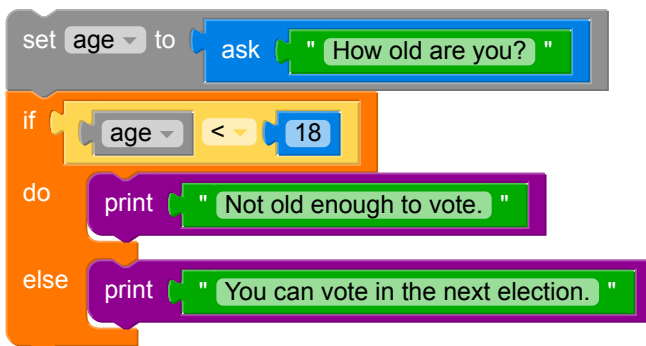
Now we can write programs that make decisions based on numbers. The example below makes a decision based on the water temperature:



The water is boiling, put in the pasta.

Change the value in **water temp** to see what happens.

We can do something similar with numbers entered by the user:



5.3.4. Problem: Soil Sensor



You're creating an automatic system to water your plants. It has a sensor to measure the moisture in the soil. If the moisture level is below 25%, the plant should be watered.

Write a program to read in the moisture level from the sensor. If the moisture level is below 25%, it should do this:

```
Moisture level: 18
Water the plants.
```

If the moisture level is 25% or above, it should do this:

```
Moisture level: 27
The plants are happy.
```

Ideal soil moisture levels (https://en.wikipedia.org/wiki/Soil#Soil_moisture_content) are actually more complicated!

Testing

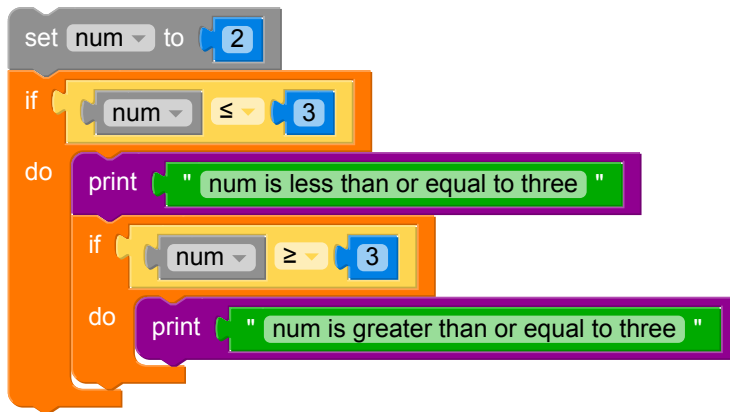
- ☐ Testing the words in the input prompt.
- ☐ Testing the capital, punctuation and spaces in the input prompt.
- ☐ Testing the words in first example in the question.
- ☐ Testing the punctuation, spaces and capital letters in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing that the program uses integers (with moisture level **100**).
- ☐ Testing a case where the plants are very dry.
- ☐ Testing the case with the minimum moisture level required to not water the plants.
- ☐ Testing the case where you only just need to water the plants.
- ☐ Testing a case where there is no moisture in the soil.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

5.4. Making complex decisions

5.4.1. Decisions within decisions

The *body* of an **if** block can contain another **if** block. This is called *nesting* or *nested if blocks*.

In the program below, we have one **if** block inside another one. That means the second comparison will only be tested if the first comparison is **True**.

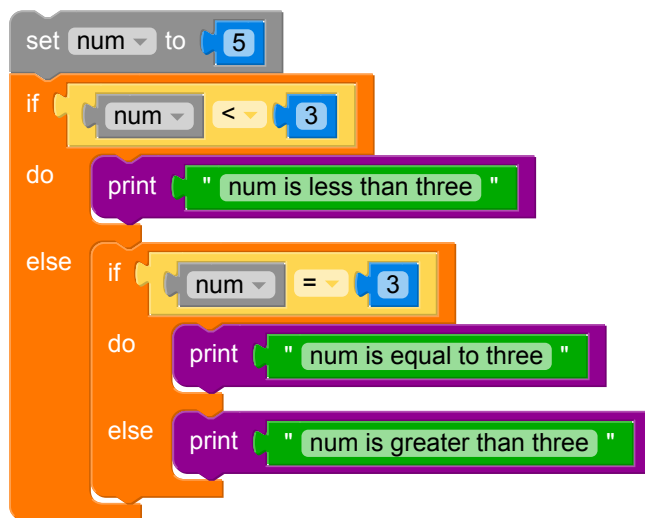


num is less than or equal to three

There is only one value for **num** that will cause both messages to be printed. Can you work out what it is

5.4.2. Decisions with multiple options

Continuing our previous example, if we want to print each case (*less than*, *equal to*, or *greater than*) separately, we need to change both **if** blocks to use an **else** part, like this:



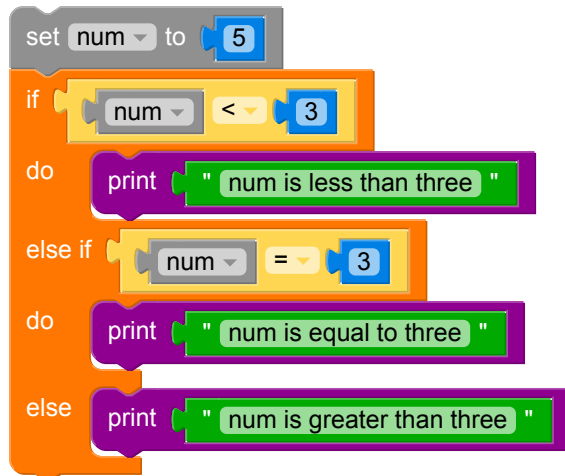
num is greater than three

Nesting gets messy when we've got lots of different cases to test.

5.4.3. Decisions with elif

Another, neater, way of making multiple decisions is to use an **elif** block. **elif** is a combination of the words **else** and **if** together.

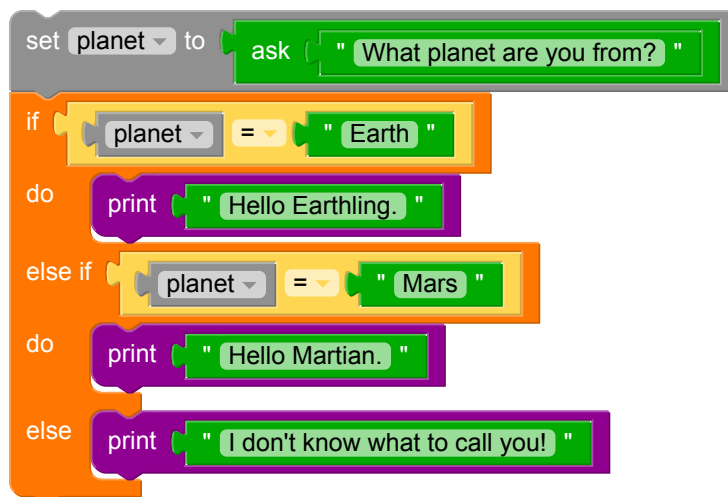
Each Boolean expression is evaluated in order, from top to bottom. It stops at the first expression to return **True** and runs its **do** part. If none of the expressions return **True**, then the **else** part is run:



num is greater than three

5.4.4. Interplanetary visitor

Remember our program that tried to greet interplanetary visitors correctly? We can now extend this so that it makes sense regardless of what planet the user is from.



5.4.5. Problem: Não compreendo?



You're in Brazil for the 2016 Summer Olympic Games! You're going to need a few Portuguese words to help you get around.

Write a program to translate some useful words from [Portuguese](https://en.wikipedia.org/wiki/Portuguese_language) (https://en.wikipedia.org/wiki/Portuguese_language). Here are the words you need:

Portuguese	English
Ola	Hello
Sim	Yes
Nao	No

Your program should ask for the Portuguese word then print out the English translation. For example:

```
What did they say? Ola
Hello
```

Here is another example:

```
What did they say? Sim
Yes
```

If the Portuguese word isn't one of the three above, your program should print **Nao compreendo.** which means "I don't understand." in Portuguese. For example:

```
What did they say? Desculpe
Nao compreendo.
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the Portuguese word for **No**.
- ☐ Testing an different Portuguese phrase (**Por favor**, which means *please*).
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

6

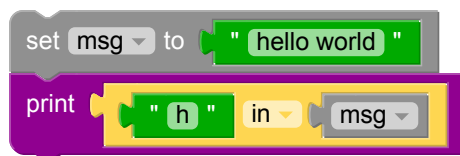
INVESTIGATING STRINGS

6.1. Investigating strings

6.1.1. Character *in* a string

We can check if two *whole* strings match, using `=`. But, what if we want to know if *part* of a string matches?

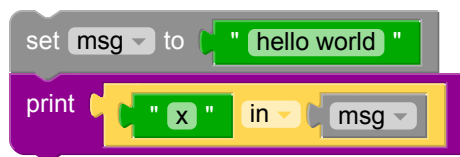
The `in` block checks if a string *contains* a character:



True

This prints `True`, because `"h"` is contained in `"hello world"`.

When the string doesn't contain the character, `in` gives `False`:

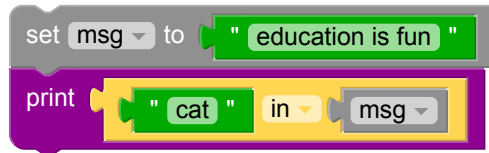


False

This prints `False` because there is no `"x"` in `"hello world"`.

6.1.2. String *in* a string

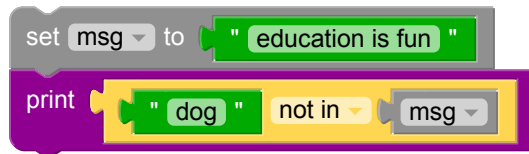
We can use `in` to check for smaller strings (*substrings*) in a string. A substring can be part of a word or phrase. For example:



True

This prints **True** because the substring "cat" appears in msg (in edu**cat**ion). Try changing "cat" to something else.

The **not in** block does the opposite to **in**. It is **True** when the string does *not* contain the substring:

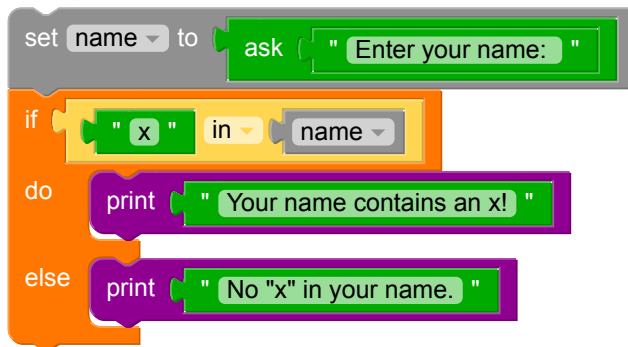


True

Click on **in** to change the block to **not in**.

6.1.3. Making decisions with strings

We can use these new comparisons on strings to make decisions. Let's check if a person's name contains the letter x, e.g. Alex.



This code only finds lowercase x's (try Xena). We'll fix this soon!

6.1.4. Problem: Forgotten Attachment?



Have you ever tried to send an email with an attachment but forgot to attach the file? Gmail and Outlook can check this for you.

Write a program that checks your email for the string `attach`, and if it's there, reminds you to attach the file. Here is an example:

Email: I will attach the document to this email.
Did you remember the attachment?

If `attach` doesn't appear in the email, then print that it was sent:

Email: Hi, how are you going?
Sent.

`attach` can appear *anywhere* in the email, even in a longer word:

Email: I have attached a photo.
Did you remember the attachment?

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing another email with a missing attachment.
- ☐ Testing an email with no attachment.
- ☐ Testing a hidden case
- ☐ Testing another hidden case.

6.2. Capitalisation and comparisons

6.2.1. When an 'apple' is not an 'APPLE'

To a computer, `"a"` and `"A"` are completely different *characters*, even if we interpret them as the same *letter*, just lower and upper case:

```
print "a" = "A"
```

False

So testing if `"apple"` is the same as `"APPLE"` will also be **False**:

```
print "apple" = "APPLE"
```

False

We'll have to **modify** the strings so that we compare only lower case letters with lower case letters. (Or, only upper case with upper case!) We've seen how to change the case of a string using `lower` and `upper`:

```
set word to "Apple"
print to lower case word
```

apple

6.2.2. Comparing strings but ignoring case

To compare whether strings are equal *ignoring case*, we have to change the strings. That way, we can compare `lowercase` with `lowercase` or `UPPERCASE` with `UPPERCASE` text.

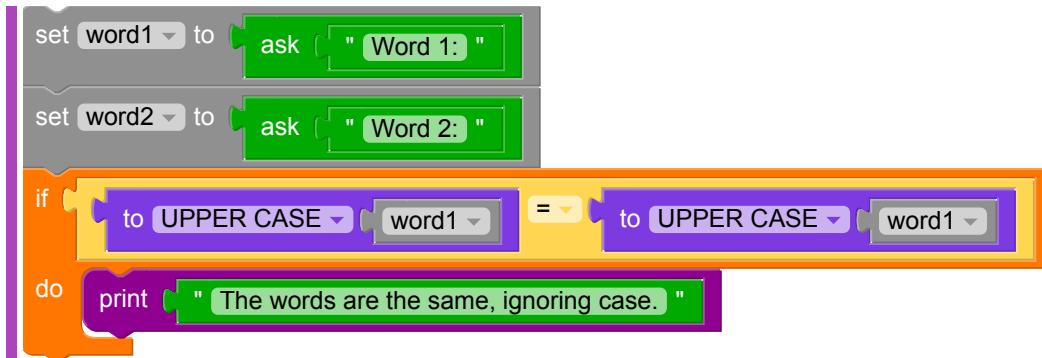
In this example, we make new variables: `lower_word1` and `lower_word2`, and then compare them:

```
set word1 to ask "Word 1: "
set word2 to ask "Word 2: "
set lower_word1 to to UPPER CASE word1
set lower_word2 to to UPPER CASE word1
if lower_word1 = lower_word2
do print "The words are the same, ignoring case."
```

Word 1: banana
Word 2: BANANA
The words are the same, ignoring case.

6.2.3. Comparing strings easily

To compare strings, we could also just compare the **lowercase** versions of the two strings:



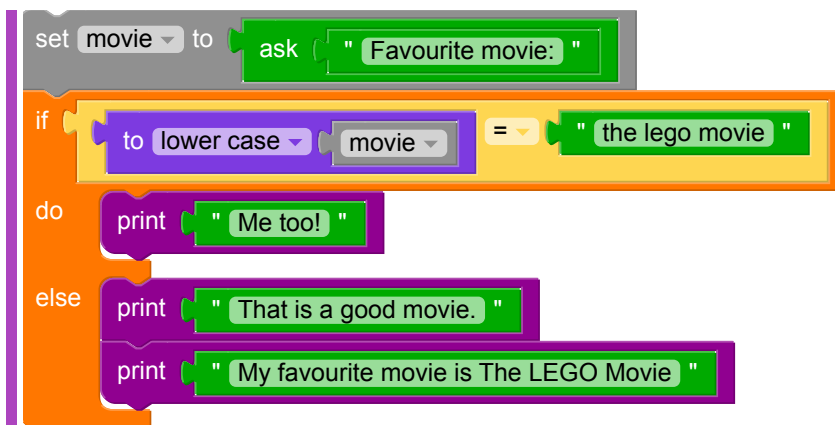
Word 1: abc
Word 2: ABC
The words are the same, ignoring case.

6.2.4. A case study...

Here's an example to show how this can work in a problem. Let's write a program to ask the user their favourite movie.

If it's the same as yours (The LEGO Movie) let's say **Me too!** Otherwise, let's say **That is a good movie.** and tell them what our favourite is.

We want to accept answers like The LEGO Movie, the lego movie or even the Lego MOVIE.



Favourite movie? the Lego movie
Me too!

Try out the example with a few different examples!

💡 Comparing cases

Because we're using **lowercase** on the string the user types in, we have to make sure what we compare it to is also **lowercase**!

We could do something like this:



That way both sides are definitely **lowercase** !

6.2.5. Problem: Need a Hug?



Before there were emojis to help you express yourself in emails and texts, there was a text based system called [emoticons](https://en.wikipedia.org/wiki/Emoticon#Japanese_style) (https://en.wikipedia.org/wiki/Emoticon#Japanese_style). In Western countries users generally had to tilt their head to the side to see the expression of the emoticon but in Japan they developed [kaomoji](https://en.wikipedia.org/wiki/Emoticon#Japanese_style) (https://en.wikipedia.org/wiki/Emoticon#Japanese_style).

Write a program to give your friend a kaomoji hug if they need one. For example:

```
Do you need a hug? Yes
\(^-^)/
Have a great day!
```

It should work regardless of whether they used upper or lower case or a mix of both. For example:

```
Do you need a hug? YES
\(^-^)/
Have a great day!
```

If they don't need a hug you should still wish them a great day:

```
Do you need a hug? no
Have a great day!
```

Any answer other than **yes** (with any capitalisation) should work the same way:

```
Do you need a hug? nup
Have a great day!
```

Testing

- ☐ Testing the words in the input prompt.
- ☐ Testing the capital, punctuation and spaces in the input prompt.
- ☐ Testing the first example in the question.
- ☐ Testing the punctuation, spaces and capital letters in the first example.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the fourth example in the question.
- ☐ Testing a case that isn't yes, but contains those letters.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

6.3. Characters in a string

6.3.1. Getting an individual character

We can get a character from a string using the *indexing* block. This gets the first character:



h

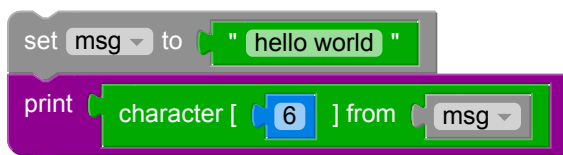
In programming, we start the index from 0 rather than from 1.

So the *first* character in a string is at index 0, the *second* is at index 1, the *third* is at index 2, and so on.

This means **hello world** is indexed like this:

0	1	2	3	4	5	6	7	8	9	10
h	e	l	l	o		w	o	r	l	d

and so, if we want to access the **w** of **world**, we use:



w

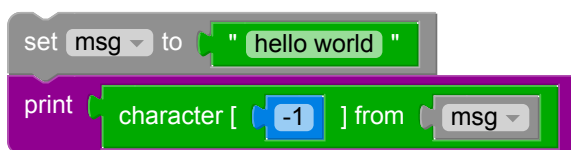
6.3.2. Character counting from the end

Just as we can count *up* to get each letter from the beginning of the word, we can count **down** to get letters from the **end** of a word.

This means **hello world** is indexed in either of these two ways:

0	1	2	3	4	5	6	7	8	9	10
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
h	e	l	l	o		w	o	r	l	d

and so, if we want to access the **d** of **world**, we use:

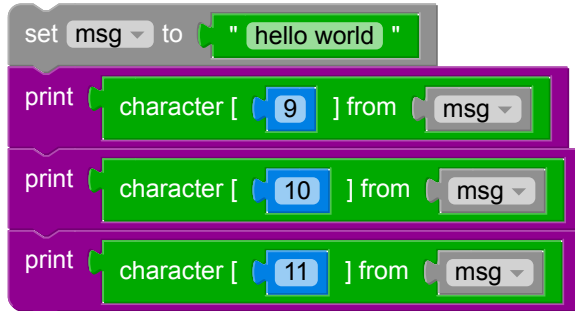


d

6.3.3. Characters that don't exist

If you try to get an index past the end of the string, Blockly will give an **IndexError** error and your program will crash.

Make sure you only get an index in the string that exists. Here is an example of what happens when you access an index beyond the end of the string:



```

1
d
Traceback (most recent call last):
  File "program.py", line 4, in <module>
    print(msg[11])
IndexError: string index out of range

```

The string **hello world** only has 11 characters — indexed 0 to 10.

We try to get index 11, which doesn't exist. This means the last **print** block fails and we get an error.

6.3.4. Problem: Double Or Nothing



Only a small number of English words start with a double letter! Words like **ee**rie, **ll**ama, and **oo**ze.

Write a program that checks if the **first letter** and the **second letter** of a word are the same character.

Here's an example:

Word: **ee**rie
Starts with a double letter!

If the first two letters are different, your program should do this:

Word: **dog**
No double letter at the start.

Double letters don't matter unless they're the first two, like this:

Word: **foot**
No double letter at the start.

Your program will only be tested on words at least 2 letters long.

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example from the question.
- ☐ Testing a word that starts with **oo**.
- ☐ Testing a word that starts with **ee**.
- ☐ Testing a word that starts with **ll**.
- ☐ Testing a word with no double letter at the start.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.
- ☐ Testing yet another hidden case.

6.3.5. Problem: Email Address



You're creating an email address for all the new students at your school. Each email address is created using the first letter of the student's first name and then their last name.

Write a program that takes the first and last name of the student, and prints out their email address. For example:

```
First name: Stephen
Last name: Merity
Your email address is smerity@example.school.edu
```

All of the letters in the email address should be **lowercase**.

Here is another example student at Example School:

```
First name: Jasmine
Last name: Constable
Your email address is jconstable@example.school.edu
```

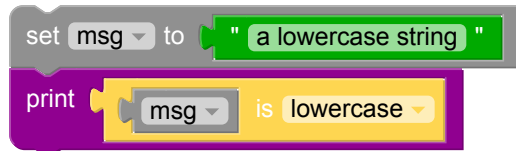
Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example from the question.
- ☐ Testing a magical email address.
- ☐ Testing a vampiric email address.
- ☐ Testing a hairy email address.
- ☐ Testing a hidden case.

6.4. Checking string case

6.4.1. Checking the case of a string

We can also check the capitalisation of a string with **is lowercase** :



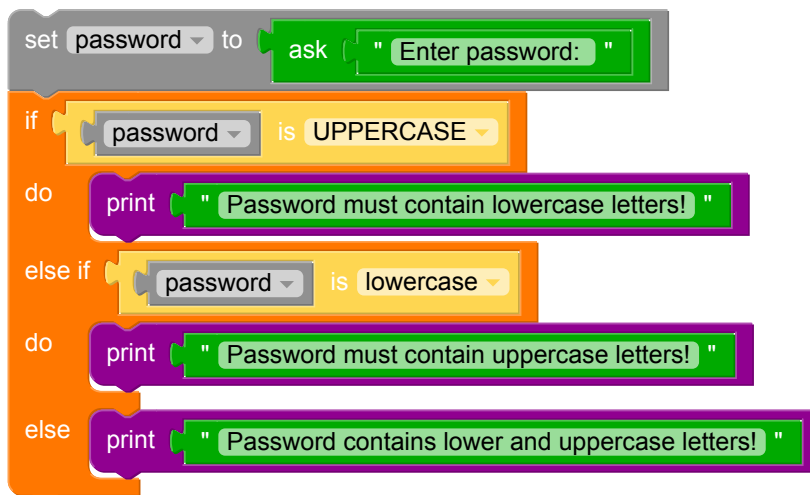
True

It can test for lower, UPPER and Title case, just like the conversion block. Try changing the **msg** string and the case.

We can use these to check that a password contains lower and uppercase letters. Try setting **msg** to **"MiXeD CaSe"**.

6.4.2. Making decisions about case

We can check the case in an **if** block to make decisions. To check if a password contains lower and uppercase letters, we can use:



For converting and checking capitalisation, the punctuation, digits, and spaces are ignored.

6.4.3. Problem: Capital Cities



In English, city names have capital letters, but sometimes you forget to type the capital letters!

Write a program to correct this mistake. If the user enters a city name which is all lowercase, your program should capitalise it with `title`. Here is an example:

```
City name: auckland
You forgot the capital! It should be: Auckland
```

If the name is not all lowercase, then we assume that it's correct:

```
City name: San Francisco
Looks fine to me.
```

Here is one more example:

```
City name: brisbane
You forgot the capital! It should be: Brisbane
```

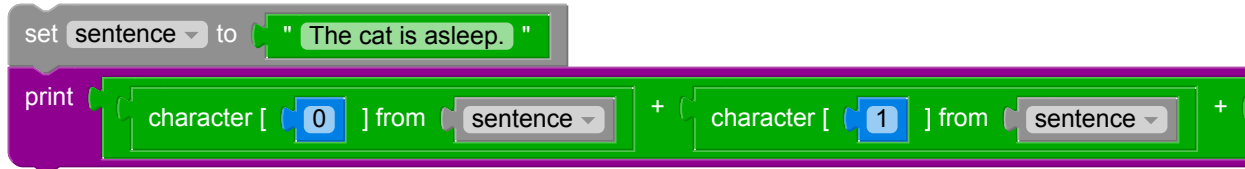
Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a correctly capitalised city name.
- ☐ Testing a lowercase city name.
- ☐ Testing a city name with very strange capitals.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

6.5. Slices and substrings

6.5.1. Slicing a string

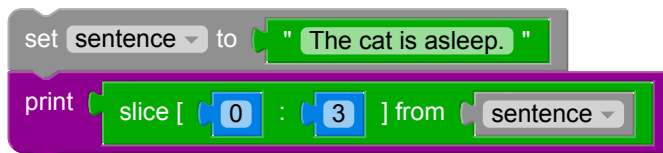
If we want to get the first 3 characters of a string (a *substring*), we can index them one at a time, and then join them:



The

But there must be a better way? There is — it's called a *slice*.

A *slice* takes two numbers that say which characters to get:



The

A slice gets characters from the first index *up to but not including* the second index.

So, slice [0 : 3] gets from char [0] to char [2].

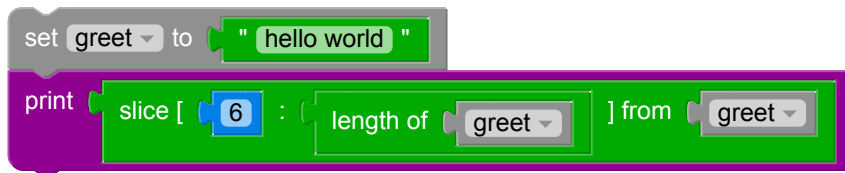
It doesn't include char [3]. Here's another example:



hello
world

6.5.2. Slicing to the end of a string

If we want to slice to *the end of the string*, we need its length:



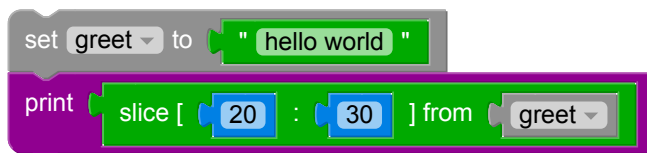
world

Blockly has a simpler block for slicing to the end of the string:



world

What happens if we try to get a slice that doesn't exist?

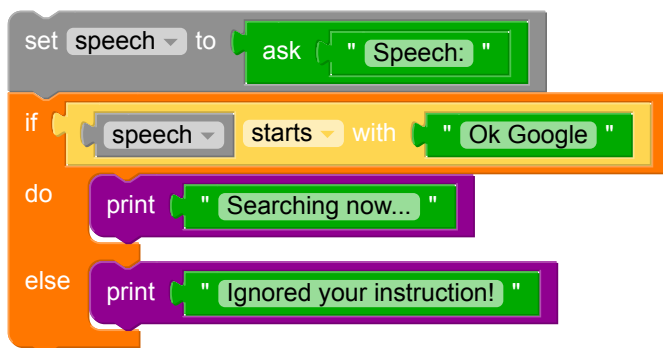


The greeting is shorter than 20 characters so `slice [20 : 30]` doesn't exist. Instead we get an empty string and it prints nothing.

6.5.3. Matching the beginning or end of a string

Last week, we saw how to check if a string starts with a substring.

We can use it to implement Google's voice search on Android:



By clicking on `starts` you can change the block to `ends with`, to compare the end of the string with another string.

6.5.4. Problem: Unprefixed



Lots of words in English start with **un-**, and they usually mean the opposite thing. For example **unwritten** means **not written**.

Write a program which tries to explain what an un-word means. If the word starts with **un** then it should remove the **un** and print this:

```
Enter a word: unwritten
That means not written
```

If the word doesn't start with **un** then it should just say that the word means what it is:

```
Enter a word: written
That means written
```

This rule doesn't work every time, so here's a funny example:

```
Enter a word: under
That means not der
```

And another funny one:

```
Enter a word: unique
That means not ique
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example from the question.
- ☐ Testing the fourth example from the question.
- ☐ Testing a word which has **un** in the middle.
- ☐ Testing another word which has **un** in the middle.
- ☐ Testing a word which has **un** at the start and in the middle.
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

6.6. Congratulations!

6.6.1. Congratulations!

Congratulations, you have just completed this module!

In this module we learned a lot about strings and we covered the following topics:

- Finding if a character or string is inside another string;
- checking if strings are upper or lowercase;
- converting strings to all-caps or lowercase letters;
- finding how long a string is;
- replacing parts of a string;
- picking out characters from a string by number.



PROJECT 2

7.1. More word games

7.1.1. Making simple games

Let's put what we've learnt about strings and decisions into practice making some more complex games.

We're going to add in the concept of decisions and string manipulations. These will help us build up to more complex games as we keep adding more skills to our toolbox.

7.1.2. Problem: Taboo!



[Taboo \(https://en.wikipedia.org/wiki/Taboo_\(game\)\)](https://en.wikipedia.org/wiki/Taboo_(game)) is a word game where one person describes a word on a card so that their partner can guess it. However, there are certain words that the person can't say when trying to describe the thing! A player might have to describe 'cereal' without using the word 'breakfast'.

Write a program to help play Taboo. Here's an example for trying to guess the word 'cereal':

Taboo word: breakfast
Description: A type of grain, like oats or bran.
Safe!

Here's another example, which includes the taboo word:

Taboo word: breakfast
Description: The thing you eat for breakfast.
Taboo!

You should print **Taboo!** even if the word is in another word. Here's an example when trying to guess the word 'wheel':

Taboo word: bike
Description: There's 4 on a car and 2 on a motorbike.
Taboo!

Sometimes games get exciting and players tend to shout. If the word occurs in **any case**, lower, upper or mixed, it still counts. For example, guessing the word 'oval':

Taboo word: sport
Description: WE HAVE SPORTS HERE!
Taboo!

Testing

- ☐ Testing that the words in the prompts are correct.
- ☐ Testing that the punctuation in the prompts is correct.
- ☐ Testing that the capitalisation in the prompts is correct.
- ☐ Testing that the white space in the prompt is correct.
- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the fourth example in the question.
- ☐ Testing a game trying to guess 'water'.
- ☐ Testing another game trying to guess 'water'.
- ☐ Testing a tricky case.
- ☐ Testing a hidden test case (to make sure your program works in general).
- ☐ Testing another hidden case.

- ☐ Testing a tricky hidden case.
- ☐ Testing a final hidden case.

7.1.3. Problem: Word Chain!



[Word chain](https://en.wikipedia.org/wiki/Word_chain) (https://en.wikipedia.org/wiki/Word_chain) is word game where players take turns saying words that start with the last letter of the previous word. You might have played this game on long car trips.

Write a program to help you and your friends play word chain. Your program should read in two words and print out whether they are valid to follow each other. You should ignore case for this game.

Here is an example of a valid pair:

```
Word 1: carrot
Word 2: turnip
Valid
```

Here is another example of an invalid pair:

```
Word 1: orange
Word 2: apple
Nope!
```

The word **apple** is rejected because it doesn't start with the letter **e** from the previous word, **orange**.

Here's one more example, with uppercase and lowercase letters:

```
Word 1: Passionfruit
Word 2: Tangerine
Valid
```

Testing

- ☐ Testing that the words in the prompts are correct.
- ☐ Testing that the punctuation in the prompts is correct.
- ☐ Testing that the capitalisation in the prompts is correct.
- ☐ Testing that the white space in the prompt is correct.
- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing a game with countries.
- ☐ Testing another game with countries.
- ☐ Testing a game with strange case.
- ☐ Testing another game with strange case.
- ☐ Testing a hidden test case (to make sure your program works in general).
- ☐ Testing another hidden case.
- ☐ Testing a final hidden case.

8

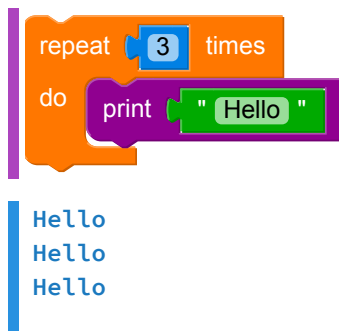
REPEATING THINGS

8.1. Repeating things

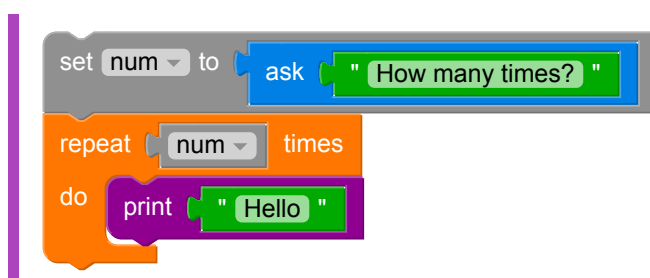
8.1.1. How (not) to repeat yourself

So far, we've learnt how to tell the computer to do specific things like ask for input or make decisions. Now, we'll learn how to make a computer do a specific thing lots of times using the **repeat** block.

In this case, we want to print **Hello** three times:

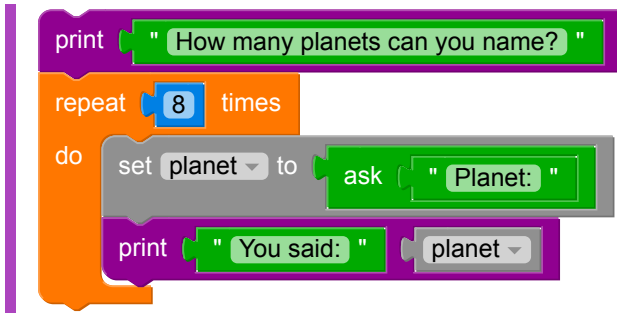


We can also read in a number and use it in the **repeat** block. Here, you can choose how many times to print out **"Hello "**:



8.1.2. When can we stop?

We might want to ask the user a question repeatedly. If we know how many times we want to ask the question, we could try something like this:



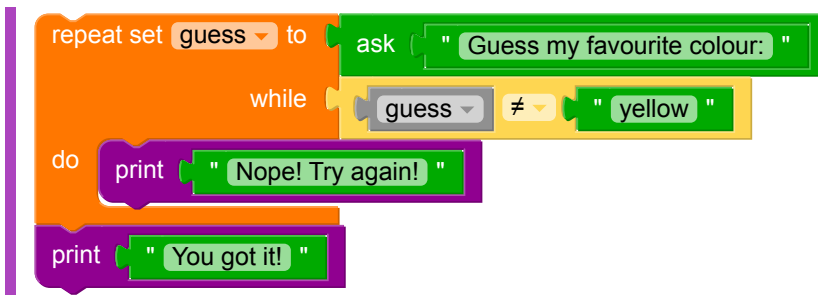
But what if the user only knows 5 planets? Or they know more than 8 (perhaps they didn't get [the news about Pluto](http://news.nationalgeographic.com/news/2006/08/060824-pluto-planet.html) (<http://news.nationalgeographic.com/news/2006/08/060824-pluto-planet.html>))

We need to keep asking until the user runs out of answers.

If we want to loop *until* something happens, **repeat** can't help us. We need a **repeat while** block!

8.1.3. Simplified read loop

Asking the user multiple questions is so common, Blockly has a **repeat ask while** block to make it easier:



The **repeat ask while** block does several steps:

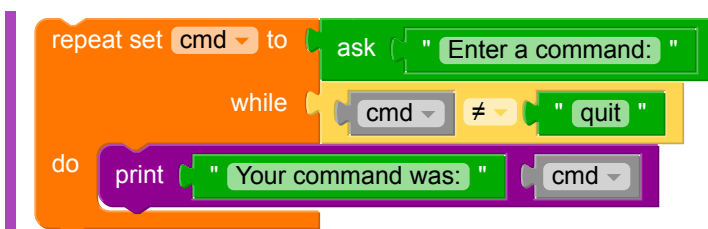
1. **ask** the user for a string;
2. store the answer in **guess**;
3. if the condition is **True** (the user correctly guessed that my favourite colour is **"yellow"**), run the body and go back to step 1;
4. if the condition is **False**, stop looping immediately.

In this example, *while* the guess is not equal to **"yellow"**, it will **print** **"Nope! Try again!"**, and redo the loop.

When the user enters **"yellow"**, the condition is **False**, the loop stops, and the next **print** block is run, telling the user they guessed correctly.

8.1.4. Read until in a read loop

Here's another example using the **repeat ask while** block.



The **repeat ask while** block follows the same steps as the last example. It will keep looping *until* the user enters **"quit"**.

This **repeat ask while** block will keep looping and looping and looping until the condition it is testing is met.

8.1.5. Problem: Snooze No More!



When your alarm goes off, you can hit the *snooze* button and fall back asleep. Write a program that won't let you snooze too long!

Your program should print out **MEEP MEEP MEEP** and then keep asking the user *Are you up?* until the user enters **I am up!**

If the user says anything else, your program should print out **MEEP MEEP MEEP** and ask the user for input again.

Once the user says **I am up!** your program should finish by printing out **Took you long enough!**

For example:

```
MEEP MEEP MEEP
Are you up? Noooo
MEEP MEEP MEEP
Are you up? I'm sleepy
MEEP MEEP MEEP
Are you up? snooze...
MEEP MEEP MEEP
Are you up? snooze...
MEEP MEEP MEEP
Are you up? OK OK!
MEEP MEEP MEEP
Are you up? I am up!
Took you long enough!
```

Here is another example. The program should run until the user enters **I am up!** exactly:

```
MEEP MEEP MEEP
Are you up? Sssssh
MEEP MEEP MEEP
Are you up? i'm up
MEEP MEEP MEEP
Are you up? I am awake!
MEEP MEEP MEEP
Are you up? I am up!
Took you long enough!
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing another simple interaction.
- ☐ Testing when the user gets up straight away.
- ☐ Testing when the user takes a long time to get up.
- ☐ Testing when the input starts with **I am up!** but is not exactly that.
- ☐ Testing a hidden case.

8.2. Reading while True

8.2.1. Comparing just the ends of a string

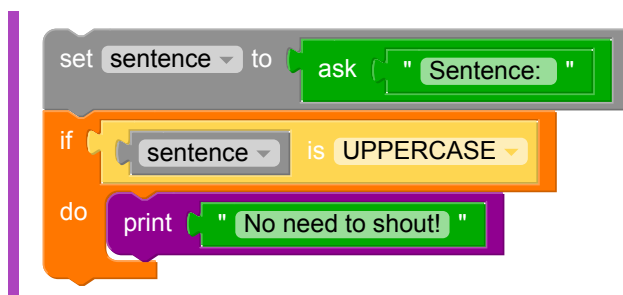
Here's a new logic blocks for comparing the start of a string!

The **starts with** block tests if one string *starts with* the characters from another string:

By clicking on **starts** you can change the block to **ends with**, to compare the end of the string with another string.

8.2.2. IS UPPERCASE?

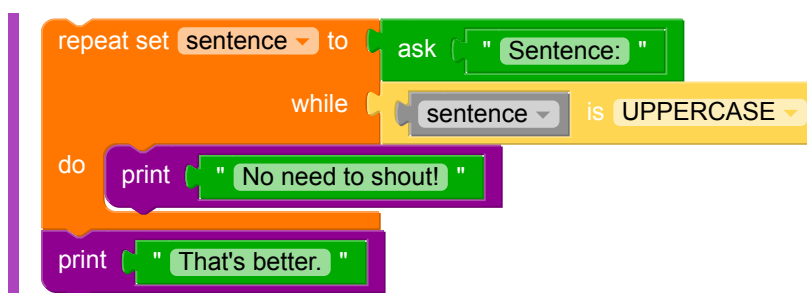
Just a reminder: the **is UPPERCASE** block tests if the letters in a string are all uppercase letters:



The same block can also check for **lowercase** and **TitleCase**.

8.2.3. Looping with **is UPPERCASE**

We can use *any comparison block*, like **is UPPERCASE**, as the **while** loop condition. It will loop *while* the condition is **True**:



This program will keep saying **No need to shout!** until the user enters something that's not in **ALL CAPS**:

```

Sentence: HELLO
No need to shout!
Sentence: OH?
No need to shout!
Sentence: Is this better?
That's better.
  
```

8.2.4. Problem: Um... no... umm...



You've got a big speech coming up, but you say "um" too often!

Write a program to practice your speech, but when you say `um` it will ask you to say it again:

```
Speech: Welcome um, everyone!
You said um!
Speech: Welcome everyone um...
You said um!
Speech: um...
You said um!
Speech: Welcome everyone!
Got it.
```

Your program isn't smart — it catches `um` anywhere, like `umbrella`:

```
Speech: You will need an umbrella!
You said um!
Speech: You should take a raincoat.
Got it.
```

Testing

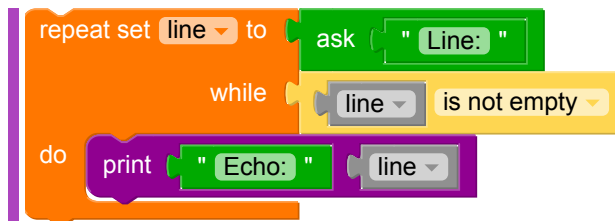
- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing another speech with `um`'s.
- ☐ Testing a speech with no `um`'s!
- ☐ Testing a case with lots of `um`'s!
- ☐ Testing a hidden case.
- ☐ Testing another hidden case.

8.3. Making decisions inside a loop

8.3.1. Looking for blank lines

A common thing you'll need to do when writing programs is to read in multiple lines until a blank line is entered.

Blockly has a special **is not empty** comparison block for this:

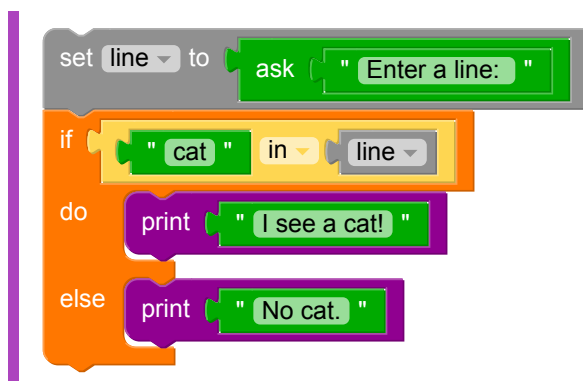


Try this program! You need to answer by pressing Enter (and nothing else) to return an empty string and stop the loop.

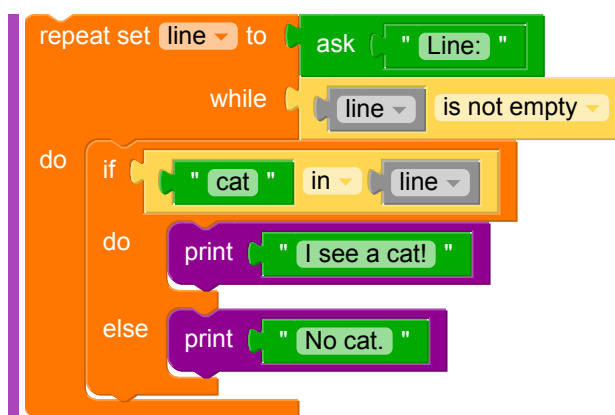
The **is not empty** block is under the **Logic** tab, but we will usually include it directly in the **read ask while** block.

8.3.2. Making decisions inside a loop

Using an **if** block, we can write a program to find cats in a line:



To check lots of lines, we remove the **ask** block and wrap the **if** in a **repeat ask while** block:



Enter a line: the cat is asleep
I see a cat!
Enter a line: cats are everywhere!
I see a cat!
Enter a line: something about a dog
No cat.
Enter a line:

In this example we've put our **if else** block *inside* a **repeat while** loop so that we can use it many times.

8.3.3. Problem: TL;DR



TL;DR (<https://en.wikipedia.org/wiki/TL;DR>) is short for *too long; didn't read*. A reader can use tl;dr to say a story is too long. An author can use tl;dr before a brief summary so readers can decide to read the full story or not.

Write a (rude!) program to complain when a line of input is too long.

Your program should read in multiple lines of input until a blank line is entered. If the line is longer than 30 characters, your program should output **TL;DR**. For lines that are 30 characters or shorter, your program should output **I read it**.

Here is an example of how your program should work:

```
Sentence: This is a short sentence.
I read it.
Sentence: This is a very long sentence and it might not get read.
TL;DR
Sentence: 
```

Here is a longer example:

```
Sentence: Oh my, what a beautiful day it is today!
TL;DR
Sentence: The warm weather is delightful.
TL;DR
Sentence: It is sunny today.
I read it.
Sentence: The weather today's fantastic!
I read it.
Sentence: 
```

 **Reminder: the `length` gets the length of a string.**

The `length` block counts the characters in a string:

Testing

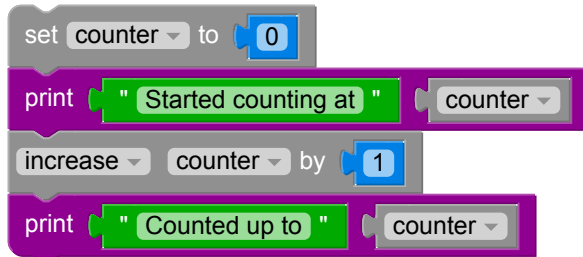
- ☐ Testing the first line of the first example in the question.
- ☐ Testing the whole first example from the question.
- ☐ Testing the first line of the second example from the question.
- ☐ Testing the whole second example from the question.
- ☐ Testing a different set of sentences.
- ☐ Testing lots of long messages.
- ☐ Testing a lot of short messages.
- ☐ Testing when the user enters a blank line straight away.
- ☐ Testing a hidden case.

8.4. Counters

8.4.1. Counting up!

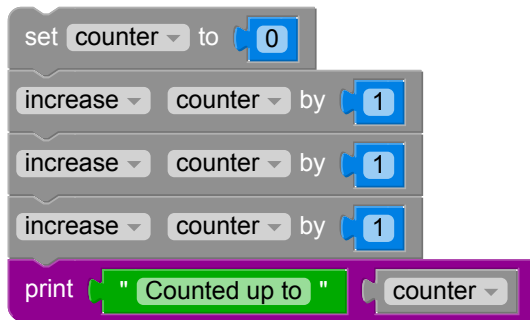
Sometimes, we want to *count* how often something happens in a loop. We need to store the count we're up to in a *counter* variable.

The `increment` `1` block adds 1 more to the counter variable:



Started counting at 0
Counted up to 1

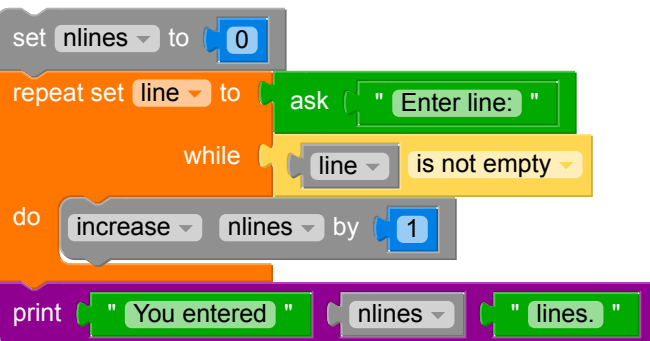
If you use `increment` multiple times, the counter keeps going up:



Counted up to 3

8.4.2. Counters

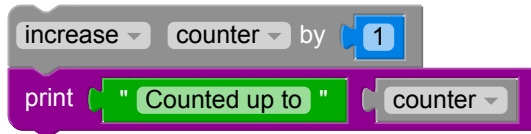
We can use a counter to count the number of lines read:



Any variable can be a counter. Here, we call the counter `nlines` (short for number of lines), set it to zero before the loop, and increment it each time we read a line.

8.4.3. Counter mistakes: part 1

The most common mistake with a counter is to **increment** a variable that doesn't exist yet!

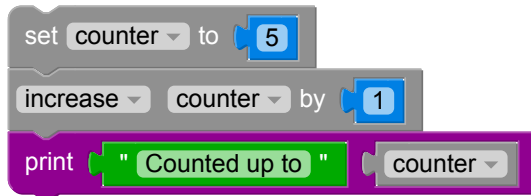


```
Traceback (most recent call last):
  File "program.py", line 1, in <module>
    counter += 1
NameError: name 'counter' is not defined
```

The last part is important: **name 'counter' is not defined**.

We tried to increment our **counter** variable but we haven't said what number to start counting from!

We must **set** the counter variable *before* trying to increment it:

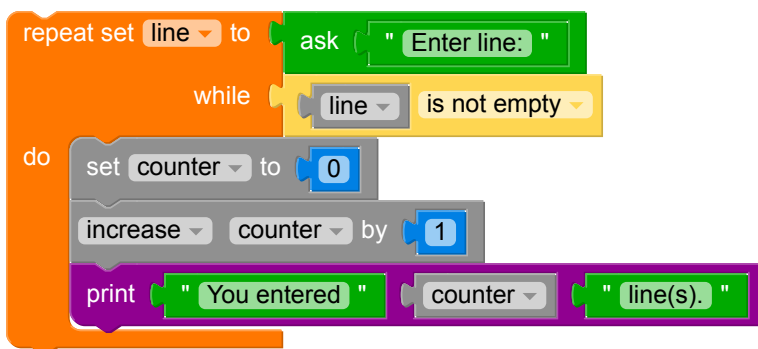


Counted up to 6

You can start counting from whatever number you like!

8.4.4. Counter mistakes: part 2

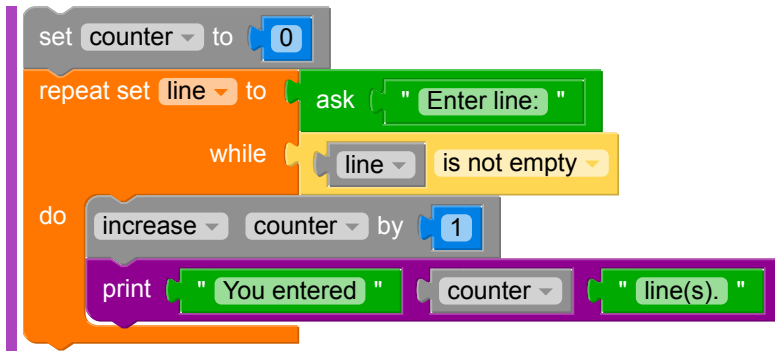
What's wrong with this code? Run it to find out!



If you enter more than one line (before entering a blank line) you'll see that the counter isn't going up! It's always 1!

That's because every time the **while** loop repeats, we're using **set** to set the counter back to zero!

If we start the counter at zero *before* the loop starts, the counter can count up without being reset:



8.4.5. Problem: Michael's Medals



Which athlete has won the most Olympic medals ever? How many guesses do you need to get the answer? **Michael Phelps** (https://en.wikipedia.org/wiki/Michael_Phelps), a swimmer from the USA. He has won 28 medals, including 23 gold!

Let's write a program to see how many attempts the user needs to get the right answer! Here is an example:

```
Who has the most medals? Ian Thorpe
Nope, guess again!
Who has the most medals? Larisa Latynina
Nope, guess again!
Who has the most medals? Dawn Fraser
Nope, guess again!
Who has the most medals? Michael Phelps
Correct!
You made 3 incorrect attempt(s).
```

Your program should keep asking forever until the user enters the exact answer. However, they might get it right the first time:

```
Who has the most medals? Michael Phelps
Correct!
You made 0 incorrect attempt(s).
```

Testing

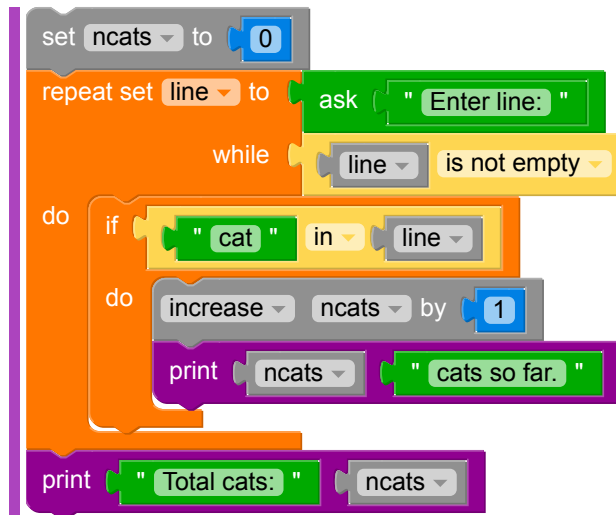
- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing an example with only one incorrect attempt.
- ☐ Testing an example which took 5 incorrect attempts.
- ☐ Testing guesses which include the correct answer.
- ☐ Testing with guesses including blank lines.
- ☐ Testing a very long list of attempts.
- ☐ Testing a hidden case.

8.5. Counting with decisions

8.5.1. Counting some things and not others

Here's a program which counts *only* the lines which contain a **cat**.

The **ncats** (short for number of cats) variable is only incremented if the string **"cat"** is in **line**.



Try it out! And pay attention to when **ncats** increases.

8.5.2. Problem: Raise Your Glass



It's a myth that you should drink at least [8 glasses of water per day](http://www.abc.net.au/science/articles/2012/11/20/3633741.htm) (<http://www.abc.net.au/science/articles/2012/11/20/3633741.htm>). Even though its not true (since there's plenty of water in your food too!) it's still good to drink more water than other drinks.

Let's write a program to track how many glasses of water we drink. Your program should read drinks from the user until they enter a blank line, counting how many glasses of water they've had so far:

```
What did you drink? water
Glasses of water: 1
What did you drink? coke
What did you drink? cordial
What did you drink? water
Glasses of water: 2
What did you drink? water
Glasses of water: 3
What did you drink?
```

Your program should keep asking `What did you drink?` until the user enters a blank line.

Here is another example, where the user drinks lots of water:

```
What did you drink? water
Glasses of water: 1
What did you drink? milk
What did you drink? water
Glasses of water: 2
What did you drink? water
Glasses of water: 3
What did you drink? water
Glasses of water: 4
What did you drink?
```

Testing

- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing an example with only one glass of water.
- ☐ Testing an example with no glasses of water.
- ☐ Testing an example where the user only drinks water.
- ☐ Testing someone who was very thirsty!
- ☐ Testing a hidden case.



PROJECT 3

9.1. More word games

9.1.1. Making simple games

Now that we've learnt how to repeat things, we can make our games much more detailed and fun to play!

Let's revisit the Taboo and Word Chain questions, and turn them into fully-fledged games.

9.1.2. Problem: Do you want to play questions?



[Questions \(https://en.wikipedia.org/wiki/Questions_\(game\)\)](https://en.wikipedia.org/wiki/Questions_(game)) is a game played by maintaining a dialogue of only questions for as long as possible.

Write a program that reads in each line of dialogue and checks that it is a question, printing **Statement!** and ending the game if a line is not a question. In this game, we will assume that everything ending in a question mark ('?') is a question.

Your program should work like this:

```
line: Do you want to play questions?
line: How long does it take?
line: Do you need to go somewhere?
line: No.
Statement!
```

Here is another example:

```
line: What time is it?
line: Don't you have a watch?
line: Is there a clock somewhere?
line: Is there one in that room?
line: I don't think so.
Statement!
```

Testing

- ☐ Testing the first example.
- ☐ Testing the second example.
- ☐ Testing an example with two valid questions.
- ☐ Testing a case with '?' in a statement.
- ☐ Testing a long game.
- ☐ Testing for termination after reading a statement.
- ☐ Testing a game where no questions are asked.
- ☐ Testing a hidden case.

9.1.3. Problem: Taboo Part Two!



[Taboo \(https://en.wikipedia.org/wiki/Taboo_\(game\)\)](https://en.wikipedia.org/wiki/Taboo_(game)) is a word game where one person describes a word on a card so that their partner can guess it. However, there are certain words that the person can't say when trying to describe the thing! A player might have to describe 'cereal' without using the word 'breakfast'.

Write a program that reads in the Taboo word, and continues to read in the description line until the line includes the taboo word. Here's an example for trying to guess the word 'cereal':

```
Taboo word: breakfast
Line: A type of grain
Safe!
Line: you might pour milk on it...
Safe!
Line: Museli is a type of breakfast blank.
Taboo!
```

The program should keep running until the line includes the taboo word (ignoring case):

```
Taboo word: bike
Line: there's a song about them on a bus.
Safe!
Line: where they go round and round?
Safe!
Line: Umm. - Oh! There are 4 of them on a car
Safe!
Line: and one in a unicycle
Safe!
Line: and two on a motorbike
Taboo!
```

Here's a short example when trying to guess the word 'wheel':

```
Taboo word: bike
Line: The things on a BIKE that go flat...
Taboo!
```

Sometimes unrelated words include the sub-word. That still counts as a Taboo! E.g. guessing 'cat':

```
Taboo word: meow
Line: An animal with whiskers
Safe!
Line: As a homeowner they might make your house smell
Taboo!
```

Testing

- ☐ Testing the words in the first example in the question.
- ☐ Testing the first example in the question.
- ☐ Testing the second example in the question.
- ☐ Testing the third example in the question.
- ☐ Testing the fourth example in the question.

- ☐ Testing a game trying to guess 'water'.
- ☐ Testing a similar 'water' game but with different case input.
- ☐ Testing another example with different case input.
- ☐ Testing a hidden test case (to make sure your program works in general).
- ☐ Testing another hidden case.

9.1.4. Problem: Word chain!



[Word Chain](https://en.wikipedia.org/wiki/Word_chain) (https://en.wikipedia.org/wiki/Word_chain) is word game where players take turns saying words that start with the last letter of the previous word. You might have played this game on long car trips.

Write a program to help you play word chain. Your program should read in words until a blank line is entered. It should print out **Invalid word** if a word is not a valid play. Your program should work for upper case and lower case words.

Here is an example:

```
Word: carrot
Word: tomato
Word: orange
Word: mandarin
Invalid word
Word: eggplant
Word: 
```

Notice that the word **mandarin** is rejected because it doesn't start with the letter **e** from the previous word: **orange**. The next word still needs to start with the letter **e** (from **orange**), rather than **n** (from the end of the invalid word, **mandarin**).

Here is another example:

```
Word: tomato
Word: okra
Word: asparagus
Word: seaweed
Word: cake
Invalid word
Word: dried apricots
Word: cake
Invalid word
Word: 
```

Here's one last example. Don't forget it should work regardless of case!

```
Word: Australia
Word: Antartic
Word: Canada
Word: England
Invalid word
Word: Denmark
Invalid word
Word: 
```

You will always read in at least two words.

💡 Hint: previous and current lines

1. Read two lines of input before you start looping.
2. Notice that the previous line becomes the current line in the next iteration.

Testing

- ☐ Testing the first example from the question.
- ☐ Testing the second example from the question.
- ☐ Testing the third example from the question.
- ☐ Testing a perfect word chain.
- ☐ Testing a word chain with multiple mistakes.
- ☐ Testing a case with lots of tricky upper and lower case words.
- ☐ Testing hidden case number 1.
- ☐ Testing hidden case number 2.
- ☐ Testing hidden case number 3.

10

PROJECT 4: PUTTING IT ALL TOGETHER

10.1. More word games

10.1.1. Making simple games

We've now learnt how to make lots of quite complex word games! Nice work!

For our last project, we're going to put everything we've learnt together to make a chatterbot, which is a robot that can reply to your messages in a seemingly intelligent way.

We'll see if we can make our chatbot smart enough that it might just be able to trick a human into thinking it was alive.

The question of whether or not you be able to tell if you were texting your friend or a computer is trickier than you might think!

10.1.2. Problem: Introducing Captain Featherbot



Ya-har. Captain Featherbot is pleased to make your acquaintance!

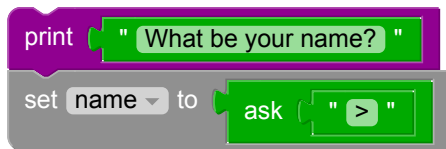
Let's start off with our chatterbot. Captain Featherbot should introduce themselves, ask for the user's name, and then ask them what's on their mind. The Captain should keep letting them talk until the user enters `go away` (or `GO AWAY`).

Here's an example interaction:

```
Arrr, I am Captain Featherbot.
What be your name?
> Sarah
Ahoy Sarah! What be on your mind?
> This is a strange way to talk to a pirate
Arrr. Go on...
> But it's fun!
Arrr. Go on...
> For a little bit.
Arrr. Go on...
> go away
Shiver me timbers!
Farewell Sarah, yer landlubber.
I will be off for more swashbuckling adventures!
```

💡 Input on another line!

Careful! In these questions, you'll need to use `input` like this, printing out a question then using a prompt string afterwards:



We've given you a start to get you going.

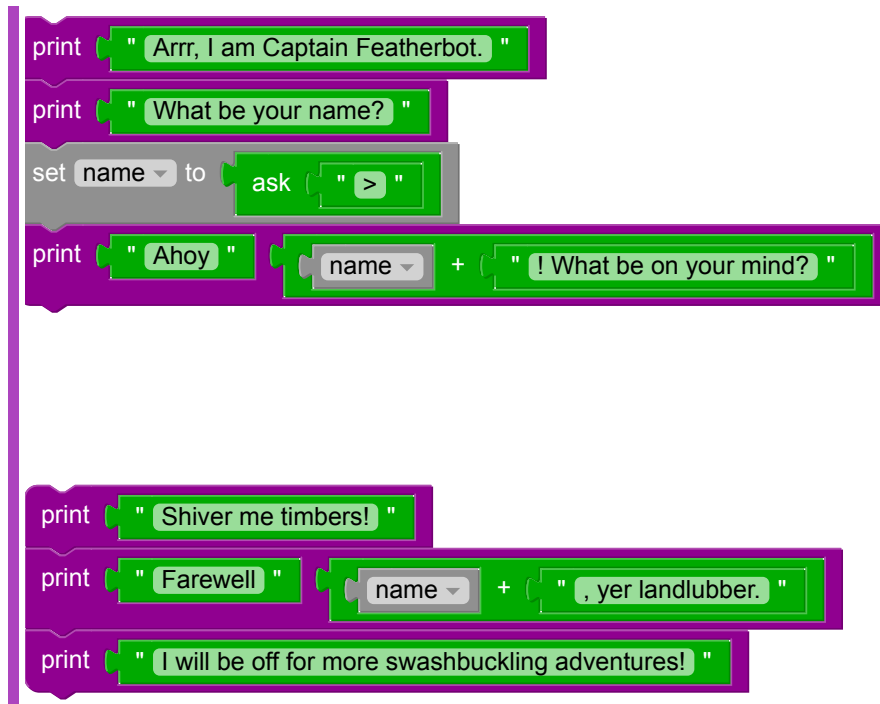
Here's another example:

```
Arrr, I am Captain Featherbot.
What be your name?
> Felix
Ahoy Felix! What be on your mind?
> I want to go sailing.
Arrr. Go on...
> The ocean looks so pretty.
Arrr. Go on...
> But I don't have a boat.
Arrr. Go on...
> I need a boat to go sailing.
Arrr. Go on...
> Do you have a boat?
Arrr. Go on...
> Will you answer my question?
Arrr. Go on...
```

Notice that the user has to type *exactly* **go away** (or **GO AWAY**) without any other characters on the line for the Captain to stop talking.

You'll need

program.blockly



Testing

- ☐ Testing the words in the first example in the question.
- ☐ Testing the capitalisation, punctuation and spaces in the first example in the question.
- ☐ Testing the words in the second example in the question.
- ☐ Testing the capitalisation, punctuation and spaces in the second example in the question.
- ☐ Testing a short example.
- ☐ Testing a longer example.
- ☐ Testing an example with tricky case.
- ☐ Testing a hidden case.

10.1.3. Problem: Captain Featherbot and the Sea



Let's make Captain Featherbot a bit more interesting to talk to.

We've given you some scaffolding for this question. You should also add the logic from the last question!

Captain Featherbot loves talking about their boat and the sea. If you mention the word 'boat' or 'sea' while you're talking to them, they will derail the conversation!

Add to your program, such that if you use the word 'boat' in the line, the Captain says: *Oh, I do love my boat, Floaty McFloatface.* Or, if you use the word 'sea' in the line, the Captain says: *Oh, the sea. Arrr to be back on the sea.*

Here's an example interaction:

```
Arrr, I am Captain Featherbot.
What be your name?
> Guybrush
Ahoy Guybrush! What be on your mind?
> I have a little row boat
Oh, I do love my boat, Floaty McFloatface.
> That's a nice name for a boat.
Oh, I do love my boat, Floaty McFloatface.
> Yes, you said you like it.
Arrr. Go on...
> But I've never been to the seaside.
Oh, the sea. Arrr to be back on the sea.
> I'd like to go!
Arrr. Go on...
> You're no help!
Arrr. Go on...
```

Here's another example. Make sure your program works with upper case and lower case!

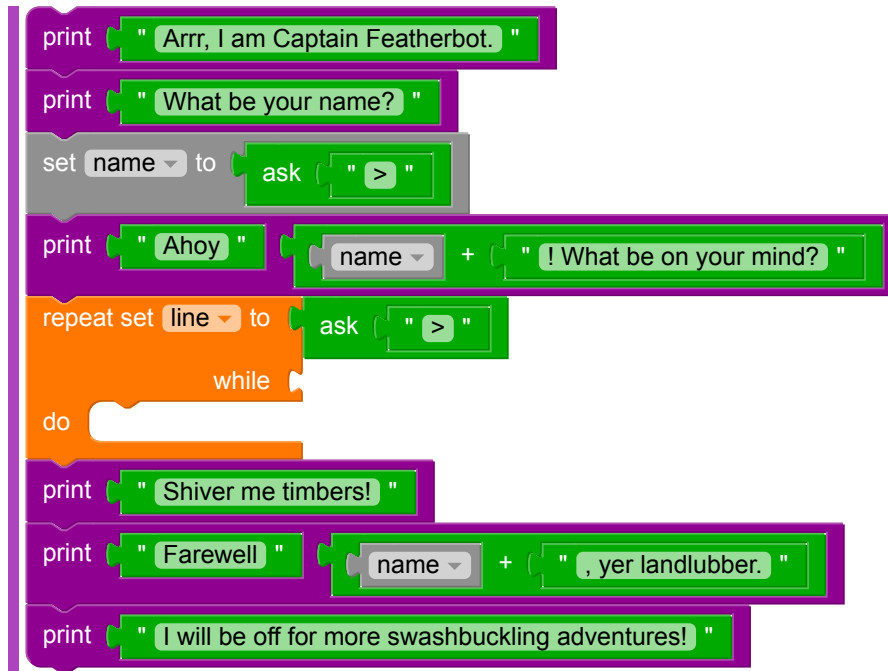
```
Arrr, I am Captain Featherbot.
What be your name? Ahmed
Ahoy Ahmed! What be on your mind?
> BOATS! I like boats.
Oh, I do love my boat, Floaty McFloatface.
> That's a funny name for a boat.
Oh, I do love my boat, Floaty McFloatface.
> Are boats all you ever talk about?
Oh, I do love my boat, Floaty McFloatface.
> STOP TALKING ABOUT YOUR BOAT
Oh, I do love my boat, Floaty McFloatface.
> go away
Shiver me timbers!
Farewell Ahmed, yer landlubber.
I will be off for more swashbuckling adventures!
```

💡 Boats or Seas? Which first?

You won't be given a line that has both 'boat' and 'sea' in it, so don't worry about which way the tests are ordered.

You'll need

 program.blockly



Testing

- ☐ Testing the words in the first example in the question.
- ☐ Testing the capitalisation, punctuation and spaces in the first example in the question.
- ☐ Testing the words in the second example in the question.
- ☐ Testing the capitalisation, punctuation and spaces in the second example in the question.
- ☐ Testing a short example.
- ☐ Testing a longer example.
- ☐ Testing an example with tricky case.
- ☐ Testing a hidden case.

10.1.4. Problem: Captain Featherbot, are you listening?



Now Captain Featherbot is starting to have a bit of personality!

Take your answer from the last question, and then we'll add to it in this question!

For lines that end in a question mark, the Captain says: `That be the real question <name>. I wish I knew.` Make sure to replace `<name>` with the user's name!

If the line ends in an exclamation mark, the Captain says: `Yo ho ho. That be a good one, <name>! Then what?`

If the line starts with `I feel`, the Captain says: `When I feel that way, I go sailing. What do you do?`

Here's an example interaction:

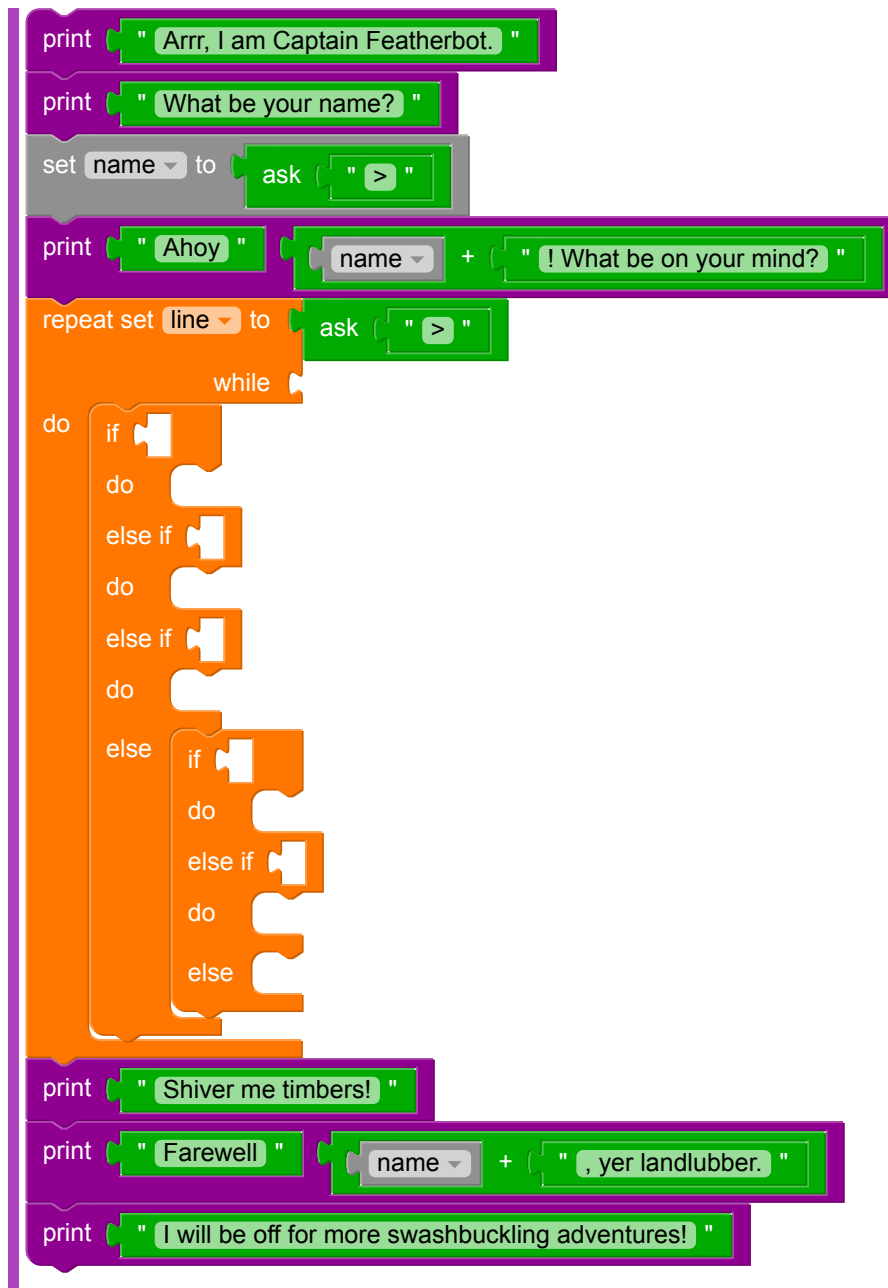
```
Arrr, I am Captain Featherbot.
What be your name?
> Bluebeard
Ahoy Bluebeard! What be on your mind?
> I'm a pirate!
Yo ho ho. That be a good one, Bluebeard! Then what?
> No, that's it.
Arrr. Go on...
> Are you following me?
That be the real question Bluebeard. I wish I knew.
> Don't follow me!
Yo ho ho. That be a good one, Bluebeard! Then what?
> I feel like I'm not being understood.
When I feel that way, I go sailing. What do you do?
> I try to explain it a different way.
Arrr. Go on...
```

Which rule first?

You won't be given a line that matches multiple criteria, so don't worry about which way the rules are ordered.

You'll need

 program.blockly



print

Testing

- ☐ Testing the words in the first example in the question.
- ☐ Testing the capitalisation, punctuation and spaces in the first example in the question.
- ☐ Testing a short example.
- ☐ Testing a longer example.
- ☐ Checking the checks for ? and ! are precise.
- ☐ Testing a hidden case.

10.1.5. Problem: Captain Featherbot!



Take your answer from the last question, and then we'll add to it in this question!

There's one more thing to add to Captain Featherbot's repertoire.

If the user enters a line that starts with **I am <something>**, the Captain should say: **When I was last <something> I stole a boat and sailed the seas.** The Captain will replace the **<something>** with what the user typed in.

This should work if the user enters a line that starts with that pattern regardless of the case of the letters.

Here's an example interaction:

```
Arrr, I am Captain Featherbot.
What be your name?
> Beets McGee
Ahoy Beets McGee! What be on your mind?
> I am hungry
When I was last hungry I stole a boat and sailed the seas.
> i am not amused
When I was last not amused I stole a boat and sailed the seas.
> go away
Shiver me timbers!
Farewell Beets McGee, yer landlubber.
I will be off for more swashbuckling adventures!
```

Here's another example.

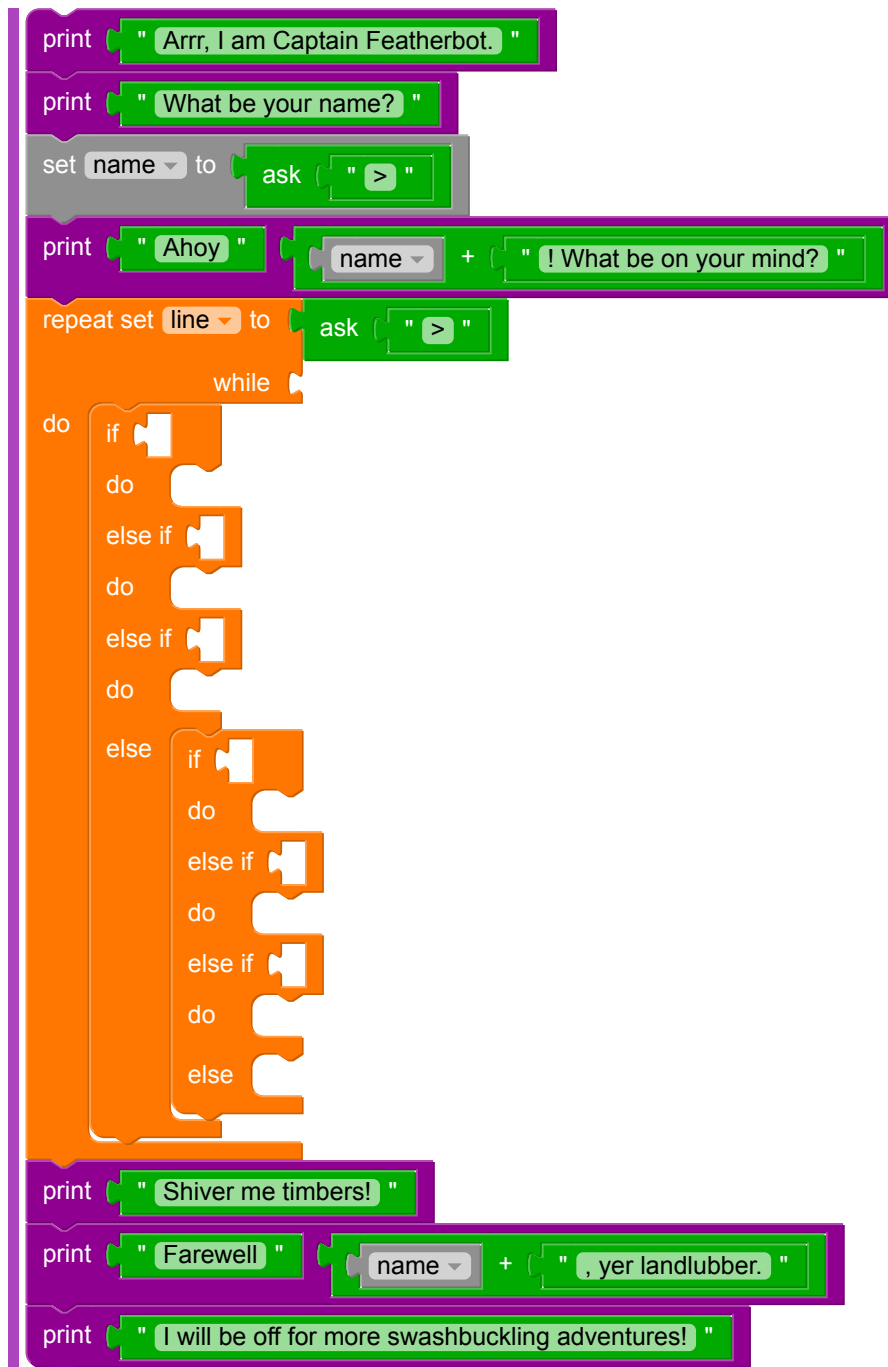
```
Arrr, I am Captain Featherbot.
What be your name?
> Captain Bitbots
Ahoy Captain Bitbots! What be on your mind?
> Bits and bytes
Arrr. Go on...
> I want a boat.
Oh, I do love my boat, Floaty McFloatface.
> Why did you name it that?
That be the real question Captain Bitbots. I wish I knew.
> You don't know?!
Yo ho ho. That be a good one, Captain Bitbots! Then what?
> I feel like you should know.
When I feel that way, I go sailing. What do you do?
> I go look at the sea.
Oh, the sea. Arrr to be back on the sea.
```

💡 Putting it all together!

This question's tricky! There's a lot to put together. Stick with it and you'll be hacking like a pirate in no time!

You'll need

 program.blockly



Testing

- ☐ Testing the words in the first example in the question.
- ☐ Testing the capitalisation, punctuation and spaces in the first example in the question.
- ☐ Testing the words in the second example in the question.
- ☐ Testing the capitalisation, punctuation and spaces in the second example in the question.
- ☐ Testing a short example.
- ☐ Testing a longer example.
- ☐ Testing an example with tricky case.
- ☐ Testing a hidden case.
- ☐ **Yar! Great job, me hearties! Time to celebrate!**

10.1.6. Congratulations, me hearties!

Arrrrr! I be impressed!

Excellent work on finishing the project, and the course! You've learnt how to solve problems with code, and make a cheeky chatbot as well!

We hope you enjoyed this course, and can't wait to see what swashbuckling adventures await you!

10.1.7. Problem: Chatbot Playground!



What's this?! You thought you were finished?

Why not have a go at writing your very own chatbot! You can write whatever code you like in this question. Consider it your personal chatbot playground!

 **Save or submit your code!**

There are no points to be earned for this question, so you can submit whatever code you like. Make sure you save programs that you want to keep!

Testing

☐ This question is a playground question! There's no right or wrong.