Australian **Computing** Academy

DT Challenge Blockly

Year 7 Geometry

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/).

1

DRAWING SHAPES WITH TURTLE

1.1. Turtle

1.1.1. Maths + Computers

Maths and computing are BFFs. Computers were first invented to do maths quickly, but computers wouldn't exist without maths.

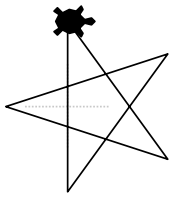
Everything on this screen uses lots of maths, including number systems, algebra, equations, and geometry – all things you'll learn about in Year 7.

Since maths and computing go so well together, you're going to *learn computer programming and maths* at the same time. With these two tools, you can conquer the world!

So let's get started...

1.1.2. Introducing the turtle

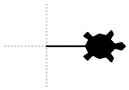
Meet the turtle (https://en.wikipedia.org/wiki/Turtle_graphics)!



In this course, you'll be using Blockly - a visual programming language - to drive the turtle. It's fun, and what you'll learn is the basis for all vector graphics (https://en.wikipedia.org/wiki/Vector_graphics) in computers. It also involves lots of geometry - something you'll be learning about in your maths class.

1.1.3. Make a move!

Let's make the turtle move! Click the ▶ button:



When you run the Blockly code, it makes the turtle move forward!

The number is the number of *turtle steps* to move. A bigger number will move the turtle further!

Try changing 30 to 100, and running it again.

Where are the turtle blocks?

Turtle blocks only work on turtle questions. If you want to try them, skip to the next question, then come back to the notes!

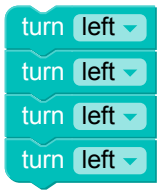
1.1.4. Turning corners

The turtle always starts off facing to the right.

Things would get boring pretty quickly if the turtle always moved in the same direction. If you want to change which way the turtle is facing, you can turn left or turn right. Here's a turn left example:

Now the turtle is facing the top of the screen.

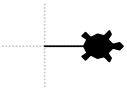
If you turn the same direction four times then the turtle will end up facing the same way it started:



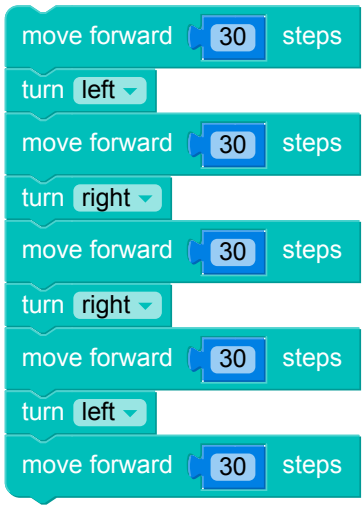
1.1.5. Moving around

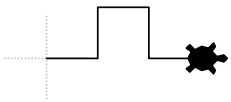
You can combine move forward and turn blocks to draw shapes:

The turtle follows instructions **from its point of view**. If the turtle is facing right, move forward will make the turtle move forward for it (but towards the right of the screen for you):



You must always give move forward the number of *turtle steps* to take. As it moves, it draws a line, so you can draw shapes (and *art*)!

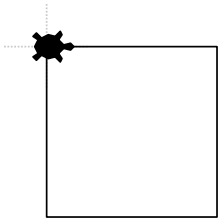




This code draws a line with a bump in the middle. See if you can work out how to change the size of the bump by changing some of the numbers in the example.

1.1.6. Problem: Fair and square

Write a Turtle program to draw a square, with each side being 100 turtle steps long. The output of your program should look like this:



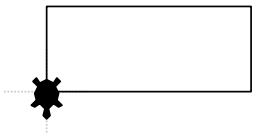
The *top left corner* of the square is where the turtle starts.

Testing

- [ ] Testing the top of the square.
- [ ] Testing the left side of the square.
- [ ] Testing the right side of the square.
- [ ] Testing the bottom of the square.
- [ ] Testing the whole square.
- [ ] Testing for no extra lines.
- [ ] **Fantastic, you've solved your first `turtle` problem!**

1.1.7. Problem: Get rect!

Write a Turtle program to draw a rectangle, with a width (top and bottom sides) of 120 turtle steps, and height (the left and right sides) of 50 turtle steps. The output of your program should look like this:



The *bottom left corner* of the rectangle is where the turtle starts.

Testing

- [ ] Testing the top of the rectangle.
- [ ] Testing the left side of the rectangle.
- [ ] Testing the right side of the rectangle.
- [ ] Testing the bottom of the rectangle.
- [ ] Testing the whole rectangle.
- [ ] Testing for no extra lines.
- [ ] **Fantastic, you've solved this `turtle` problem!**

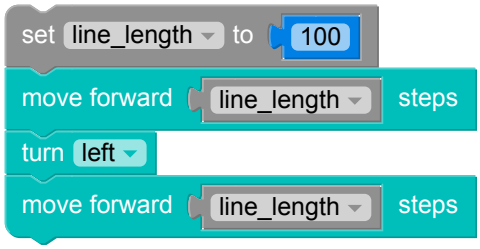
1.2. Variables and input

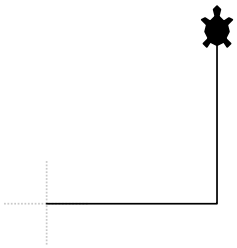
1.2.1. Remembering things

When drawing our square, we repeated the side length 4 times in our code. If we had made a mistake when typing out the length, we wouldn't end up with a square.

Since squares use the same side length each time, it would be great if we could store the side length somewhere and reuse it - it would prevent mistakes, and make our code easier to change.

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





The set line_length block creates a new variable called `line_length`. It holds the value 100. We can then use the line_length block to use that number as often as we want.

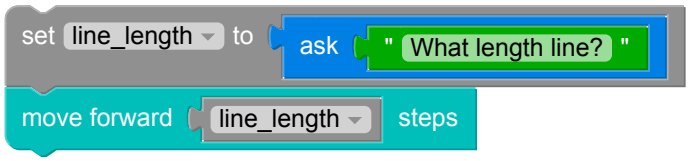
Change the number in the set line_length block to different values, and you'll see that all of the lines in the drawing change to that value, and you only have to make one change to your code!

Creating a variable

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

1.2.2. Asking the user for information

Variables save you from having to type the same values out a lot of times, but they also give us a way to store information that the user might give the program. Let's write a program that lets the user decide how long the line should be:



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

You will need to type a number and press `Enter`:

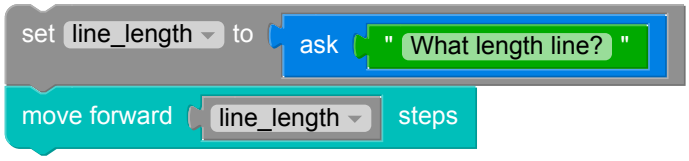
The ask block needs a question string to *ask the user*. It returns the user's answer to our program as a number. Our program stores the answer in the line_length variable so we can use it to draw later. You will find the ask block in the input menu.

Run it again with a different number. You should also try changing the wording of the question.

1.2.3. Different types of data

The computer uses words and numbers in different ways. This means we need a way to differentiate between them, and to declare whether we are using a string like " what length? " or a number like 10. We call these **data types**.

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



We write a question in words, so we need to use a green string block for our question.

If we expect the user to type in a number, then we need to make sure that our ask block is blue, since we'll be using the user's answer as a number.

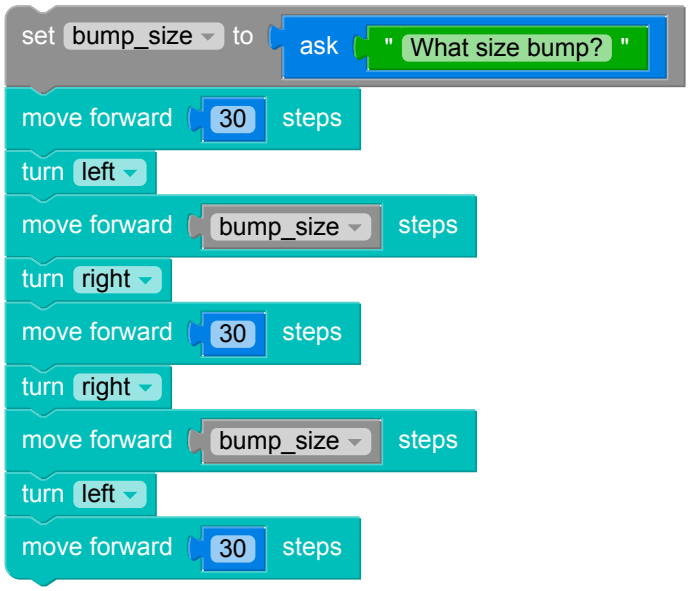
Since moving the turtle around means specifying distances, we need to use the blue blocks in our move forward blocks.

Variable blocks are special - they are grey because they take on the colour of the value last assigned to them - in this case, the blue of the ask block.

1.2.4. Custom shapes

Turtles would be boring if they drew the same shape every time.

Now that we can ask the user for input, let's ask them how high our bump should be:



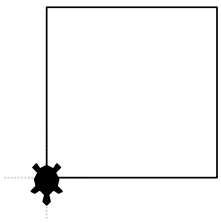
Try running this again and enter different heights!

1.2.5. Problem: Any square

Write a program which draws a square of any size, depending on what the user types in.

The user will type a side length, and your program should use that side length input (in turtle steps) to draw the square.

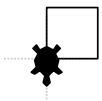
```
Side length: 100
```



The bottom left corner should be in the center of the page, where the turtle starts.

Here's an example where the user has chosen a much smaller side length.

```
Side length: 30
```



You should test your program with a whole range of different sizes! How big can the square be before it goes off the edge of the page?

Hint

Remember that you will find the ask block in the Input menu.

Testing

- [ ] Testing the bottom line from the first example in the question.
- [ ] Testing the right hand side of the square in the first example.

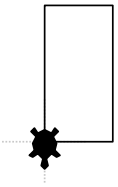
- [ ] Testing the line at the top of the square in the first example.
- [ ] Testing the left hand side of the square in the first example.
- [ ] Testing the whole first example from the question.
- [ ] Testing there are no extra lines in the first example from the question.
- [ ] Testing the second example from the question.
- [ ] Testing there are no extra lines in the second example from the question.
- [ ] Testing a little square.
- [ ] Testing a big square.
- [ ] Testing a hidden test case.
- [ ] Testing another hidden test case.

1.2.6. Problem: Any rectangle

Write a program which draws a rectangle of any height and width, depending on what the user types in.

The user will be asked for a height then a width, and your program should use those input values (in turtle steps) to draw the rectangle. Here's an example of a tall rectangle.

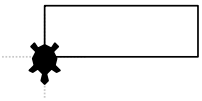
```
Height: 80
Width: 40
```



The bottom left corner should be in the center of the page, where the turtle starts.

Here's an example where the user has chosen a wide rectangle.

```
Height: 30
Width: 90
```



Make sure you test different heights and widths so you are confident your code is correct.

Testing

- [ ] Testing the bottom line from the first example in the question.
- [ ] Testing the right hand side of the rectangle in the first example.
- [ ] Testing the line at the top of the rectangle in the first example.

- [ ] Testing the left hand side of the rectangle in the first example.
- [ ] Testing the whole first example from the question.
- [ ] Testing there are no extra lines in the first example from the question.
- [ ] Testing the second example from the question.
- [ ] Testing there are no extra lines in the second example from the question.
- [ ] Testing another short rectangle.
- [ ] Testing another tall rectangle.
- [ ] Testing a square (equal height and width)
- [ ] Testing a hidden test case.
- [ ] Testing another hidden test case.

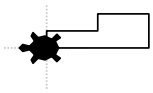
1.2.7. Problem: Step up

Write a program that draws two steps of any height and width, depending on what the user types in.

The user will be asked for a step height then width, and your program should use those input values (in turtle steps) to draw the steps.

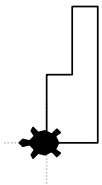
The steps will go up and to the right of the screen, as shown in the example below.

```
Step height: 10
Step width: 30
```



Here's another example.

```
Height: 40
Width: 15
```



Make sure you test different heights and widths so you are confident your code is correct.

> **Hint**
>
> Remember that you need to close the base of the steps, so the turtle will need to move some extra distance at the end of your drawing.

Testing

- [ ] Testing the bottom step from the first example in the question.

- [ ] Testing the whole first example from the question.
- [ ] Testing there are no extra lines in the first example from the question.
- [ ] Testing the second example from the question.
- [ ] Testing there are no extra lines in the second example from the question.
- [ ] Testing some really small steps.
- [ ] Testing some really big steps.
- [ ] Testing a hidden test case.
- [ ] Testing another hidden test case.

2

ANGLES, VARIABLES AND CALCULATIONS

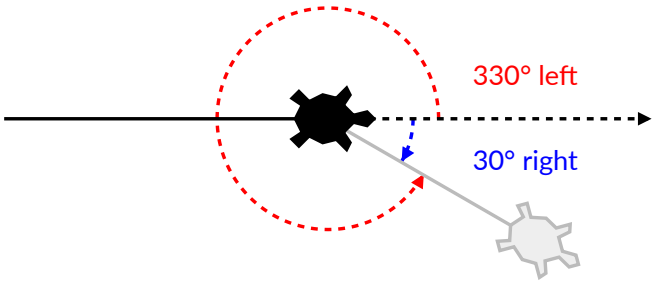
2.1. Calculating angles

2.1.1. Angles and direction

You can think of an *angle* as a *change of direction*. The angle between two lines is the turn you'd make to go from one line to the other. Here, the lines are the turtle's old and new directions.

Angles can be measured in *degrees* (written as °). A 360° turn is a complete circle (a *revolution*). Other turns are fractions of 360°.

Try our interactive diagram! You can drag the grey turtle around.



Turning a quarter of a circle (to face sideways) is 360° ÷ 4 = 90°.
This is called a *right angle* (don't confuse it with turning right!)

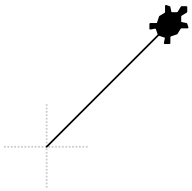
Turning half a circle (to face backwards) is 360° ÷ 2 = 180°.
This is called a *straight angle* (it looks like a straight line).

2.1.2. How much do I turn?

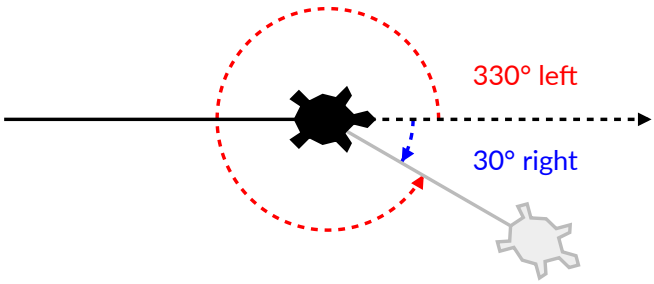
So far we've only turned right angles (90° to the left or right).

If we want to turn more or less, we need to use different angles! Blockly has a turn block that includes the angle to turn.

For example, if the turtle is pointing right, and we want to go to the top right of the screen, we should turn left by 45°:

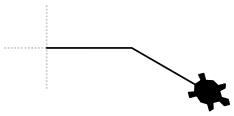


Notice that you can end up in the same direction by turning either left or right. The two angles sum to 360°:



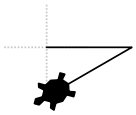
2.1.3. Some turns need angle calculations

When you want the angle *between* two lines to be 30°, you might try turning 30°, then drawing the next line:



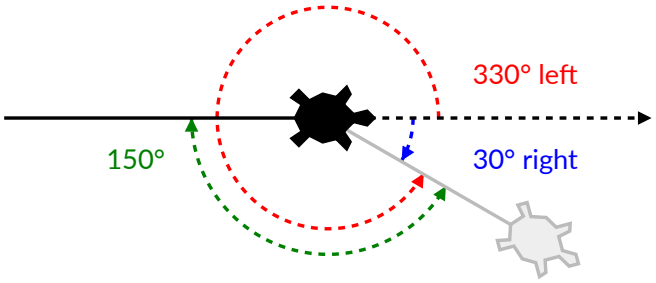
but turning 30° is not far enough! The turtle needs to turn much further to make the angle 30° between the lines.

The correct turn is actually 180° – 30° = 150°. We want to *almost* reverse (turn 180°) but stop when we're 30° short:



2.1.4. Use our turn calculator!

We've added a green *internal* angle between the two lines in our diagram. You can see the 180° – 150° = 30° turn you need:



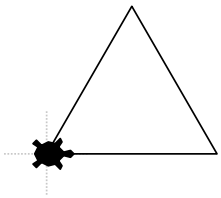
The 30° angle and 150° are called *supplementary angles*, because they sum together to 180°. Knowing this relationship between supplementary angles and the straight angle is important for angle calculations with the turtle.

If you get stuck with angle calculations, use the diagram! It also helps to get some paper and draw a quick diagram by hand.

2.1.5. Problem: Equilateral triangle

Write a Turtle program to draw an equilateral triangle, with the sides being 100 turtle steps long. All angles in an equilateral triangle are 60°.

The output of your program should look like this:



The *bottom left corner* of the triangle is where the turtle starts.

Hint

You need to do an angle calculation to draw this shape!

Testing

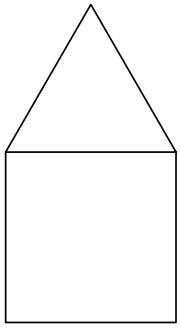
- [ ] Testing the bottom of the triangle.
- [ ] Testing the right side of the triangle.
- [ ] Testing the left side of the triangle.
- [ ] Testing the whole triangle.
- [ ] Testing for no extra lines.
- [ ] **Fantastic, you've drawn an equilateral triangle with `turtle`!**

2.1.6. Problem: Get your house in order

Use the turtle to draw a house!

The triangle at the top should have angles that are all 60° and all sides of the house should be 100 turtle steps long. The sides of the roof will also be 100 steps long.

The result should look like this:



The top left corner of the square is where the turtle starts.

Optional challenge!

Try drawing the house in one line, without drawing over the same line twice. Think about the *order* you need to draw the lines in.

Testing

- [ ] Testing the bottom of the roof (top of the square).
- [ ] Testing the right wall of the house (the right side of the square).
- [ ] Testing the left wall of the house (the left side of the square).
- [ ] Testing the floor of the house (the bottom of the square).
- [ ] **Testing the room of the house** (the square).
- [ ] Testing the left side of the roof.
- [ ] Testing the right side of the roof.
- [ ] **Testing the roof of the house** (the triangle).
- [ ] **Testing the whole house.**
- [ ] Testing for no extra lines.
- [ ] **Well done, your Turtle is now a master builder!**

2.2. Variables in calculations

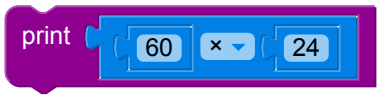
2.2.1. Blockly, the calculator

Blockly can do maths too!

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

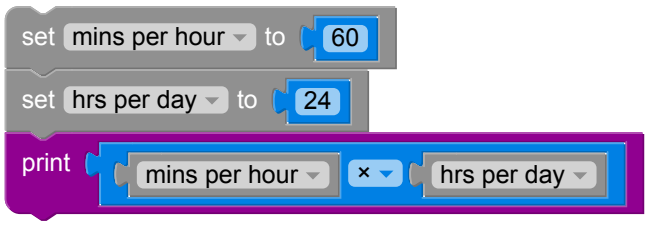
60 minutes per hour × 24 hours per day

In Blockly, this calculation looks like:




```
1440
```

The same calculation can be done with variables:



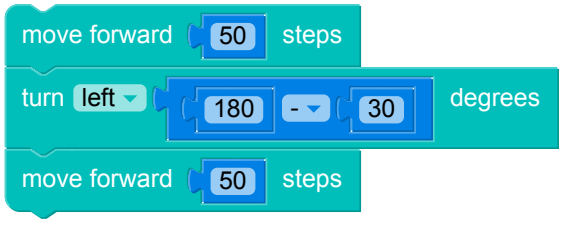

```
1440
```

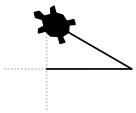
This code is longer, but a lot easier to understand.

2.2.2. Straight angle calculations

Our turn calculator showed us the result of converting an external angle into an internal angle, using our knowledge that a straight angle is 180°.

We can do this in blocks using our new numbers blocks:





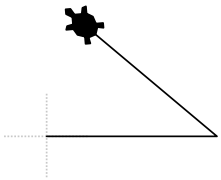
This means we no longer need to do our angle calculations by hand - we can let the computer do them for us!

2.2.3. Problem: Draw any angle

We can use answers to questions for more than just the number of steps, write a program where the turtle asks `What angle?` and then draws an angle of that size. Make the lines 100 turtle steps long.

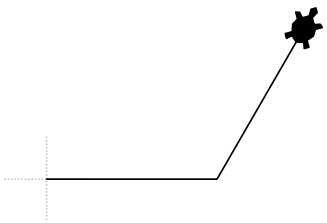
Here's an example of a 40° angle:

```
What angle? 40
```



Here's an example with a bigger angle, 120°:

```
What angle? 120
```



Notice that the angle is being drawn from the *right side* of the base.

Hint: Use the maths blocks

Remember that you need to calculate the correct turn angle from the user's input.

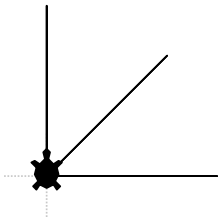
Testing

- [ ] Testing the first example from the question, 40°.
- [ ] Testing the second example from the question, 120°.
- [ ] Testing another acute angle, 5°.

- [ ] Testing another obtuse angle, 146°.
- [ ] Testing a right angle, 90°.
- [ ] Testing a hidden case.

2.2.4. Going backwards

We've been using move forward a lot so far, but did you know the turtle could also move backward?



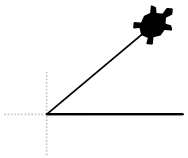
This makes it much easier to draw lines and go back to the start of the line without having to turn around 180°.

2.2.5. Problem: Draw any angle, backwards!

You're going to write a program similar to the last one where the turtle asks `What angle?` and then draws an angle of that size. This time, make the lines 80 turtle steps long, but draw them from the left!

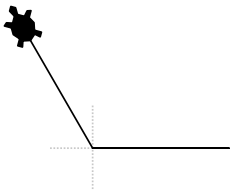
Here's an example of a 40° angle:

```
What angle? 40
```



Here's an example with a bigger angle, 120°:

```
What angle? 120
```



Hint: Use the backward block

This question gets a lot easier when you use the backward block!

Testing

- [ ] Testing the first example from the question, 40°.
- [ ] Testing the second example from the question, 120°.
- [ ] Testing another acute angle, 5°.
- [ ] Testing another obtuse angle, 146°.

- [ ] Testing a right angle, 90°.
- [ ] Testing a hidden case.

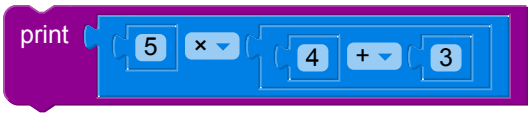
2.2.6. Doing maths

Blockly can do all the maths a calculator can do:

Sign	Name
+	add
–	subtract
×	multiply
÷	divide

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

Blockly calculates the inner maths blocks first, so:




```
35
```

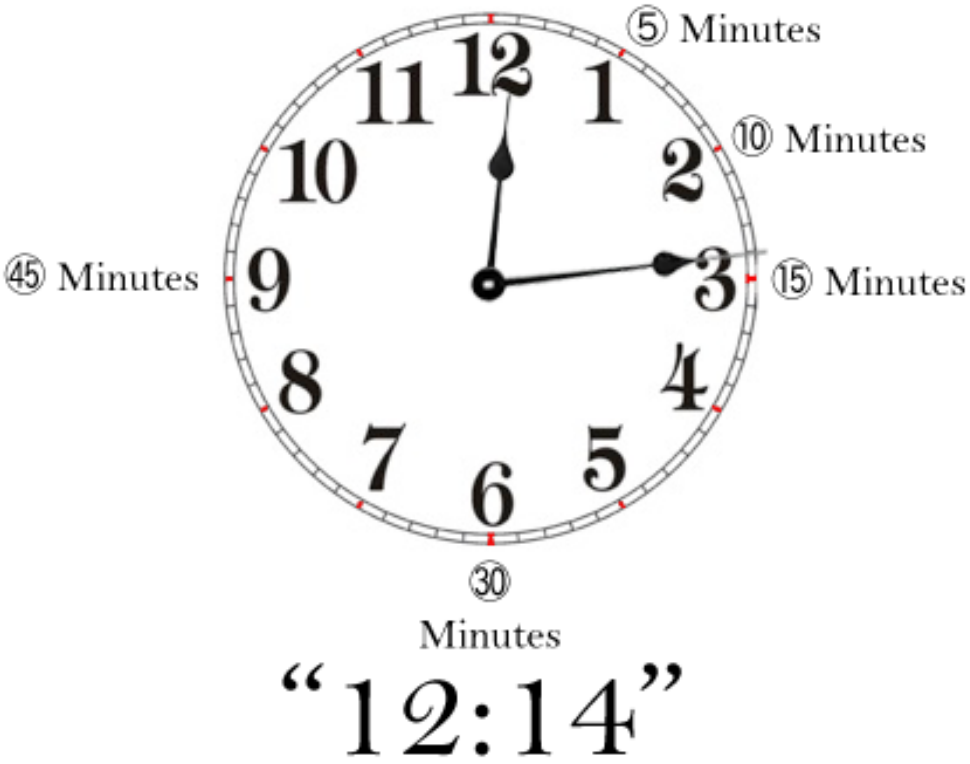
calculates the `4 + 3` first (to give `7`) and then does `5 × 7`.

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

2.2.7. Problem: What's the time?

The numbers on a clock face are evenly spaced, with the number 12 at the top. Since there are twelve numbers on a clock and 360° in a circle, the angle between each number on the clock face is 360° ÷ 12 = 30°. This means the number 1 is 30° to the right of the 12, 2 is 60° right from the 12, and so on.

Write a program where the turtle asks `What time is it?` and then draws that time. The user will only ever type in a single number from 1 to 12, and the time is always that number "o'clock". The minute hand should be 80 turtle steps long, and the hour hand 30 turtle steps long.



This clock shows that the time is 12:14

Here's an example of a 1 o'clock:

```
What time is it? 1
```

Here's an example at 7 o'clock:

```
What time is it? 7
```

The turtle starts *in the centre* of the clock, so you need to work out your angles from there.

> **Hint**
>
> There is a relationship between the hour and the angle that you can use to write a simple equation for your turn angle.

Testing

- [ ] Testing the first example from the question, 1 o'clock.
- [ ] Testing the second example from the question, 7 o'clock.
- [ ] Testing 2 o'clock.
- [ ] Testing 3 o'clock.
- [ ] Testing 4 o'clock.
- [ ] Testing 5 o'clock.
- [ ] Testing 6 o'clock.
- [ ] Testing 8 o'clock.
- [ ] Testing 9 o'clock.
- [ ] Testing 10 o'clock.
- [ ] Testing 11 o'clock.
- [ ] Testing 12 o'clock.
- [ ] **Excellent, Your clock works for every time!**

2.2.8. Problem: Scaleable House

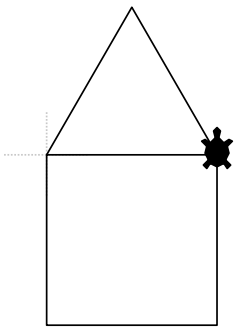
Let's draw a house for ants, or for giants!

You might have already learned about enlargement transformations - where we can take any shape and then scale it up by multiplying all of the sides by the same number. We'll use this transformation to draw a house of any size!

Write a program which asks the user what size the house floor is (in turtle steps), the scale they want the house to be transformed by, and then uses the turtle to draw the house.

The triangle at the top should have angles that are all 60° and all sides of the house should be the same number of turtle steps. Here's an example of a 50 turtle step house, scaled by 2 (so the sides all end up 50 × 2 = 100 steps long).

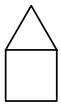
```
Size: 50
Scale: 2
```



The top left corner of the square is where the turtle starts.

Here's another example of a much smaller house:

```
Size: 10
Scale: 3
```



Copy Blocks

> Did you know you can copy blocks you've already made? Just right-click on the block in your answer and choose Duplicate!

Testing

- [ ] Testing the bottom of the roof (top of the square) in the first example from the question.
- [ ] Testing the right wall of the house in the first example from the question.
- [ ] Testing the left wall of the house in the first example from the question.
- [ ] Testing the floor of the house in the first example from the question.
- [ ] **Testing the room of the house** (the square) in the first example.
- [ ] **Testing the roof of the house** (the triangle) in the first example.
- [ ] **Testing the whole house** in the first example.
- [ ] Testing for no extra lines in the first example.
- [ ] Testing the second example house from the question (with 10 × 3 turtle steps).
- [ ] Testing a house where the size is 60 × 1 turtle steps.
- [ ] Testing a very very tiny house (3 × 5 turtle steps).
- [ ] Testing a hidden test case.
- [ ] Testing another hidden test case.

3

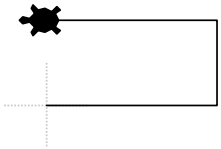
PARALLEL LINES AND CONTROLLING THE PEN

3.1. Parallel lines: using the pen

3.1.1. Parallel lines

Two lines are **parallel** if they're always the same distance apart. We can draw parallel lines by moving the turtle any distance 90° from our first line, then drawing another line in the same direction as the first one (by turning another 90°).

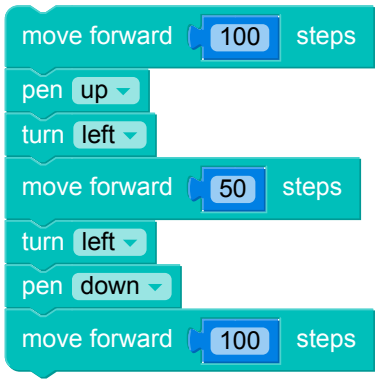
In the drawing below, the two horizontal lines at the top and bottom are parallel to one another.

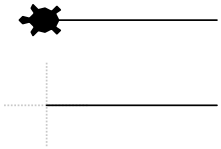


When we draw parallel lines, we don't want our construction line (the vertical one) to be there. We can move the turtle without drawing a line by lifting our pen up, and putting it back down again.

3.1.2. Pen up and down

To move the turtle around into the right position without drawing anything, use the pen up block like this:





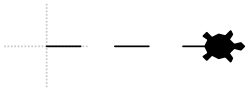
Imagine the turtle is holding a pen, after you've lifted up the pen with pen up the pen is off the paper and the turtle won't draw anything as it moves around. After you've put the pen back down with pen down, the turtle will draw as it moves again.

3.1.3. Problem: Dashed line

We'll put our understanding of pen up and down to the test by drawing a dashed line.

Write a program that asks the user `How long is the line?` (in turtle steps), then draws a dashed line made up of five parts - 3 with the pen down, and 2 with the pen up. Here's an example of a 100 step line.

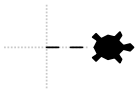
```
How long is the line? 100
```



The turtle will face and move to the right.

Here's another example:

```
How long is the line? 35
```



Testing

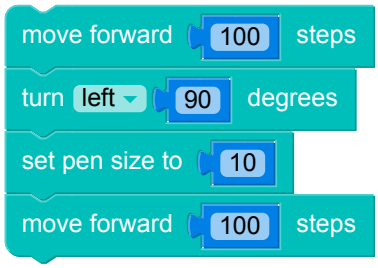
- [ ] Testing the first dash in the line in the first example.
- [ ] Testing dash-blank-dash from the line in the first example.
- [ ] Testing the full dashed line in the first example.
- [ ] Testing for no extra lines in the first example.
- [ ] Testing the second example from the question (35 turtle steps).

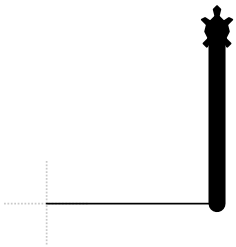
- [ ] Testing a line where the length is 60 turtle steps.
- [ ] Testing a short line (15 turtle steps).
- [ ] Testing a hidden test case.
- [ ] Testing another hidden test case.

3.1.4. Drawing thicker lines

The set pen size block sets the width of the pen that the turtle uses to draw. The width must always be a whole number.

The default pen width we have used so far is 1.





Try it out with different sizes!

3.1.5. Problem: It's growing!

We can change the thickness of the pen to create interesting shapes and drawings. Making the pen bigger as we move in the same direction can create simple tree-like shapes!

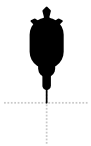
Write a program that asks the user `How tall is it?` (in turtle steps), then draws a line straight up with four even-length segments (the height divided by 4). The bottom segment should be drawn with a pen size of 1, the next segment a pen size of 5, the third segment a pen size of 10, and the top segment a pen size of 20.

```
How tall is it? 80
```

The turtle will face and move up. It starts at the bottom of the tree.

Here's another example:

```
How tall is it? 40
```



Testing

- [ ] Testing the base of the tree in the first example.
- [ ] Testing the bottom half of the tree in the first example.
- [ ] Testing the top half of the tree in the first example.

- [ ] Testing the tree in the first example.
- [ ] Testing for no extra lines in the first example.
- [ ] Testing the second example from the question (40 turtle steps).
- [ ] Testing a tree with height 60 turtle steps.
- [ ] Testing a short tree (12 turtle steps).
- [ ] Testing a hidden test case.
- [ ] Testing another hidden test case.

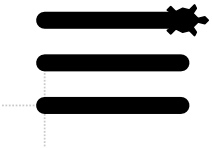
3.1.6. Problem: The hamburger icon

All over the web you would have seen the hamburger button (https://en.wikipedia.org/wiki/Hamburger_button) on menus and graphical user interfaces. You'd be forgiven for thinking it is a relatively new invention given its rise in popularity, but it's actually been around since 1982 (https://blog.placeit.net/history-of-the-hamburger-icon/).

The hamburger button is commonplace in modern GUIs. This version can be found on Iconfinder and was made by Timothy Miller (https://www.iconfinder.com/icons/134216/hamburger_lines_menu_icon).

Write a program that asks the user for a width, gap and thickness (in that order), then draws a version of the hamburger button using those specifications. The width will be the length of each of the lines in turtle steps, the gap is the height between each line in turtle steps, and the thickness will be the thickness of the pen.

```
Width: 80
Gap: 25
Thickness: 10
```



The turtle will start in the bottom left corner of the icon.

Here's another example:

```
Width: 40
Gap: 15
Thickness: 6
```

Testing

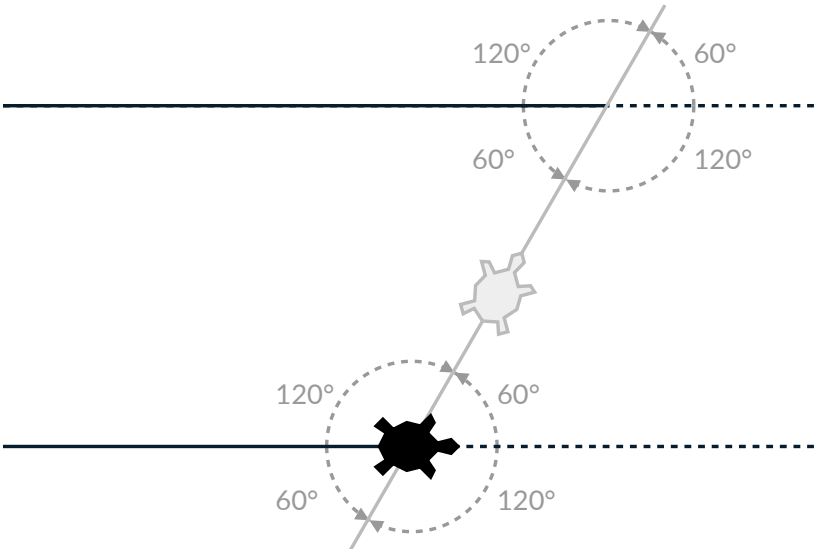
- [ ] Testing the bottom of the hamburger in the first example.
- [ ] Testing the middle of the hamburger in the first example.
- [ ] Testing the top of the hamburger in the first example.
- [ ] Testing the whole hamburger in the first example.
- [ ] Testing for no extra lines in the first example.
- [ ] Testing the second example from the question.
- [ ] Testing a hamburger with width 40, gap 10, thickness 4.
- [ ] Testing a small hamburger (15, 6, 2).
- [ ] Testing a hidden test case.
- [ ] Testing another hidden test case.

3.2. Angles on parallel lines

3.2.1. Transversals

When another line crosses parallel lines, it's called a **traversal**.

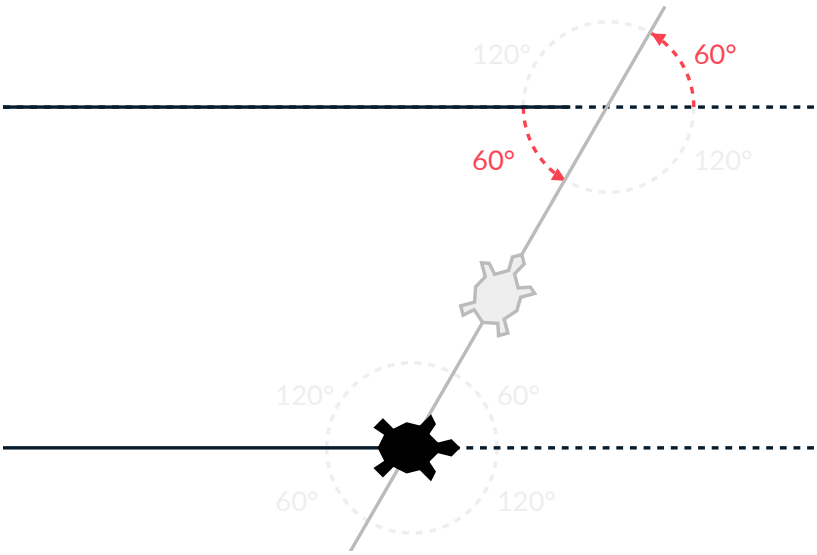
The black lines below are parallel, with a grey *traversal* line crossing them. **You can drag the grey turtle to change the traversal angle:**



As you move the traversal line around, you'll see that different pairs of angles are related in different ways. Let's explore some of these angle relationships...

3.2.2. Opposite angles

When any two lines intersect, the angles that are opposite to each other are equal. There are four pairs of *opposite angles*, and we've highlighted one pair in red:



Try to spot all four pairs, and then click these to check your answer:

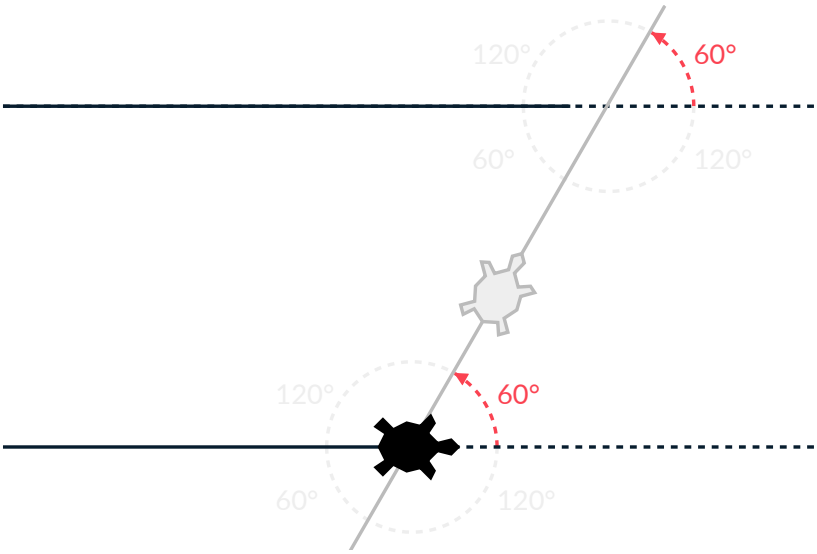
Pair 1 | Pair 2 | Pair 3 | Pair 4 | Clear

Vertically opposite

Opposite angles are also called *vertically opposite angles*. They are called *vertical* because they share a common *vertex*, not because they have to be one on top of the other.

3.2.3. Corresponding angles

Corresponding angles on parallel lines are equal. *Corresponding* means they appear in the same position on each parallel line. For example, the two red angles below are corresponding:

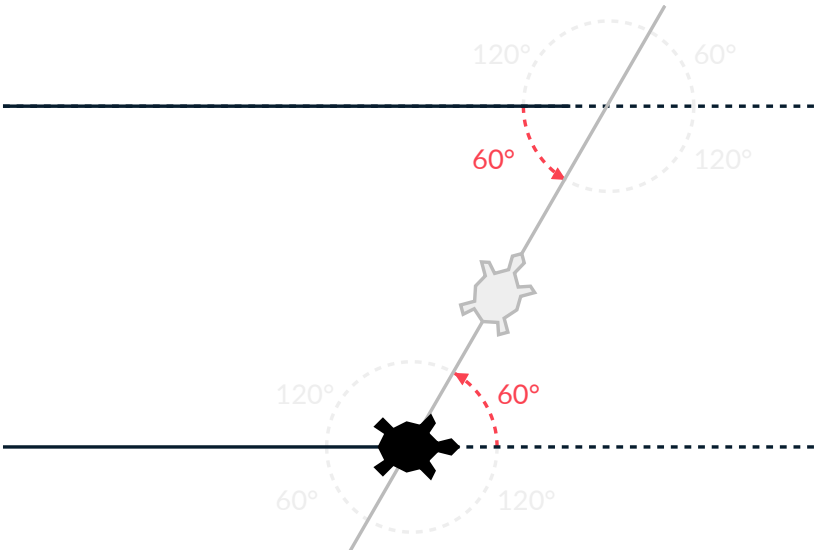


Try to spot all four pairs, and then click these to check your answer:

Pair 1 Pair 2 Pair 3 Pair 4 Clear

3.2.4. Alternate angles

Alternate angles on parallel lines are also equal. There are only two pairs of alternate angles, and we've highlighted one pair in red:

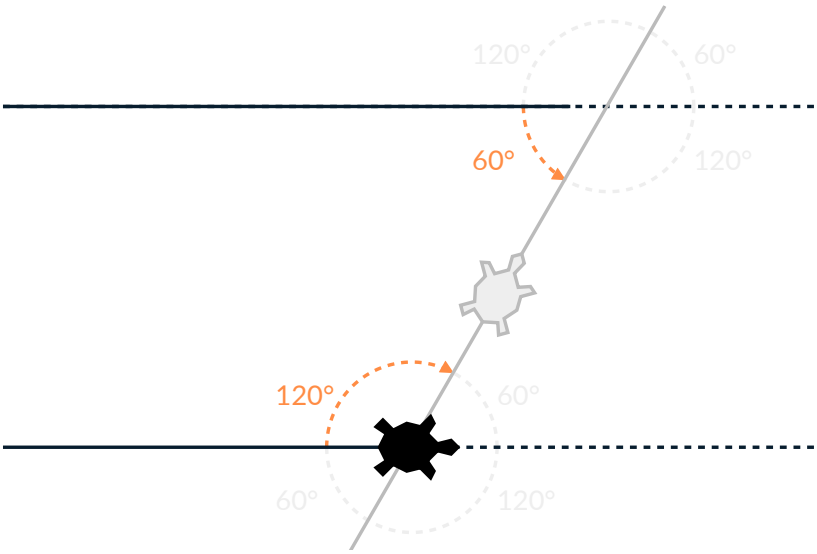


Try to spot the other pair, and then click these to check your answer:

Pair 1 Pair 2 Clear

3.2.5. Co-interior angles

Co-interior angles are different to all the rest: a pair of co-interior angles will sum to 180° (this is called *supplementary*). We've highlighted one of the two pairs of co-interior angles in orange:



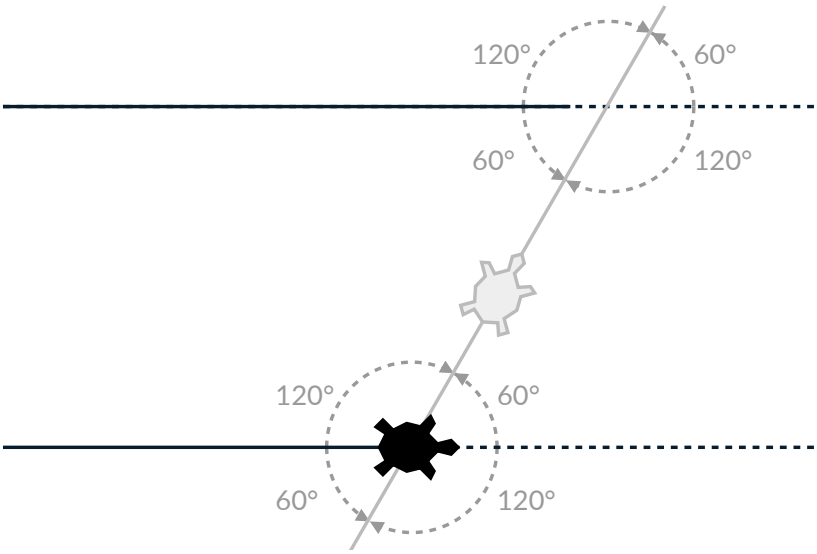
Try to spot the other pair, and then click these to check your answer:

Pair 1 Pair 2 Clear

> **Complementary or supplementary?**
>
> If you find yourself getting the two confused, remember that complementary angles sum to 90° and supplementary angles sum to 180° because 90 is smaller than 180, and C comes before S in the alphabet!

3.2.6. Angle relationships summary

Knowing which angles crossing parallel lines and within shapes are related makes it much easier to draw them! Use the diagram and table below to help you if you get stuck:



Relationship	Property	Examples (click number to show)
Opposite	are the same	Pair 1 Pair 2 Pair 3 Pair 4

Relationship	Property	Examples (click number to show)
Corresponding	are the same	Pair 1 Pair 2 Pair 3 Pair 4
Alternate	are the same	Pair 1 Pair 2
Cointerior	sum to 180°	Pair 1 Pair 2

Clear diagram

3.2.7. Problem: Letter Z

Write a Turtle program to draw the letter Z. The top and bottom lines should be 50 turtle steps long and **parallel to each other**. The diagonal line should be 100 turtle steps long, and the pen size shoudl be set to 4.

In order to draw the Z, you'll need to calculate the size of the turns you need to make if **the angle between the top and diagonal line is 60˚**. The turtle should start drawing from the top left corner:

Hint

Try drawing the shape out on a piece of paper and calculating the angles you need before you start coding!

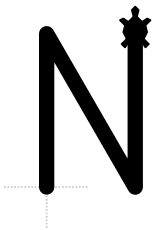
Testing

- [ ] Testing the top line of the Z.
- [ ] Testing the diagonal line of the Z.
- [ ] Testing the bottom line of the Z.
- [ ] Testing the whole letter Z.
- [ ] Testing for no extra lines.

3.2.8. Problem: Letter N

Write a Turtle program to draw the letter N. The left and right lines should be 90 turtle steps long and **parallel to each other**. The diagonal line should be 104 turtle steps long, and the pen size should be 9.

In order to draw the N, you'll need to calculate the size of the turns you need to make if **the angle between the right and diagonal line is 30˚**. The turtle should start drawing from the bottom left corner:



Hint

Try drawing the shape out on a piece of paper and calculating the angles you need before you start coding!

Testing

- [ ] Testing the left line of the N.
- [ ] Testing the diagonal line of the N.
- [ ] Testing the right line of the N.
- [ ] Testing the whole letter N.
- [ ] Testing for no extra lines.

3.2.9. Problem: Parallelogram

A *parallelogram* is a shape where opposite sides are equal (like a rectangle) but it's tilted to one side. Opposite sides of a parallelogram are also parallel to each other!

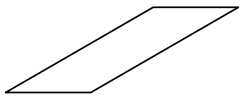
Write a program which asks for an angle, then draws a parallelogram using that angle for the bottom left (and top right!) corners. The top and bottom sides should be 50 steps long, and the left and right sides should be 100 steps long.

The bottom left corner should be where the turtle starts.

When the angle entered is acute (less than 90°), the bottom left corner is acute and the parallelogram leans to the right. For example:

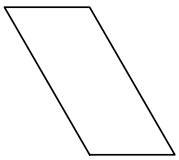
```
Enter the angle: 30
```

produces a parallelogram like this:



If the entered angle is obtuse (greater than 90°), the bottom left angle is obtuse and the parallelogram leans to the left:

```
Enter the angle: 120
```



You'll need

program.py

Testing

- [ ] Testing the left side of the 30° parallelogram.
- [ ] Testing the top of the 30° parallelogram.
- [ ] Testing the right side of the 30° parallelogram.
- [ ] Testing the bottom of the 30° parallelogram.
- [ ] **Testing the whole 30° parallelogram.**
- [ ] Testing for no extra lines in the 30° parallelogram.
- [ ] Testing the left side of the 120° parallelogram.
- [ ] Testing the top of the 120° parallelogram.
- [ ] Testing the right side of the 120° parallelogram.
- [ ] Testing the bottom of the 120° parallelogram.
- [ ] **Testing the whole 120° parallelogram.**
- [ ] **Testing a 40° parallelogram.**
- [ ] Testing a special kind of parallelogram (a rectangle!).
- [ ] Testing a very flat parallelogram (170°).
- [ ] Testing a hidden case.

4

PROJECT: VECTOR GRAPHICS

4.1. Vector graphics

4.1.1. Vector graphics

The turtle *represents* a drawing mathematically as lines and angles. So do vector graphics () tools, e.g. Adobe Illustrator (http://www.adobe.com/uk/products/illustrator.html), and file formats, e.g. Portable Document Format (https://en.wikipedia.org/wiki/Portable_Document_Format) (PDF).

A *vector* in vector graphics is just a line. These lines can be moved (e.g. scaled or rotated), so vector graphics can be drawn at any size or angle, and be perfectly crisp and smooth.

Fonts are also represented as vectors, allowing a browser to draw:

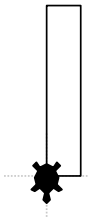
using a single mathematical description of how to draw an **H**.

The 3D graphics and models used in computer games, computer generated imagery (CGI) in movies, and computer aided design (CAD) are also represented using vectors.

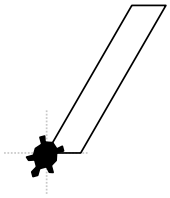
> **Interested in writing games?**
>
> Then learn as much geometry (and mathematics) as you can!

4.1.2. Letters as shapes

You've already drawn some vector graphics! Capital I is just a long skinny rectangle:



And the italic version *I* is just a parallelogram:



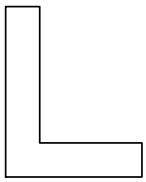
4.1.3. Problem: Straight L

Let's try a slightly trickier letter!

Write a program that draws a straight letter L (all right angles), so that the bottom left corner is in the center of the space.

- The thickness of each of the bars of the L should be 20 steps;
- The height of the L should be 100 steps;
- The width of the L should be 80 steps.

It should look like this:



Testing

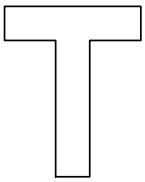
- [ ] Testing the left vertical side of a straight L.
- [ ] Testing the top horizontal side of a straight L.
- [ ] Testing the inner vertical side of a straight L.
- [ ] Testing the inner horizontal side of a straight L.
- [ ] Testing the right side of a straight L.
- [ ] Testing the bottom of a straight L.
- [ ] **Testing the whole straight L shape.**
- [ ] Testing for no extra lines in the straight L.

4.1.4. Problem: Straight T

Time to take a up a notch with the letter T!

Write a program that draws a straight letter T (all right angles), so that the bottom left corner is in the center of the space.

- The thickness of each of the bars of the T should be 20 steps;
- The height of the T should be 100 steps;
- The width of the T should be 80 steps.



Testing

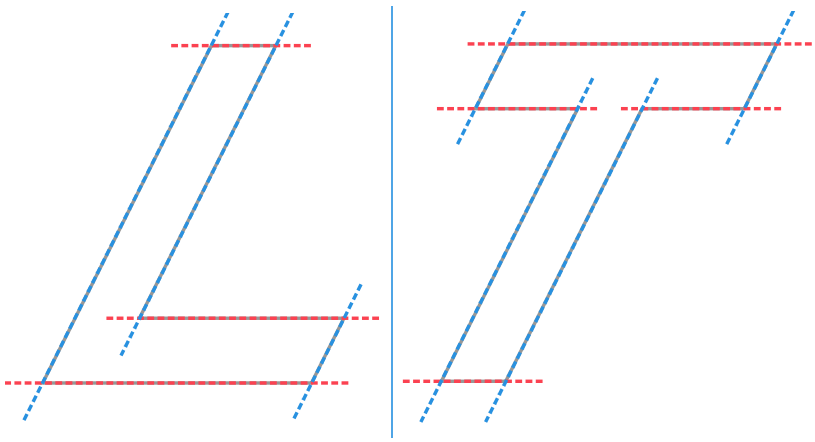
- [ ] Testing the stem's left side on a straight T.
- [ ] Testing the left arm's underside on a straight T.
- [ ] Testing the left arm's side on a straight T.
- [ ] Testing the bottom of a straight T.
- [ ] Testing the stem's right side on a straight T.
- [ ] Testing the right arm's underside on a straight T.
- [ ] Testing the right arm's side on a straight T.
- [ ] Testing the top of a straight T.
- [ ] **Testing the whole straight T shape.**
- [ ] Testing for no extra lines in the straight T.

4.2. Italics

4.2.1. Real world examples

The rest of the questions in this course are a bit tricky. We're going to apply parallel line angle relationships to real world examples in vector graphics!

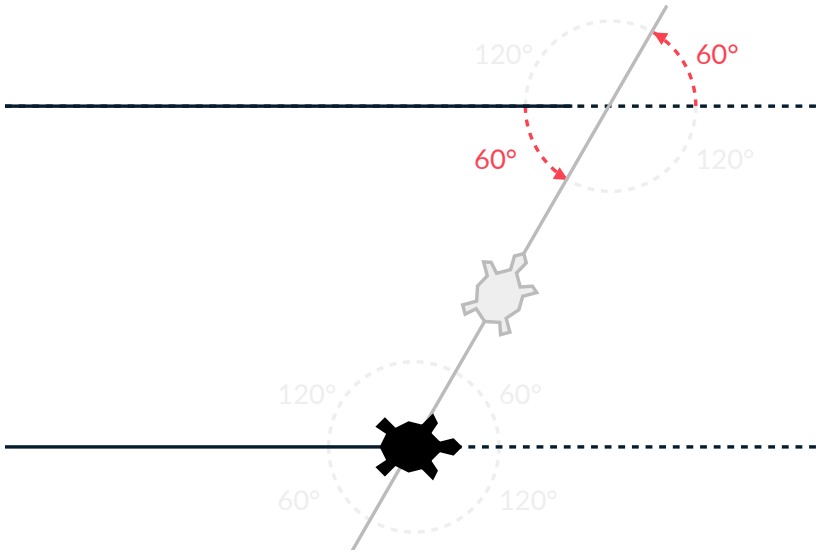
You'll be drawing italics versions of the letters L and T. There are several sets of parallel lines in the letters L and T:



But first, let's revise the angle relationships...

4.2.2. Opposite angles

When any two lines intersect, the angles that are opposite to each other are equal. There are four pairs of *opposite angles*, and we've highlighted one pair in red:



Try to spot all four pairs, and then click these to check your answer:

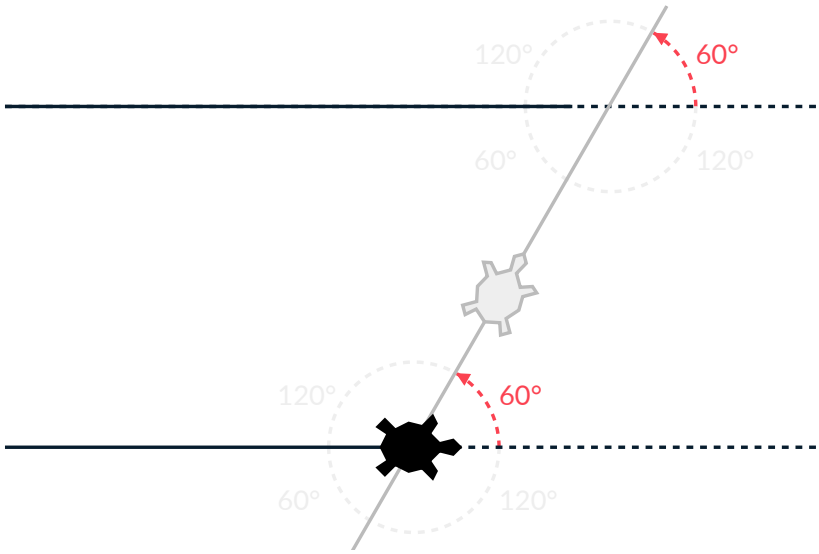
Pair 1 Pair 2 Pair 3 Pair 4 Clear

Vertically opposite

Opposite angles are also called *vertically opposite angles*. They are called *vertical* because they share a common *vertex*, not because they have to be one on top of the other.

4.2.3. Corresponding angles

Corresponding angles on parallel lines are equal. *Corresponding* means they appear in the same position on each parallel line. For example, the two red angles below are corresponding:

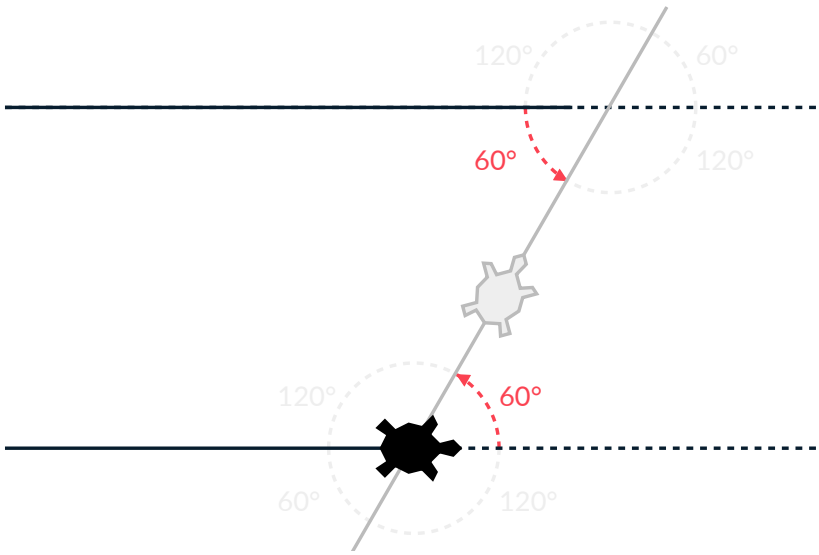


Try to spot all four pairs, and then click these to check your answer:

Pair 1 Pair 2 Pair 3 Pair 4 Clear

4.2.4. Alternate angles

Alternate angles on parallel lines are also equal. There are only two pairs of alternate angles, and we've highlighted one pair in red:

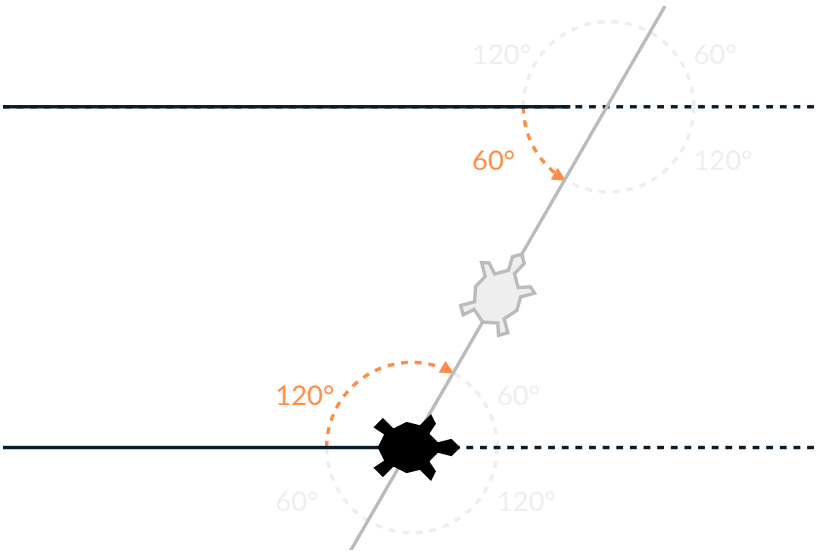


Try to spot the other pair, and then click these to check your answer:

Pair 1 Pair 2 Clear

4.2.5. Co-interior angles

Co-interior angles are different to all the rest: a pair of co-interior angles will sum to 180° (this is called *supplementary*). We've highlighted one of the two pairs of co-interior angles in orange:



Try to spot the other pair, and then click these to check your answer:

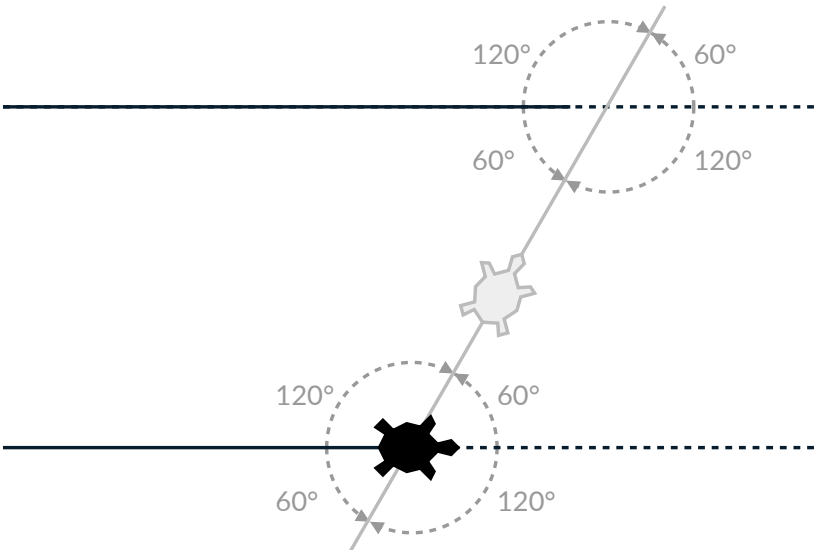
Pair 1 Pair 2 Clear

Complementary or supplementary?

If you find yourself getting the two confused, remember that complementary angles sum to 90° and supplementary angles sum to 180° because 90 is smaller than 180, and C comes before S in the alphabet!

4.2.6. Angle relationships summary

Knowing which angles crossing parallel lines and within shapes are related makes it much easier to draw them! Use the diagram and table below to help you if you get stuck:



Relationship	Property	Examples (click number to show)
Opposite	are the same	Pair 1 Pair 2 Pair 3 Pair 4
Corresponding	are the same	Pair 1 Pair 2 Pair 3 Pair 4

Relationship	Property	Examples (click number to show)
Alternate	are the same	Pair 1 Pair 2
Cointerior	sum to 180°	Pair 1 Pair 2

Clear diagram

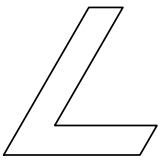
4.2.7. Problem: Italic L

Using *italics* is important for adding emphasis and changing the voice in which a word or phrase is spoken.

Write a program that draws an italic letter L, so the angle in the bottom left corner is 60°.

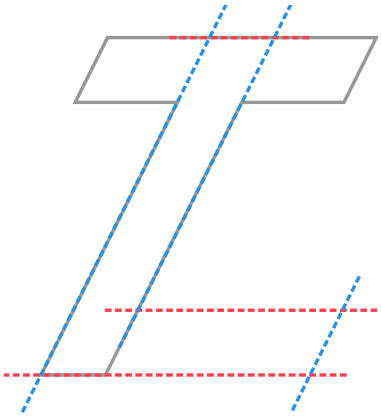
Only the angles of the turns should change from the previous straight L question, the lines should all be the same length:

- The thickness of each of the bars of the L should be 20 steps;
- The height of the L should be 100 steps;
- The width of the L should be 80 steps;



Calculating turns

This diagram shows the two sets of parallel lines in the letter L in blue and red.



Use angle relationships on parallel lines (previous slide) to calculate the turns for your drawing.

Testing

- [ ] Testing the left side of a 60° tilted L.
- [ ] Testing the top of a 60° tilted L.
- [ ] Testing the right side of a 60° titled L.
- [ ] Testing the top of the flat part of a 60° tilted L.
- [ ] Testing the right end of a 60° titled L.

- [ ] Testing the bottom of a 60° tilted L.
- [ ] **Testing the whole 60° tilted L.**

4.2.8. Problem: Italic T

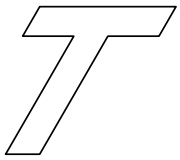
Using *italics* is important for adding emphasis and changing the voice in which a word or phrase is spoken.

Write a program that draws an italic letter T, so the angle in the bottom left corner is 60°.

Only the angles of the turns should change from the previous straight T question, the lines should all be the same length:

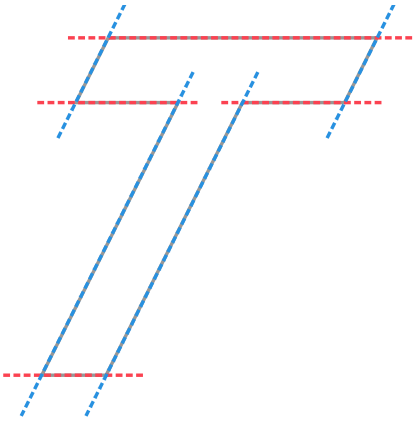
- The thickness of each of the bars of the T should be 20 steps;
- The height of the T should be 100 steps;
- The width of the T should be 80 steps.

It should look like this:



Calculating turns

This diagram shows the two sets of parallel lines in the letter T in blue and red.



Use angle relationships on parallel lines (previous slide) to calculate the turns for your drawing.

Testing

- [ ] Testing the stem's left side on a 60° tilted T.
- [ ] Testing the left arm of a 60° tilted T.
- [ ] Testing the bottom of a 60° tilted T.
- [ ] Testing the stem's right side on a 60° tilted T.

- [ ] Testing the right arm of a 60° tilted T.
- [ ] Testing the top of a 60° tilted T.
- [ ] **Testing the whole 60° tilted T.**

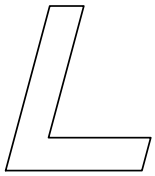
4.2.9. Problem: Any Italic L

Let's let the user decide how much *italicisation* they want.

Write a program that asks the user for the angle of the bottom left corner, and tilts the letter L to that angle.

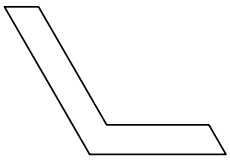
For example, if the user enters 75:

```
Enter the angle: 75
```



Or an obtuse angle to tilt to the left:

```
Enter the angle: 120
```



An angle of 90° is the straight letter L.

The angles of the turns should change depending on what the user inputs, but the lines should all be the same length as in the Straight L and Italic L questions:

- The thickness of each of the bars of the L should be 20 steps;
- The height of the L should be 100 steps;
- The width of the L should be 80 steps;

Testing

- [ ] Testing the left side of a 75° tilted L.
- [ ] Testing the top of a 75° tilted L.
- [ ] Testing the right side of a 75° tilted L.
- [ ] Testing the top of the flat part of a 75° tilted L.
- [ ] Testing the right end of a 75° tilted L.
- [ ] Testing the bottom of a 75° tilted L.
- [ ] **Testing the whole 75° tilted L shape.**
- [ ] Testing for no extra lines in the 75° tilted L.
- [ ] Testing the left side of a 120° tilted L.
- [ ] Testing the top of a 120° tilted L.
- [ ] Testing the right side of a 120° titled L.
- [ ] Testing the top of the flat part of a 120° tilted L.
- [ ] Testing the right end of a 120° titled L.
- [ ] Testing the bottom of a 120° tilted L.
- [ ] **Testing the whole 120° tilted L.**
- [ ] **Testing a 20° tilted L.**
- [ ] Testing a 130° (backwards) tilting L.
- [ ] Testing a hidden case.
- [ ] **Well done, this was a tough problem!**

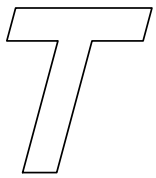
4.2.10. Problem: Any Italic T

Let's let the user decide how much *italicisation* they want.

Write a program that asks the user for the angle of the bottom left corner, and tilts the letter T to that angle.

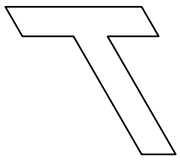
For example, if the user enters 75:

```
Enter the angle: 75
```



Or an obtuse angle to tilt to the left:

```
Enter the angle: 120
```



An angle of 90° is the straight letter T.

The angles of the turns should change depending on what the user inputs, but the lines should all be the same length as in the Straight T and Italic T questions:

- The thickness of each of the bars of the T should be 20 steps;
- The height of the T should be 100 steps;
- The width of the T should be 80 steps;

Testing

- [ ] Testing the left side of a 75° tilted T.
- [ ] Testing the top of a 75° tilted T.
- [ ] Testing the right side of a 75° tilted T.
- [ ] Testing the top of the flat part of a 75° tilted T.
- [ ] Testing the right end of a 75° tilted T.
- [ ] Testing the bottom of a 75° tilted T.
- [ ] **Testing the whole 75° tilted T shape.**
- [ ] Testing for no extra lines in the 75° tilted T.
- [ ] Testing the left side of a 120° tilted T.
- [ ] Testing the top of a 120° tilted T.
- [ ] Testing the right side of a 120° titled T.
- [ ] Testing the top of the flat part of a 120° tilted L.
- [ ] Testing the right end of a 120° titled T.
- [ ] Testing the bottom of a 120° tilted T.
- [ ] **Testing the whole 120° tilted T.**
- [ ] **Testing a 20° tilted T.**
- [ ] Testing a 130° (backwards) tilting T.
- [ ] Testing a hidden case.
- [ ] **Well done, this was a tough problem!**

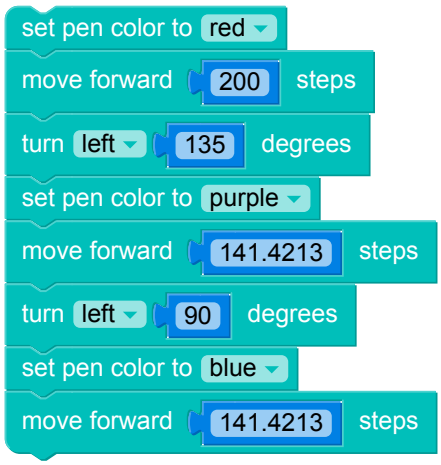
5

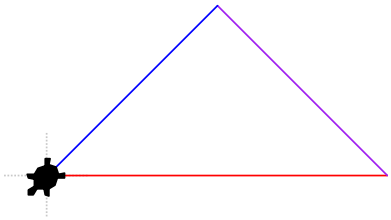
COLOURS AND LOOPS WITH TURTLE

5.1. Drawing with colour

5.1.1. With flying colours!

The colour of the line can be changed using the set pen color block:





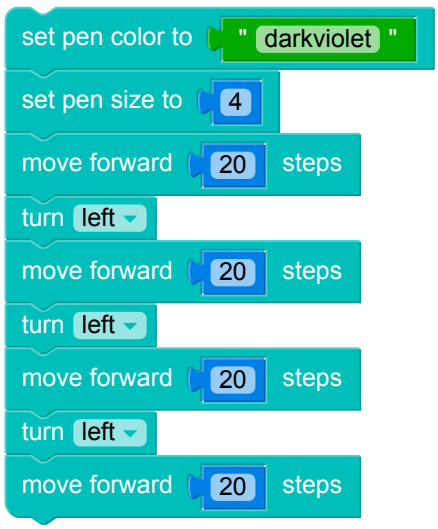
Try changing the colours in the isosceles triangle (https://en.wikipedia.org/wiki/Isosceles_triangle) above!

Most code uses *color* (American spelling)!

Most programs and modules (like `turtle`) will spell colour with the American spelling (c-o-l-o-r, no u) – so watch out!

5.1.2. More colour choices

Blockly has a version of the set pen color block that describes the colour as a string rather than a drop-down list:



The square example above also changes the pen size - you can change the colour and the thickness of the line in the same program.

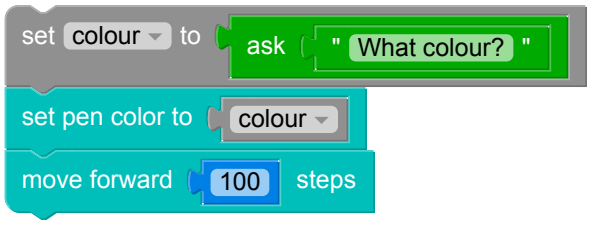
The full set of colour strings that the `turtle` knows about are here (http://wiki.tcl.tk/37701). Just remember that with all colours you need to take the spaces out (so `"forest green"` becomes `"forestgreen"`).

Here are some of our favourite colours:

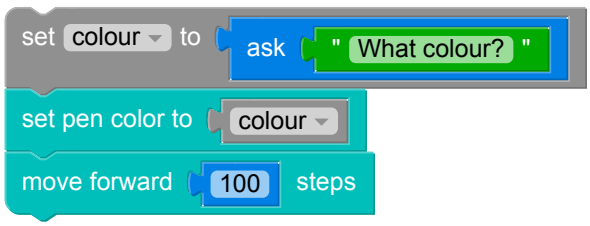
			gold	yellow
		springgreen	lawngreen	green
				skyblue
				plum
		pink	lightpink	mistyrose
			tan	wheat
			lightgray	white

5.1.3. Text input

Just like we can ask for numbers, we can also ask the user for text. This looks a little bit different - since text is a string, we use a green ask block this time.



Since strings and numbers are different data types, if you try to use the number (blue) ask block, your program will throw an error when you type in your text value:

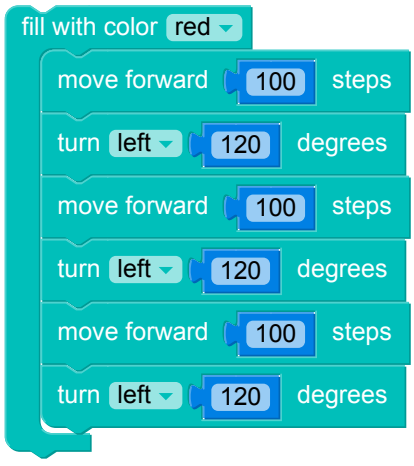


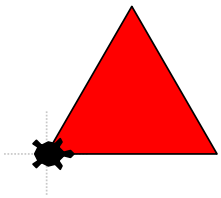
Be careful when choosing which ask block to use! Use the blue one for numbers, and the green one for text.

5.1.4. Filling with colour

The fill with color block can be used to fill in a shape with colour.

Any shapes which you draw inside of the fill block will be filled with the colour you choose (at the end of drawing the shape).

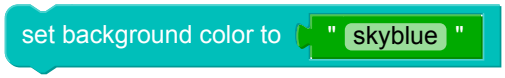


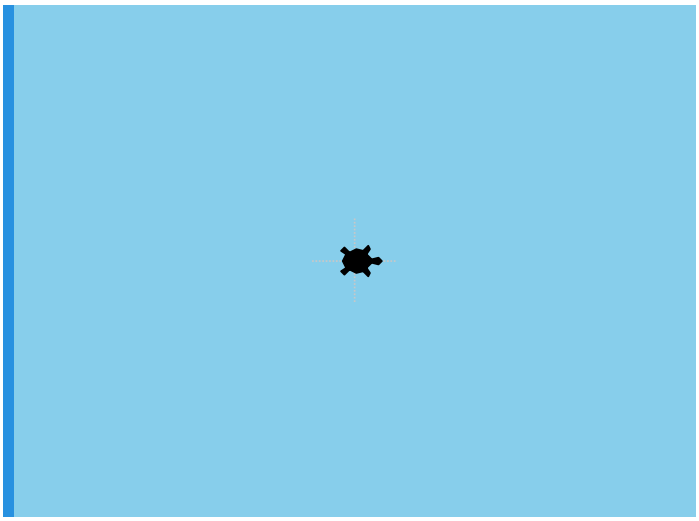


Try guessing what this program will draw before running the example! Then try changing the fill colour!

5.1.5. Background colours

It's boring to always have a plain white background. That's why we have a background color block.





Also try some different colours.

The colour names that work are the same as the pen color and fill color blocks.

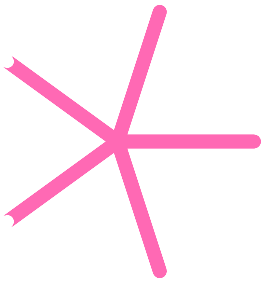
5.1.6. Problem: Simple Star

Let's draw a 5-pointed star with a thicker pen and some colour!

Write a program to:

- Set the pen size to 8.
- Set the pen colour to `'hotpink'`.
- Draw a 5 pointed star with arms 80 turtle steps and turning 72° between each arm.

When it's finished, the result should look like this:



Testing

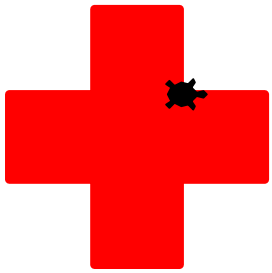
- [ ] Testing the first "arm" of the star.
- [ ] Testing the first and second "arms" of the star.
- [ ] Testing the top half of the star.
- [ ] Testing the whole star.
- [ ] Testing that there are no extra lines.

5.1.7. Problem: Red Cross

Write a program to draw a red, filled cross where all the sides are 50 turtle steps long. Your program should ask the user for the line thickness and then draw a red cross with that thickness.

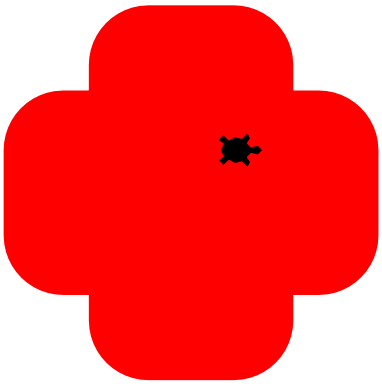
Here's an example of a red cross with line thickness 5:

```
Line thickness: 5
```



Here's an example of a red cross with line thickness 70:

```
Line thickness: 70
```



Don't forget Duplicate!

Remember that once you've made a block you're going to re-use in your code, you can right-click on it to make a copy.

Testing

- [ ] Testing the first example from the question, 5.
- [ ] Testing the second example from the question, 70.
- [ ] Testing another line thickness, 25.
- [ ] Testing another line thickness, 2.

- [ ] Testing a hidden case.

5.1.8. Problem: Witches hats

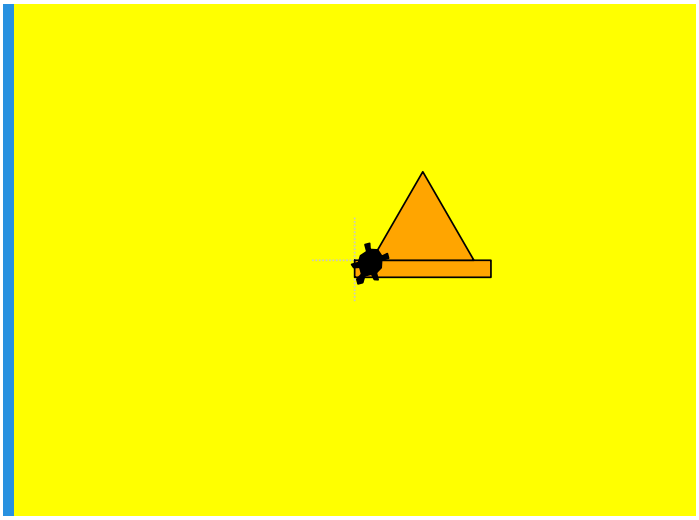
Witches hats (also known as traffic cones (https://en.wikipedia.org/wiki/Traffic_cone)) are seen everywhere, especially on roads where there is some form of construction or work going on. You've probably also seen them in your PE class! They get their name from their shape, which is the conical shape of hats typically worn by witches in woodcuts from the Middle Ages (https://en.wikipedia.org/wiki/Witch_hat).

Write a program that asks the user for a **`Colour`** and **`Background`** (in that order), then draws a witches hat to those specifications. The user can type in any of the colours the turtle knows about (http://wiki.tcl.tk/37701). The hat has the following characteristics:

- base: a rectangle with width 80 and height 10
- hat: an equilateral triangle with side lengths 60
- the hat is positioned horizontally at the centre of the base, and shares an edge with it

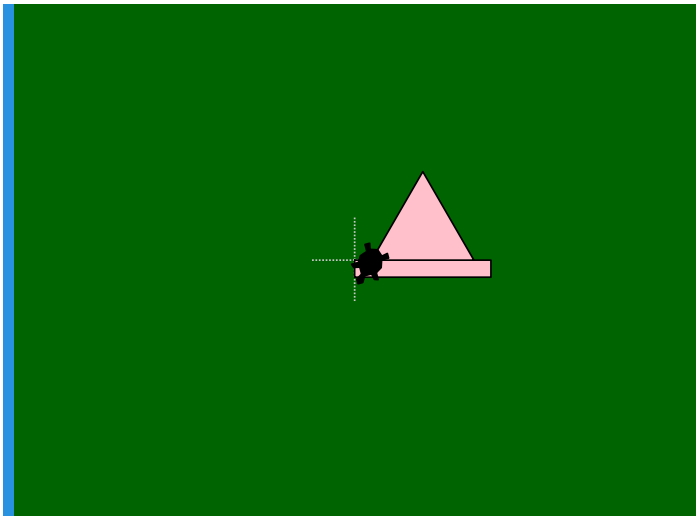
The turtle starts at the *top left corner of the base of the hat*. Here's an example:

```
Colour: orange
Background: yellow
```



And another example:

```
Colour: pink
Background: darkgreen
```



Testing

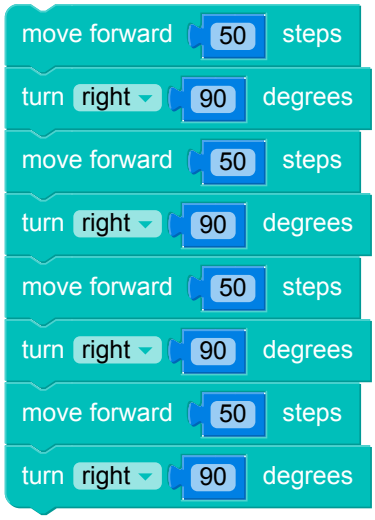
- [ ] Testing the background colour in the first example.
- [ ] Testing the witches hat in the first example.
- [ ] Testing for no extra lines in the first example.
- [ ] Testing the second example from the question.
- [ ] Testing a tomato witches hat on a lightgray background.
- [ ] Testing a wheat witches hat on a mediumblue background.
- [ ] Testing a hidden test case.
- [ ] Testing another hidden test case.

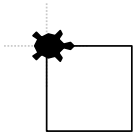
5.2. Loops

5.2.1. Drawing shapes with loops

You probably noticed that you repeated yourself a lot in turtle programs. Using loops makes turtle much less repetitive!

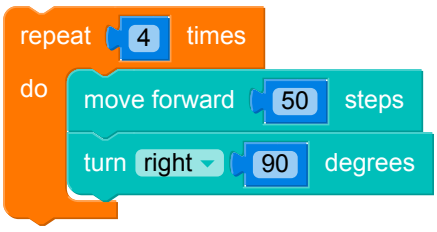
Drawing a square the long way:

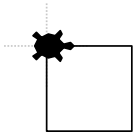




If we used a loop, then we wouldn't have to repeat same two blocks over and over again.

Here's a much shorter way of drawing a square, using loops:



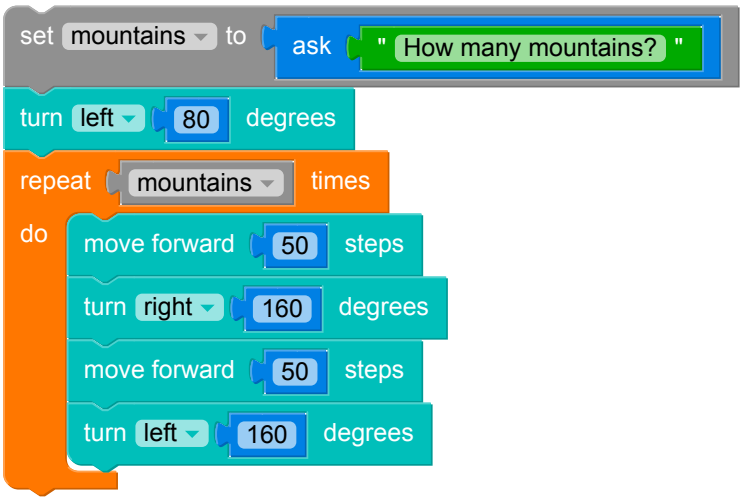


5.2.2. Repeating shapes

Using loops also lets us do things that we couldn't do before.

We can ask the user to enter a number, and we loop a different number of times depending on that number.

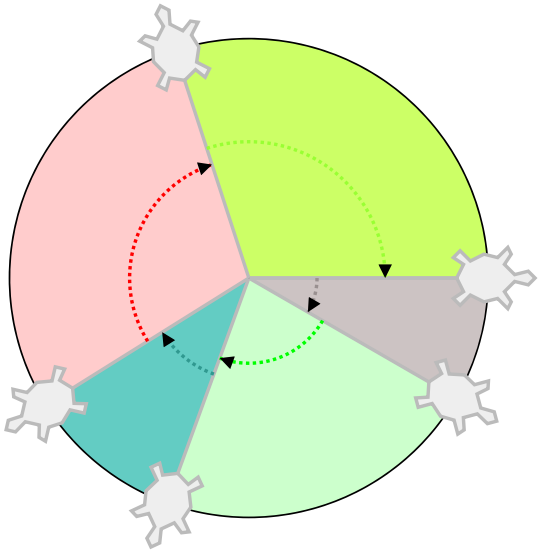
Here's a program which draws a row of mountains (a zig-zag line):



Using this one program, we can draw any number of mountains! Try it out with several different numbers.

5.2.3. Adding up angles

When a set of lines meet at a point, the angles between all of those lines always add up to 360°.

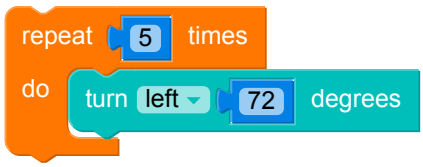


30° + 80° + 38° + 104° + 108° = 360°

Imagine the turtle is turning left to face each line, one at a time. If the angle of each turn adds up to 360° then it has turned a full circle. In fact, here's a program for the turtle to turn 5 times and it ends up facing where it started.

$$5 \times 72° = 360°$$

Since we're turning left 5 times and each time we're turning 72°, we end up turning one full circle.



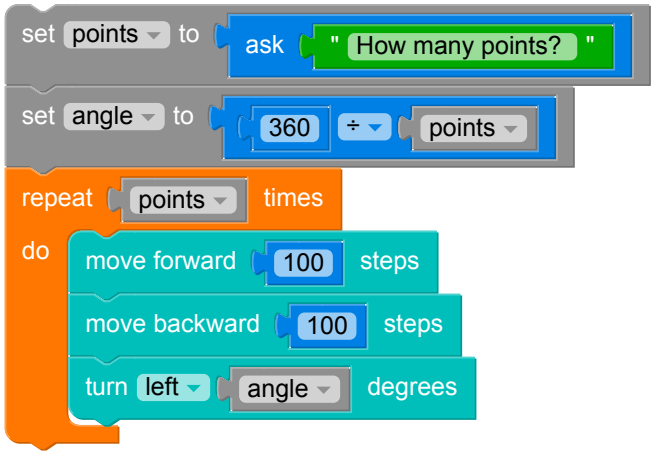
5.2.4. Drawing stars

Using our knowledge that turning a full circle should add up to 360° we know that if we want to make 8 even turns, we should turn 45° each time.

$$\frac{360°}{8} = 45°$$

The angle to turn is always 360° divided by the number of turns we want to make.

We can use this formula more generally to draw the same shape (in this case a line) many times around in a circle to make a star with any number of points!

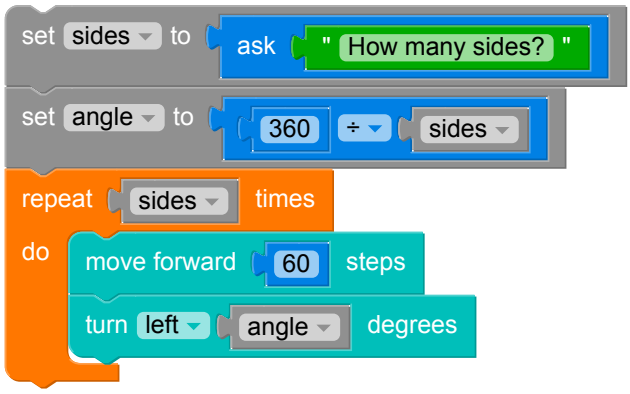


5.2.5. Regular polygons

We can use the same calculation to draw regular shapes!

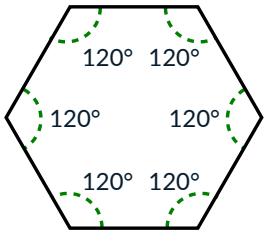
You can imagine a turtle drawing a shape by turning left at each corner. The turns that the turtle makes are the *exterior angles*.

To draw the whole shape, the turtle must turn one full circle to walk around all the edges, so all the turns it makes (the exterior angles) add up to 360°.



5.2.6. Problem: Hexagon

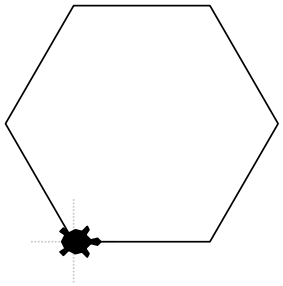
A *hexagon* is a shape with six sides. A regular hexagon has internal angles of 120°:



Write a program that asks the user for a side length, then draws a regular hexagon. The turtle starts at the bottom left corner of the bottom side.

Here's an example with a side length of 80:

```
Side length: 80
```



The prefix hex (or ἕξ) means six in Ancient Greek.

Start with just a couple of lines

If you're not sure how to draw the hexagon, start with just a move, turn and move. Then run it to see how it looks. You'll need to think about how to calculate the turn angle.

Remember to use your new loop blocks!

Testing

- [ ] Testing the hexagon in the example.
- [ ] Testing for no extra lines.
- [ ] Testing a smaller hexagon (side length 25).
- [ ] Testing another hexagon (side length 55).
- [ ] Testing a hidden case.
- [ ] Testing another hidden case.

5.2.7. Problem: A row of houses

You learned how to draw a single house before. Using your new knowledge about loops, you're going to draw a whole street of houses!

Write a program that asks for two things - the size of each house, and the number of houses (in that order) - then draws the whole street!

```
How big is a house? 20
How many houses? 4
```

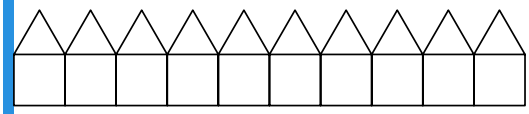
Your program should start off by moving 200 turtle steps to the left so that it starts off at the left-edge of the screen. Your program should work for any number of houses.

Important things to note:

- You need to make sure you pick the pen up and put it down at the right time.
- The turtle starts at the ceiling of each house - the left of the line where the room meets the roof.

Here's another example:

```
How big is a house? 30
How many houses? 10
```



Testing

- [ ] Testing the first example in the question.
- [ ] Testing the second example in the question.
- [ ] Testing with just two houses of size 50.
- [ ] Testing with three houses of size 45.
- [ ] Testing with 38 houses of size 10.
- [ ] Testing a hidden case.
- [ ] Testing another hidden case.

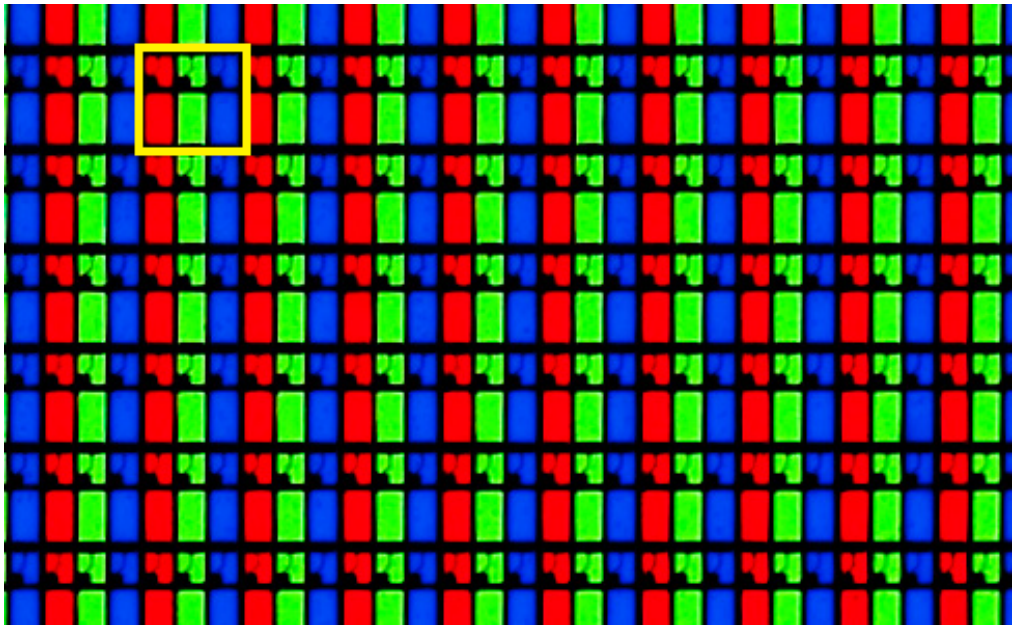
6

MORE COLOURS AND LOOPS

6.1. Representations of colour

6.1.1. Colour in computers

Maths is important in all aspects of computing, especially colours! A computer stores the colour information about pixel colour as three numbers - an amount of red, green and blue. The computer then combines these individual amounts of colour to create the colour you see on your screen.



This is what an LCD monitor looks like when you zoom in. A single pixel is made up of red, green and blue parts.

This RGB Colour Model (https://en.wikipedia.org/wiki/RGB_color_model) is an example of an additive colour model, because the final colour is created by adding the separate colours together.

6.1.2. The use color range block

The blocks we use to write our programs are actually generating code in the Python programming language (https://www.python.org/). You would have seen the python code appear underneath your blocks in the questions you've been answering.

When using the turtle, Python needs to know what range you are using to describe each of your colours. We do this using the use color range block, and it has two settings:

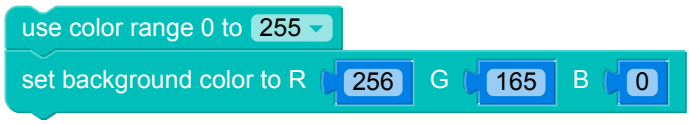
- 255 - Colour values are set as a range from 0 - 255
- 1.0 - Colour values are set as a range from 0.0 - 1.0

We'll be using the 255 setting in all of our examples and problems, so you should always choose that setting in this course.

The value you set is a measure of the intensity of the colour, so the higher the value the more of that colour is added to the mix. A value of zero means no colour (black), and a value of 255 means that colour is set to its highest value (e.g. 255 on the red channel means 100% red).

6.1.3. Numbers outside the range

Once you set your color range you need to make sure you always use values that are in that range, otherwise your program will raise an error. The example below tries to use a value of 256 which is larger than the maximum 255 allowed in the selected color range.



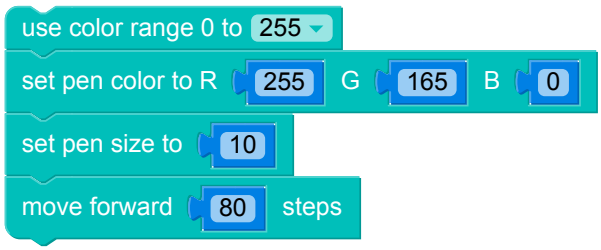

```
Traceback (most recent call last):
  File "program.py", line 5, in
    bgcolor(256, 165, 0)
  File "/opt/grok/python3/turtle.py", line 2237, in bgcolor
    return WORLD.api_bgcolor(check_color(*color))
  File "/opt/grok/python3/turtle.py", line 1597, in check_color
    raise ValueError('{} component of colour must be in the range [0, {}] (not {})'.f
ValueError: red component of colour must be in the range [0, 255] (not 256)
```

The error message generated gives you a hint about the error in your code - this is telling us that the red component of our colour needs to be in the range 0 - 255, not the 256 we have used. You can change the values in the example above to see how the error message changes for incorrect green and blue values.

6.1.4. Using RGB with blocks

Once you've defined your colour range, you can then use the set pen color to RGB block any time you want to set a colour. This means you can access even more colours to use in your programs (in fact, with 255 values possible in each of red, green and blue, that's 255 × 255 × 255 = 16,777,216 different colours!).

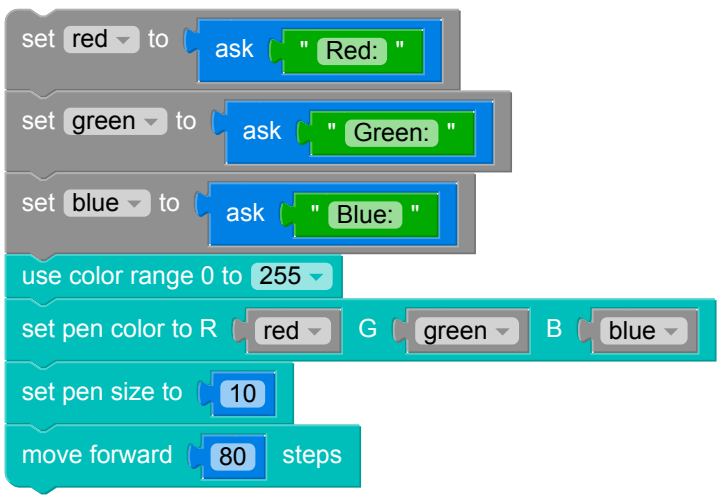
Use the example code below to draw a line of any colour by changing the value of the red, green and blue values, and see what cool colours you can make!



Remember, you should always set your colour range before using the RGB blocks!

6.1.5. Variables and RGB

To make your program even more flexible so that it changes each time it is run, you can use variables to collect values from the user for each of the RGB channels, then use these in the set pen color to RGB block. You don't have to modify your code, and the user can now create any of the 16 million colours themselves.

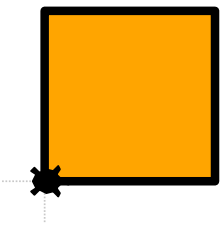


There are also RGB versions of the set fill color and set background color blocks, so you can use RGB colours for any part of your drawings.

6.1.6. Problem: Any coloured square

Write a program that allows you to draw any coloured square you can think of! It should do this by using a black pen of thickness 5 to draw a 100 × 100 square, then filling that square in with a colour defined by three values, typed in by the user. Your program should ask the user for their red component first, then the green, then the blue, just like in this example:

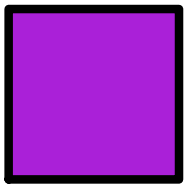
```
Red: 255
Green: 165
Blue: 0
```



The turtle starts at the bottom-left corner of the square.

Here's another example with different values:

```
Red: 170
Green: 32
Blue: 216
```



Testing

- [ ] Testing the first example in the question.
- [ ] Testing the second example in the question.

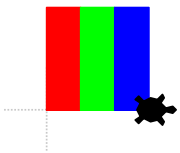
- [ ] Testing an example with lots of red (235, 40, 15).
- [ ] Testing an example with lots of green (50, 240, 50).
- [ ] Testing an example with lots of blue (10, 35, 230).
- [ ] Testing a hidden case.
- [ ] Testing another hidden case.

6.1.7. Problem: Pixel

On the first slide in this module we showed you a close up of an LCD screen. In this problem, you're going to write a program that draws a zoomed in view of a single pixel.

Ask the user for a value, then use this value to draw the individual red, green and blue parts of the pixel. Your pixel will be a square with side lengths of 60 steps, and the red, green and blue parts will be equal in width. Your pen should be the same colour as the part it is drawing, since we do not want to see any lines on the screen.

```
Value: 255
```



The turtle starts at the bottom-left corner of the pixel.

And another example:

```
Value: 57
```

Testing

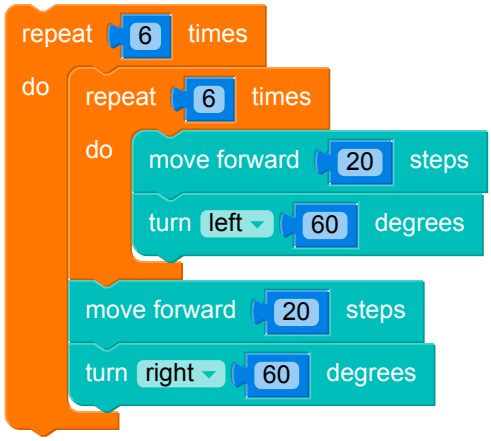
- [ ] Testing the first example in the question.
- [ ] Testing the second example in the question.

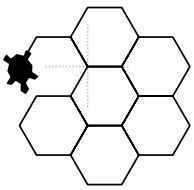
6.2. Loops in loops

6.2.1. Loops inside loops

When we use a loop block, it repeats everything inside of it. If we put another loop inside a loop block we can do even more fun things!

Try to work out what this is going to print before you click run!



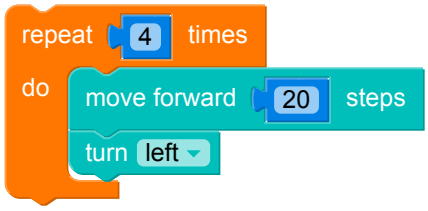


The whole inside loop is repeated each time the outside loop repeats.

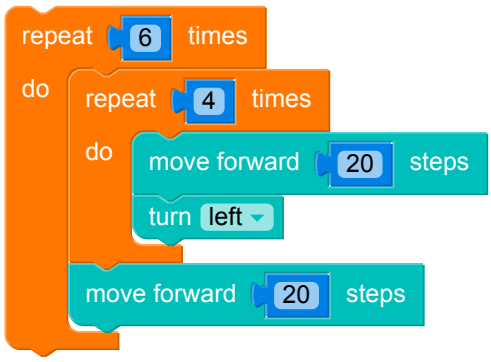
6.2.2. Looping shapes

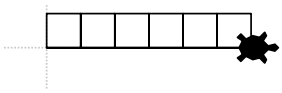
A lot of the shapes we've been drawing throughout the course can be simplified with loops. This is because the way the shape is drawn is by repeating the instructions you use to move the turtle.

Here's a loop that draws a square:



We can then use another loop to repeat our square code, moving the turtle between squares to reposition it for the next one:

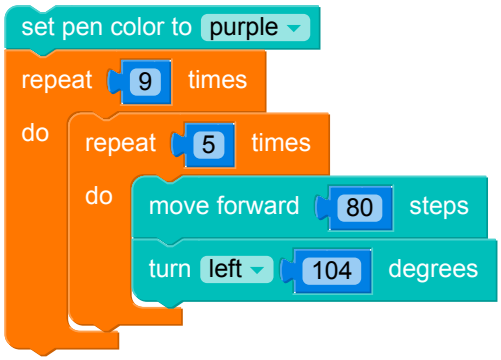


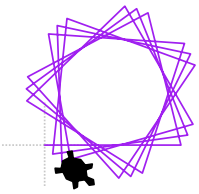


When creating code with a loop in a loop, you should always make sure the inner loop is correct first. So, in this example, draw one square and move the turtle to its new starting position before you make the outer loop.

6.2.3. Interesting patterns

You can make all kinds of interesting patterns by trying different things with your loops.





This type of drawing is known as a spirograph (https://en.wikipedia.org/wiki/Spirograph).

Notice how in this example, the turtle ends up back where it started? This is due to the mathematical relationship between our angle and the number of times we loop in the program.

$$104^\circ \times 5 = 520^\circ$$

In our inner loop, our turtle turns 520°. Since this isn't a multiple of 360°, the turtle doesn't end up starting where it finishes (which is what we want for this pattern!)

$$520^\circ \times 9 = 4680^\circ$$

$$4680^\circ \div 360^\circ = 13$$

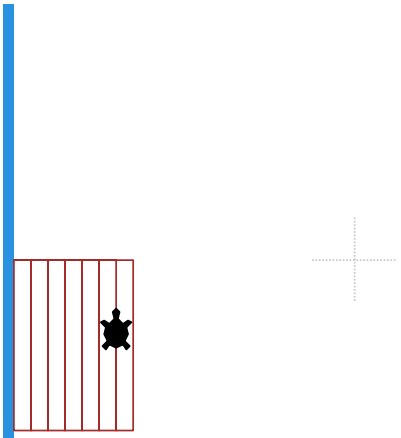
If we repeat our inner loop 9 times, the turtle rotates a total of 4680°, which is a multiple of 360° since it divides equally into 13.

Use this relationship to try different values and see what other patterns you an create!

6.2.4. Problem: Wooden Fence

To build the whole fence it will take 40 planks of wood, but you don't have that many. Write a program to see how much fence you can build with the planks that you have:

```
How many planks? 11
```



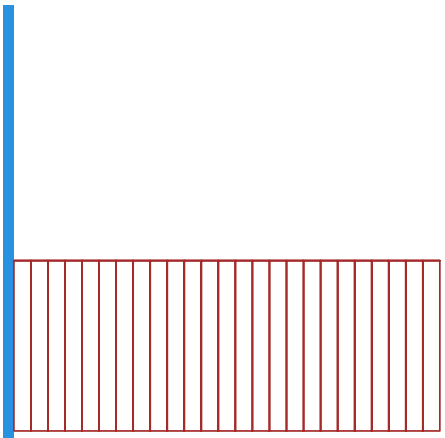
Your program should start off by moving 200 turtle steps to the left so that it starts off at the left-edge of the screen. Your program should work for any number up to 40 planks.

Important things to note:

- All lines should use pen colour `" brown "`;
- Each plank should be 100 steps high and 10 steps wide;
- The top of the fence should be the center of the space.

Here's another example:

```
How many planks? 25
```



Hint

You can use a loop to make one fence post, and then a loop to draw the fence itself.

Testing

- [ ] Testing the first example in the question.
- [ ] Testing the second example in the question.
- [ ] Testing with just two planks.
- [ ] Testing with three planks.
- [ ] Testing with 39 planks.
- [ ] Testing with all 40 planks.
- [ ] Testing a hidden case.
- [ ] Testing another hidden case.

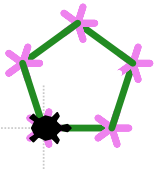
6.2.5. Problem: Daisy Chains

Ever picked some daisies and made a daisy chain crown or necklace? Let's draw a daisy chain with the turtle!

```
Number of flowers: 1
Petals on each flower: 5
```

When there's more than 2 flowers it should form a circle like this:

```
Number of flowers: 5
Petals on each flower: 5
```

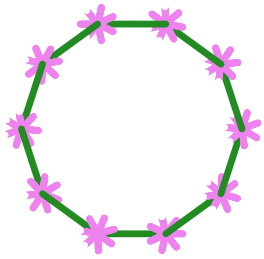


Some details:

- All lines should use pen size 4.
- The stems of each flower should be `'forestgreen'`.
- Each stem should be 40 steps long.
- The petals of each flower should be `'violet'`.
- Each petal should be 10 steps long.
- The first petal should be in the same direction as the stalk.

Here's an example with lots of lowers with more petals on each flower:

```
Number of flowers: 10
Petals on each flower: 7
```



Testing

- [ ] Testing the first (one flower) example from the question.
- [ ] Testing the second example from the question.
- [ ] Testing a single flower with 7 petals.
- [ ] Testing the third example from the question.
- [ ] Testing a single flower with 4 petals.
- [ ] Testing an octagonal ring of four petal flowers.
- [ ] Testing a hidden case.

7

TRANSLATION AND ROTATION

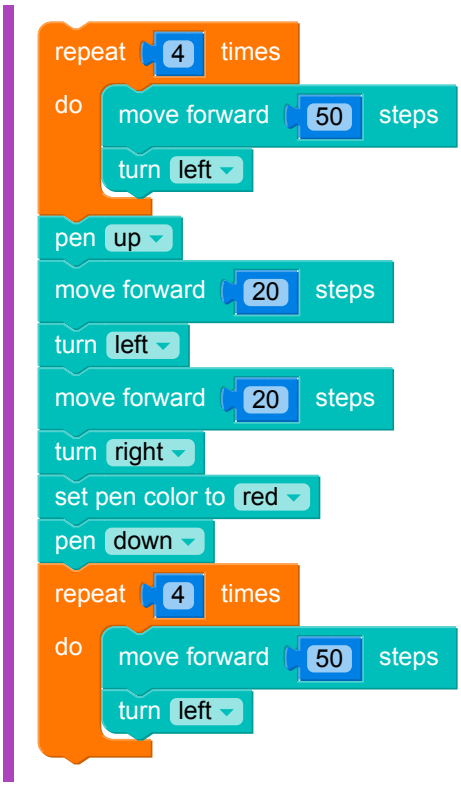
7.1. Translation

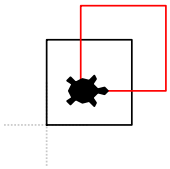
7.1.1. What is translation?

This module focuses on geometric transformations - ways that we can change how our shapes are drawn. The first of these we'll investigate is translation.

Translation is a transformation that moves every point in the shape the same distance in the same direction. We've already been using translation a lot in our course - quite a few of our loop questions involved some translation.

The example below draws a 50 × 50 square, then changes the pen colour and draws another 50 × 50 square in a new position. That position is 20 turtle steps up and 20 turtle steps right from the original square's position:

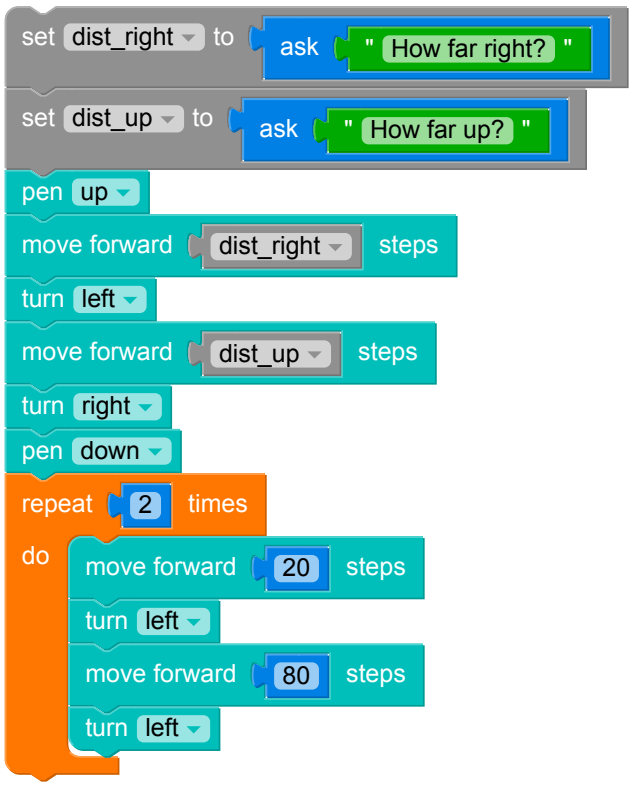




7.1.2. Uses of translation

Translation is used in computers all the time to position images and text on your screen. Each image or letter is defined by drawing some lines in a specific direction (just like we explored in the Vector Graphics module), and then translation is used to place it in the correct location on the screen.

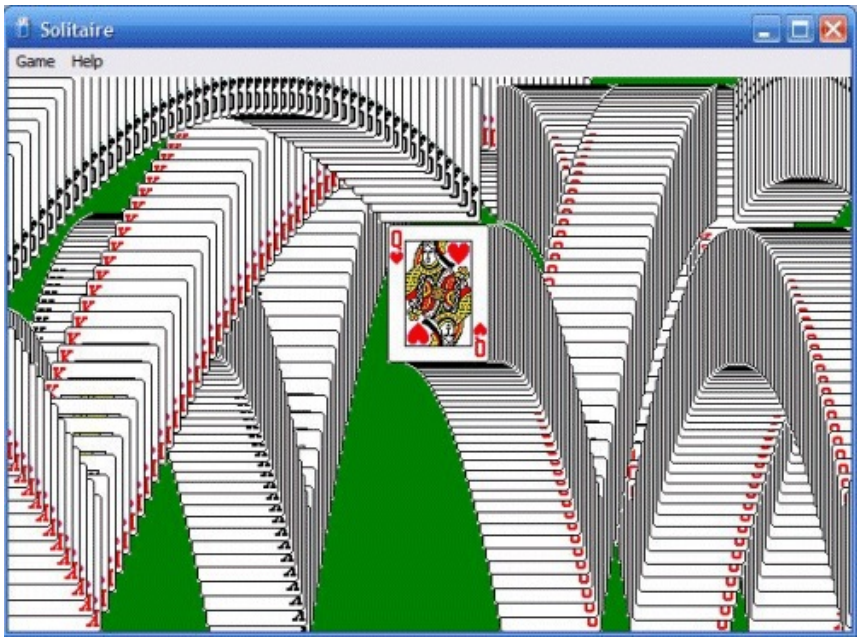
The example below asks you to indicate where you would like to position the letter I (a rectangle) by specifying how far you want to move it right, and how far you want to move it up. If you want to move it left or down, you just use negative numbers:



Run this example and watch how the turtle moves when you use different values. You can position the I anywhere on the screen - try different positive and negative numbers and see what happens.

7.1.3. Problem: Solitaire

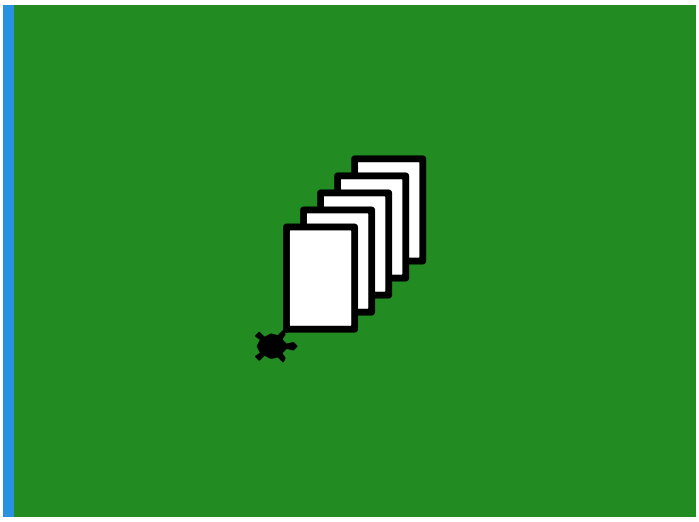
When you win at Microsoft Solitaire (https://en.wikipedia.org/wiki/Microsoft_Solitaire) the piles of cards bounce out at you!



Screenshot used with permission from Microsoft.

Write a program to draw a pile of cards! Your program should ask how many cards in the pile, and draw that many.

```
How many cards? 5
```

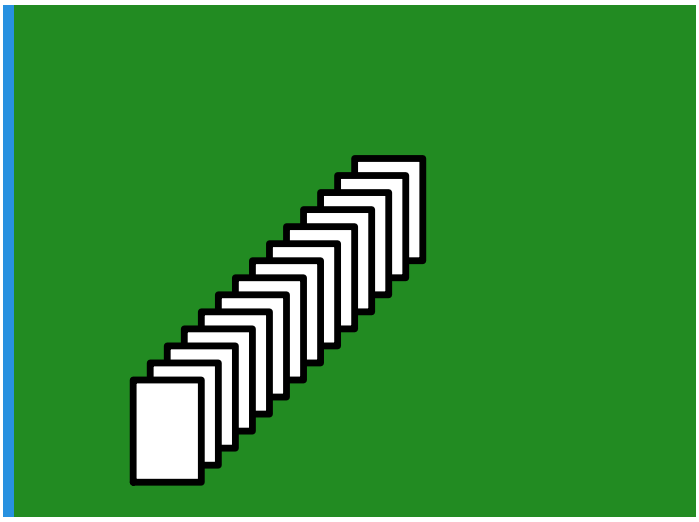


Some details:

- The background should be coloured `'forestgreen'`.
- The first card's bottom-left corner should be in the centre of the space (where the turtle starts).
- The pen should be `'black'` and size 4.
- The fill colour `'white'`.
- Each card should be 40 steps wide and 60 steps high.
- Each card is 10 steps lower and 10 steps to the left of the card below.

Here's an example with more cards!:

```
How many cards? 14
```



Testing

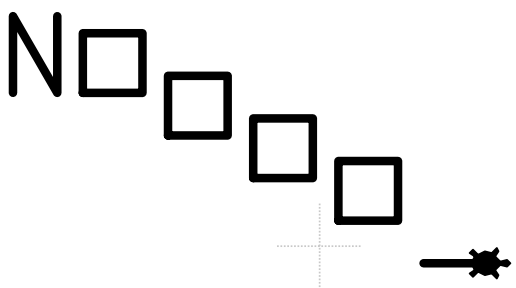
- [ ] Testing just one card.
- [ ] Testing with two cards.
- [ ] Testing the first example from the question (5 cards).
- [ ] Testing the second example from the question (14 cards).
- [ ] Testing with 9 cards.
- [ ] Testing a hidden case.

7.1.4. Problem: Nooooooooooo

Any time a character in a comic, cartoon or some other kind of animated short falls off a cliff or building, we often hear or see the word "Nooooooooo...." as they fall to their demise. You can use your knowledge of translation to have the turtle draw your own comic text for this exact situation.

Write a Turtle program to draw the letter N in the top left corner, followed by six letter Os. Some details about the letters:

- The pen size for all letters should be 5.
- The left and right lines of the N should be 45 turtle steps long and **parallel to each other**.
- The diagonal line should be 52 turtle steps long, and should be 30° from the top of the left line.
- The starting position for the N will be 180 turtle steps left from the start position, and 130 steps up.
- Each O will be drawn as a square that will have a side length of 35 steps.
- Each letter will be 15 steps to the right of the right edge of the previous letter.
- The first O will be at the same height as the N, but each subsequent O will be 25 steps lower than the previous letter.



Hint

Work out all of the distances and directions you will need to be translating your letter Os before you start coding - it will make it much easier!

Testing

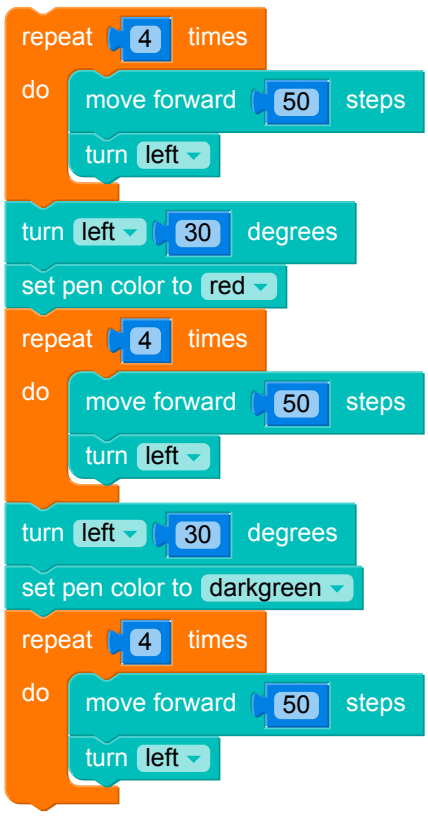
- [ ] Testing the letter N.
- [ ] Testing the first O.
- [ ] Testing Noooooo.
- [ ] Testing for no extra lines.

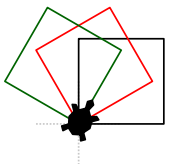
7.2. Rotation

7.2.1. What is rotation?

Rotations describes how shapes move around a point in two-dimensional space. Where translation moves the turtle to a new position but leaves it facing the same way, rotation changes the starting direction of the turtle.

The example below draws a 50 × 50 square, changes the pen colour and draws another 50 × 50 square with a rotation angle of 30°, and does this one more time but with a green pen:

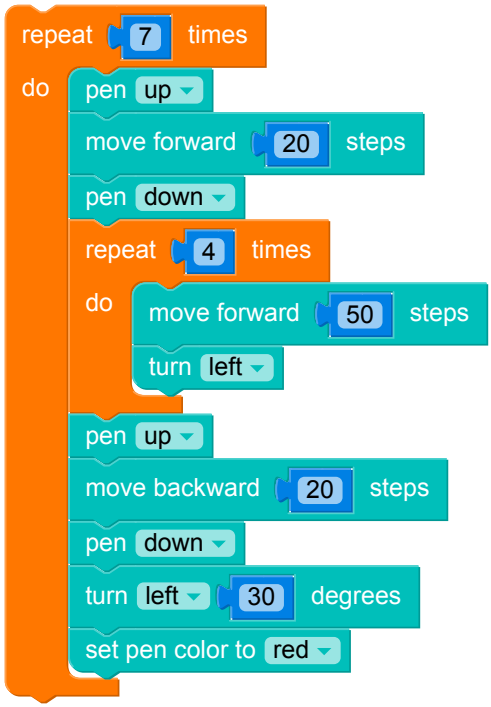


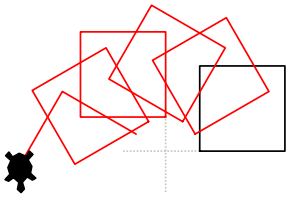


7.2.2. Rotation around different points

In our previous example, we rotated the shape around the turtle's starting point, meaning the squares all shared a common corner. Sometimes, you want to rotate an image around a point that isn't on the corner of the shape.

To make this work, you need to combine translation and rotation. You include the initial translation from the origin point inside the code that you repeat, and make sure the turtle moves back to the starting point before you perform the rotation:





Here, we draw our original square in black, moving the turtle forward 20 steps before putting the pen down. After we finish the square, we move the turtle back to the start, then draw 6 more squares using the same process, but with the red pen.

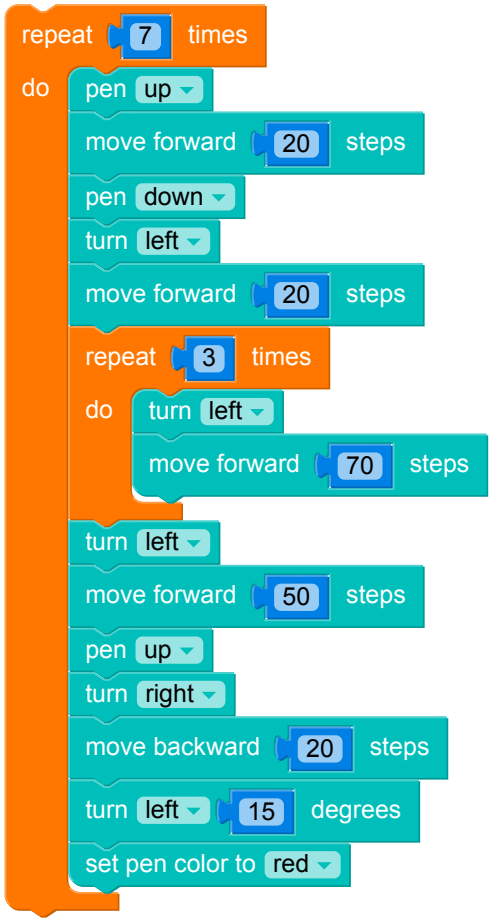
Notice that instead of sharing a common corner, all of these squares are 10 steps away from the origin point, but each one is rotated 30° from the previous one.

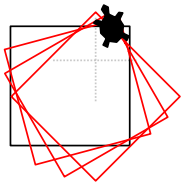
7.2.3. Using rotation

Rotation is used all the time in computers. In everything from a word processor or image editing program through to computer games, items on the screen are often rotated either by the user or the player. The items themselves aren't changed - instead, the items are rotated around a defined point by setting a

rotation angle and performing the same kind of instructions we saw on the last slide.

If you want to rotate a shape around a point inside the shape, then you draw the shape as if the turtle's starting point is inside it. Here's an example:



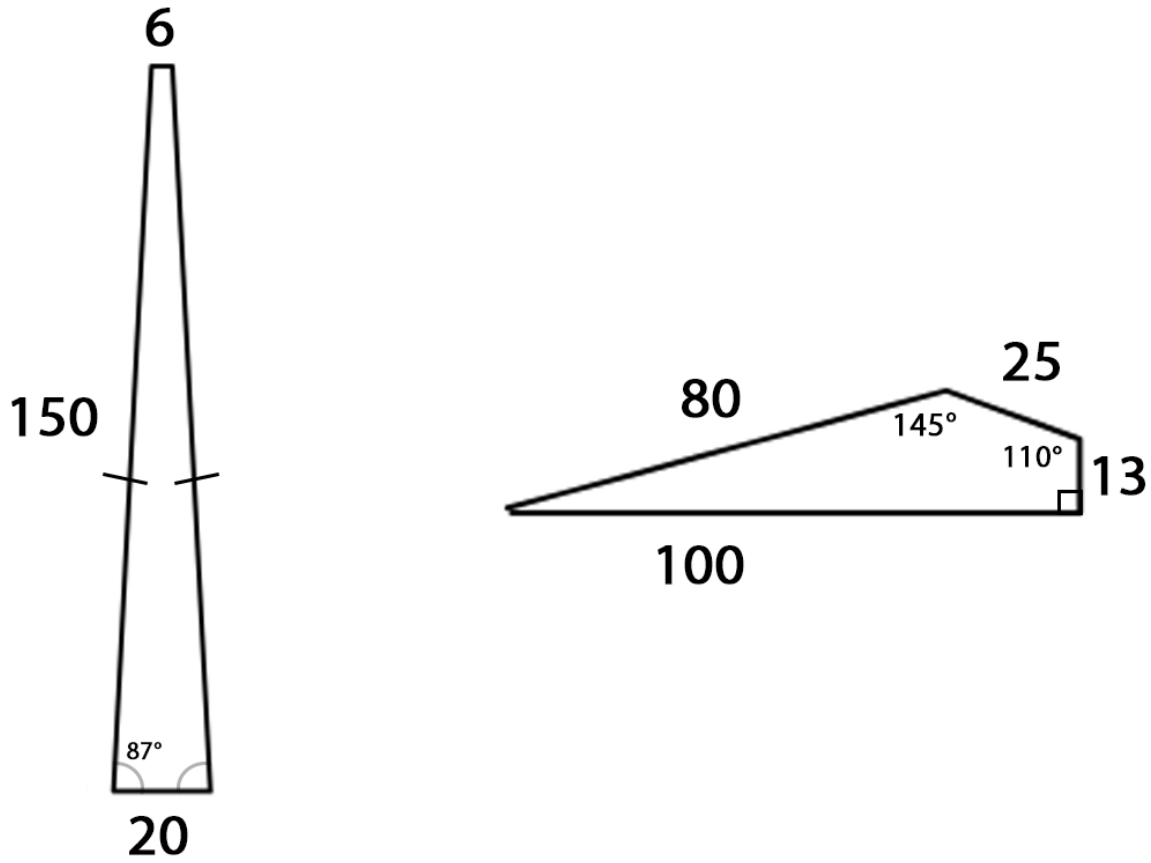


7.2.4. Problem: Wind turbine

Modern wind turbines (https://en.wikipedia.org/wiki/Wind_turbine) can use any number of blades, with two or three blades the most common. Usually, the blades are all the same shape and evenly spread around the rotor. We can use this information to draw a simple wind turbine using the turtle.

The details of the wind turbine drawing are:

- The background colour is sky blue.
- The base of the turbine is coloured rgb(225, 225, 225).
- The turbine blades are white.
- The base is an isosceles trapezium with dimensions shown in the image below.
- Each of the blades is a quadrilateral with the dimensions shown in the image below.
- The turtle starts at the centre of the top of the base, which is also the rotation point for the turbine blades.



Dimensions for the wind turbine base and blades

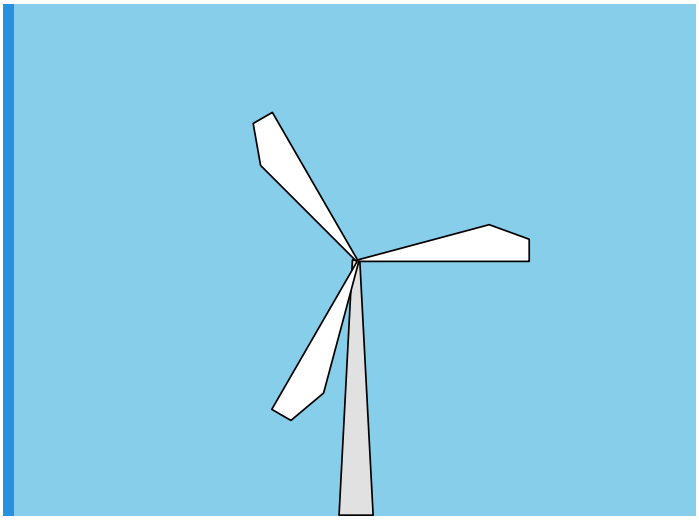
You've been provided with some starter code that draws the base of the turbine for you, but is using the wrong colours.

Your program should be able to draw a wind turbine with any number of blades.

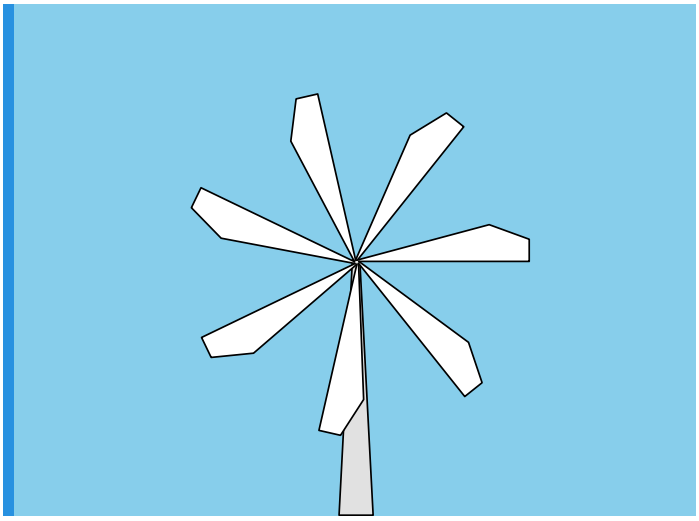
Since turbine blades are spread evenly in a circle, you will need to calculate the turn angle based on the number of blades chosen by the user.

Don't forget that the turtle should return to the correct orientation and rotation point between drawing each blade.

Here's what it should look like for 3 blades:

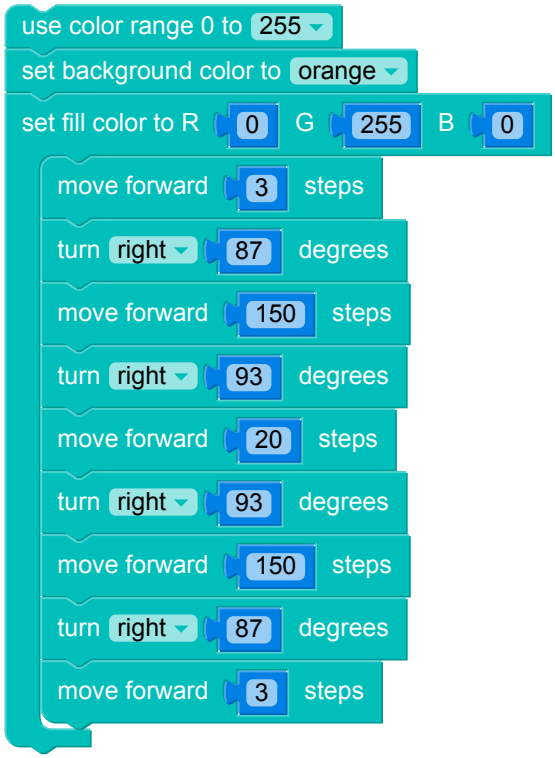


And for 7 blades:



You'll need

program.blockly



Testing

- [ ] Testing a wind turbine with 3 blades.
- [ ] Testing that there are no extra lines when drawing 3 blades.
- [ ] Testing a wind turbine with 7 blades.
- [ ] Testing a wind turbine with 4 blades.
- [ ] Testing a wind turbine with 6 blades.
- [ ] Testing a hidden case.
- [ ] Testing another hidden case.

7.2.5. Problem: Triangle wreath

A common sight on doors at Christmas is the Christmas Wreath (https://en.wikipedia.org/wiki/Wreath). You can use what you know about rotation to draw a simple wreath made up of triangles.

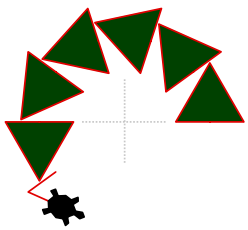
Using the turtle, make a wreath with the following properties:

- Each leaf of the wreath will be an equilateral triangle with side length 40.
- The colour of each leaf will be rgb(0, 65, 0).
- Each leaf will have an outline that is rgb(225, 0, 0).
- The wreath will have a radius of 50. The radius will extend to the **centre** of the base of each leaf triangle.
- The base of the first leaf will be on the horizontal the turtle starts, directly in front of its starting position.
- The program will ask how many leaves to draw, and will space them out evenly around the circle.

You should start your program by drawing one leaf, making sure that you position the pen correctly. You can then add a loop to draw the correct number of leaves as required by the user.

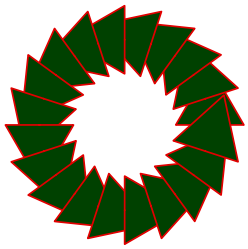
Here's what it should look like for 10 leaves:

```
Leaves: 10
```



And here's what a 20 leaf wreath looks like:

```
Leaves: 20
```



Hint

You will need to think carefully about the position of the turtle and the direction the turtle is facing when you start and end your loop.

Testing

- [ ] Testing the first example in the question, 10 leaves.
- [ ] Testing that there are no extra lines with 10 leaves.
- [ ] Testing a wreath with 20 leaves.
- [ ] Testing a wreath with 6 leaves.
- [ ] Testing a wreath with 17 leaves.
- [ ] Testing a hidden case.
- [ ] Testing another hidden case.

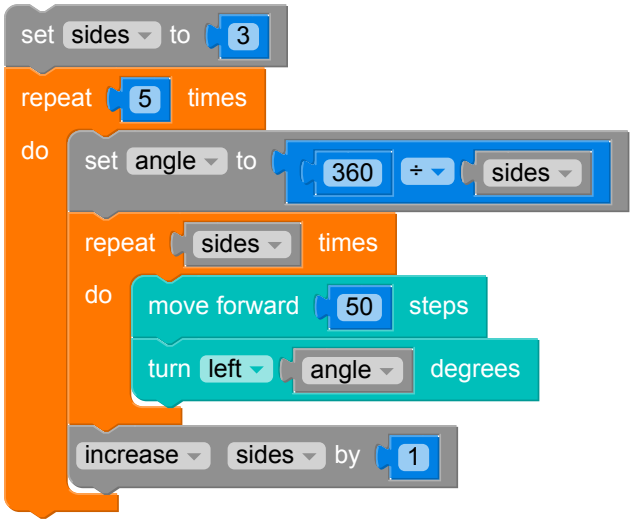
8

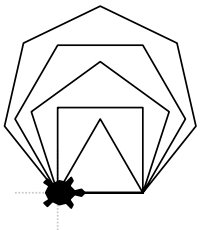
PROJECT 2: FIREWORKS DISPLAY

8.1. Changing things in loops

8.1.1. Loops can be different

Just because loops are used to repeat code, it doesn't mean that the code being repeated can't do different things. Run the code example below, and see what it creates:



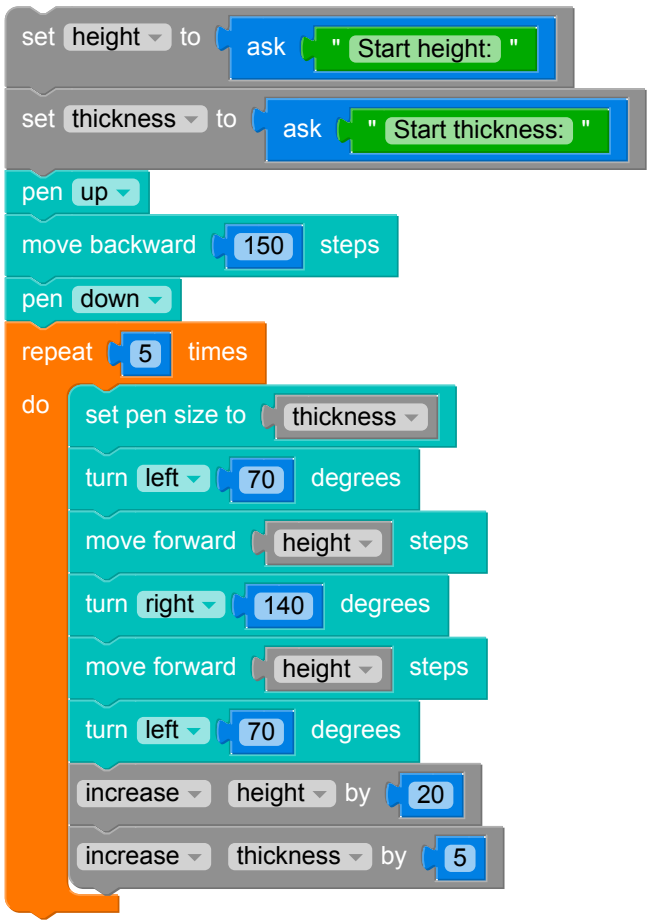


The key to having your repeated code doing different things each time is to make sure that you change some of the variables inside the loop. This way, the next time the code uses that variable, it will have a different value and can do different things!

8.1.2. Anything is variable

Any variable can be changed in the loop, which means any block that uses a variable can be updated to change what you draw. You can change more than one thing in the loop if you like, you just need to have more variables and make sure you change all of the right values in the loop.

This code asks the user for a start height and pen thickness, then draws a "mountain range" where each mountain gets higher and the pen used to draw it gets thicker each time through the loop. Run the code and try different values (30 for height and 2 for thickness are a good start).



By changing the thickness and height at the end of the loop, the next time through the variables are bigger, and the mountains get higher and thicker.

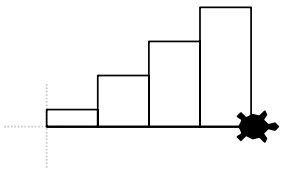
8.1.3. Problem: Staircase

Write a program that draws a staircase! You can think of a staircase as a series of individual rectangles, where each rectangle gets a bit bigger than the one before it.

The user will be asked for the height of the first step then the width of all steps. Your program should use those input values (in turtle steps) to draw the steps. After the first step, each subsequent step will be 20 turtle steps higher than the last one. It should draw a total of 4 steps.

The steps will go up and to the right of the screen, as shown in the example below.

```
Starting height: 10
Step width: 30
```



Make sure you test different heights and widths so you are confident your code is correct.

Testing

- [ ] Testing the bottom step from the first example in the question.
- [ ] Testing the whole first example from the question.
- [ ] Testing there are no extra lines in the first example from the question.
- [ ] Testing some really small steps.
- [ ] Testing some really big steps.
- [ ] Testing a hidden test case.
- [ ] Testing another hidden test case.

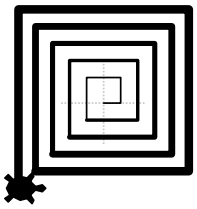
8.1.4. Problem: Thickening spiral

Spirals (https://en.wikipedia.org/wiki/Spiral) are common patterns in both the natural and man-made environment. Write a program that draws a spiral that gets thicker as it gets bigger.

Your program should ask the user to specify a starting size and thickness for the spiral, in that order. It should then draw the spiral, with the arms increasing in size by 5 turtle steps every time the turtle turns left. Each time the turtle completes one revolution and is facing to the right again, it should increase the thickness of the line by 1. It should do 5 complete revolutions.

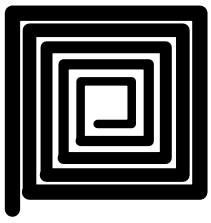
The spiral will start with the turtle moving to the right and turning left, as shown in the example below.

```
Starting size: 10
Starting thickness: 1
```



Here's another example, with a longer starting arm and thicker pen size:

```
Starting size: 20
Starting thickness: 5
```



Make sure you test different starting parameters so you are confident your code is correct.

Testing

- [ ] Testing the first example from the question.

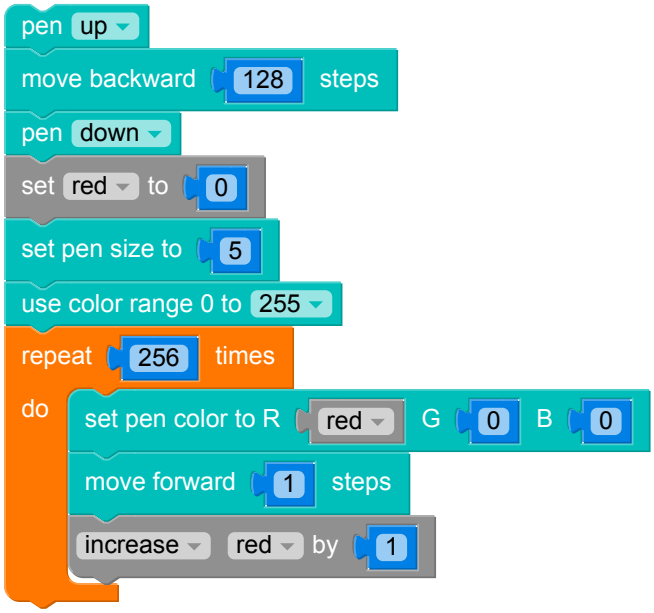
- [ ] Testing there are no extra lines in the first example from the question.
- [ ] Testing the second example from the question.
- [ ] Testing a spiral that starts out really small.
- [ ] Testing a spiral that starts out big.
- [ ] Testing a hidden test case.
- [ ] Testing another hidden test case.

8.2. Making gradients

8.2.1. Gradients are loops

A colour gradient (https://en.wikipedia.org/wiki/Color_gradient) is a gradual transition from one colour to another - a bit like a sunset that goes from orange through to blue. They are often used as a background on slides in a slide deck but are also common in user interfaces as colour selection tools.

Gradients can easily be created with the turtle by altering the RGB channels inside a loop. The example below creates a line that is a gradient from black through to red:



The turtle starts by using a pen colour of black - or rgb(0,0,0) - and each time it draws one pixel, it increases the value of the red channel by 1. The last pixel it draws is rgb(255,0,0) - or bright red. The pixels between gradually more from black to red as more red colour is added to them.

Be careful with your RGB values!

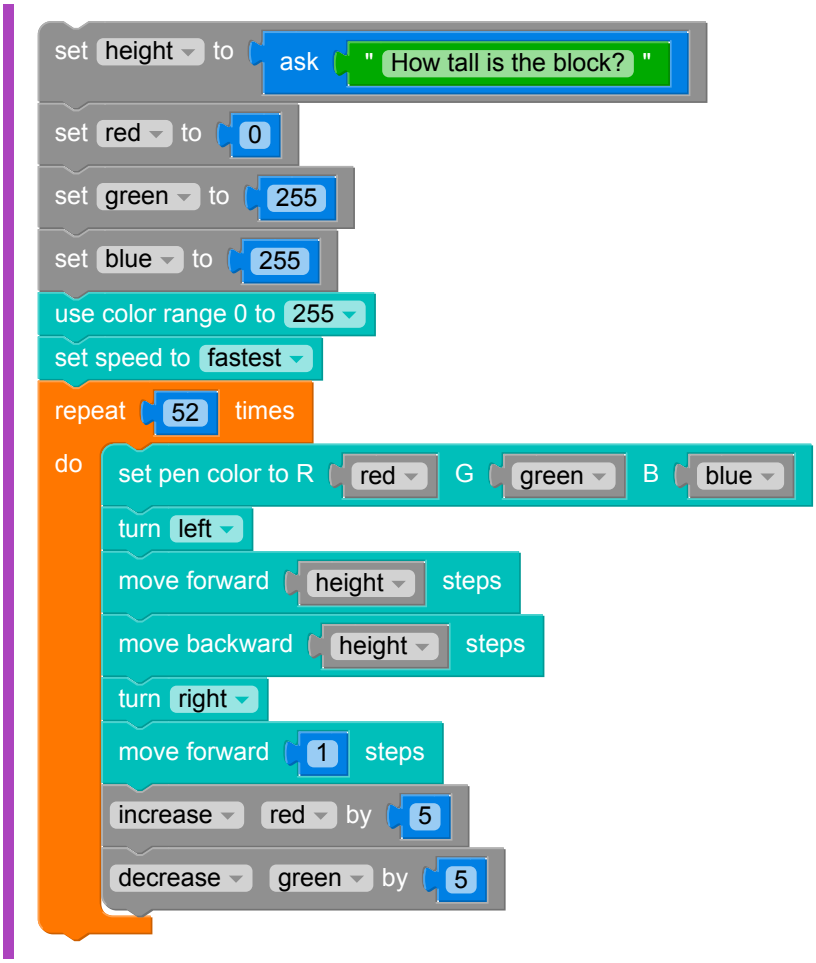
Since we are using `color range 0 to 255`, we need to make sure we don't try to use a value outside of this range in our `set pen color` block, otherwise our program will crash with an error.

8.2.2. RGB values in gradients

You can use a similar approach to create gradients through different colours. Here are the RGB values for some standard colours:

Colour	R	G	B
Black	0	0	0
Red	255	0	0
Green	0	255	0
Blue	0	0	255
Yellow	255	255	0
Magenta	255	0	255
Cyan	0	255	255
White	255	255	255
Colour	R	G	B

In the code below, we create a gradient that goes from Cyan to Magenta, and rather than drawing a single line we create a block based on the height typed in by the user.



We've changed the rate at which our colour changes by increasing and decreasing the RGB values by 5 instead of 1 (and changing how many times our loop repeats), and we've also introduced the set speed block, since drawing these many pixels takes a long time!

8.2.3. Problem: Ring of fire

Using your newly discovered knowlege of gradients, write a program that draws a ring of fire that gradually changes from red to yellow.

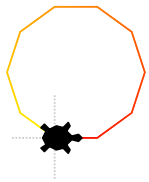
Your program will behave in the following way:

- Ask the user `"How many sides?"` the ring has - this can be any number from 3 through to 256.
- Each side of the ring will be 256 ÷ `sides` turtle steps in length.
- The pen colour will start out at rgb(255, 0, 0) and will finish at rgb(255, 255, 0). Each pixel the turtle draws will increase the green value of the colour by one.
- After drawing each pixel, the turtle will turn 360 ÷ `sides` degrees to the left.

The turtle starts out moving to the right from its starting location.

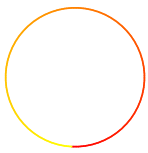
Here's an example with a ring of 10 sides:

```
How many sides? 10
```



And another example with 64 sides:

```
How many sides? 64
```



Speed things up!

Don't forget that you can use the set speed block to make the turtle move faster when drawing lots of pixels.

Testing

- [ ] Testing the first example in the question.
- [ ] Testing the first example with no extra lines.
- [ ] Testing the second example in the question.
- [ ] Testing the second example with no extra lines.
- [ ] Testing a ring with 3 sides.
- [ ] Testing a ring with 256 sides.
- [ ] Testing a hidden case.
- [ ] Testing another hidden case.

8.2.4. Problem: Baby you're a firework!

Let's create our first firework! Our first firework design will be an 8 pointed star that gradually transitions from magenta to cyan.

The dimensions of the firework are as follows:

- The pen size used by the turtle is 2 pixels;
- The colour starts at magenta, which is rgb(255,0,255);
- The colour finishes at cyan, which is rgb(0,255,255);
- The firework has 8 arms, meaning they are at an angle of 45° relative to each other;
- Each arm of the firework is 32 pixels long;
- The pen changes colour after the turtle moves 1 pixel;
- The colour channels red and green change at the rate of 1 per pixel moved by the turtle.

Here's what it will looks like, drawn at fastest speed:

Careful with that pen!

Make sure you don't draw over the top of your nicely created gradients...

Testing

- [ ] Testing that the firework starts out magenta.
- [ ] Testing the first arm of the firework.
- [ ] Testing that the firework finishes cyan.
- [ ] Testing the complete firework.
- [ ] Checking for no extra lines.

8.2.5. Problem: Firework spinner

If all of our fireworks looked the same then it would be a bit boring. There are lots of different types of fireworks, one of which is a spinner that can be attached to a fence or post while it spins sending sparks flying in all directions.

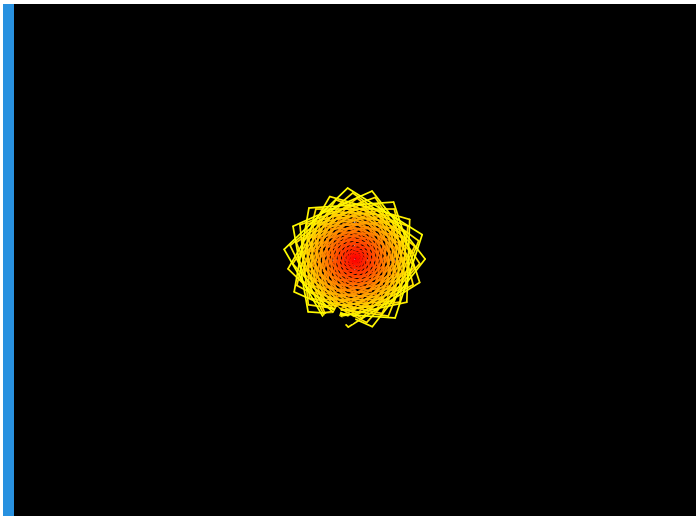
In this problem you'll draw a spinner firework design, and make it so that each time you run the program you can generate a different result!

Your program should ask the user to specify an angle, then draw a firework according to the following requirements:

- The background colour of the scene is rgb(0,0,0);
- The pen size used by the turtle is 1 pixel;
- The colour starts at red, which is rgb(255,0,0);
- The colour finishes at yellow, which is rgb(255,255,0);
- The turtle moves 256 times;
- The distance of the turtle's first movement is 1 step in the forward direction (i.e. to the right)
- After each step, the turtle turns left by the angle typed in by the user;
- The pen changes colour after each movement of the turtle;
- The green colour channel of the pen changes at the rate of 1 per movement of the turtle;
- Each time the turtle moves after the first time, the distance it travels increases by 0.2 turtle steps.

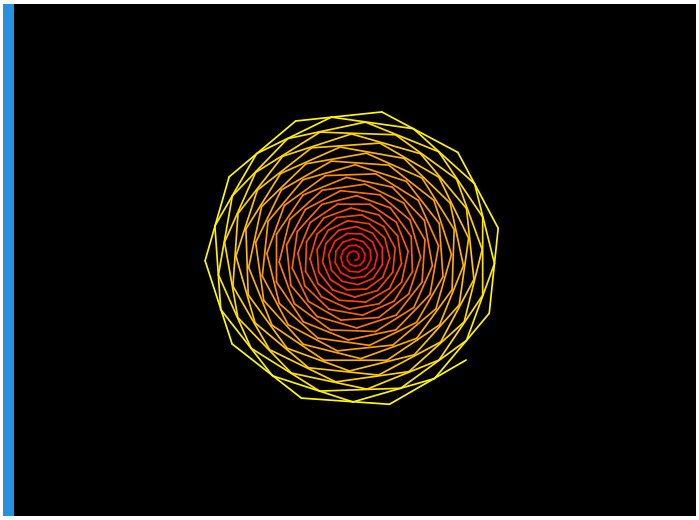
Here's what it will looks like, drawn at fastest speed:

```
Angle: 76
```



And here's what it looks like when it turns at a different angle:

```
Angle: 34
```



Testing

- [ ] Testing the first example in the question.
- [ ] Testing the first example for extra lines.
- [ ] Testing the second example in the question.
- [ ] Testing the second example for extra lines.
- [ ] Testing an angle of 113 degrees.
- [ ] Testing an angle of 90 degrees.
- [ ] Testing a big angle.
- [ ] Testing a hidden case.
- [ ] Testing another hidden case.

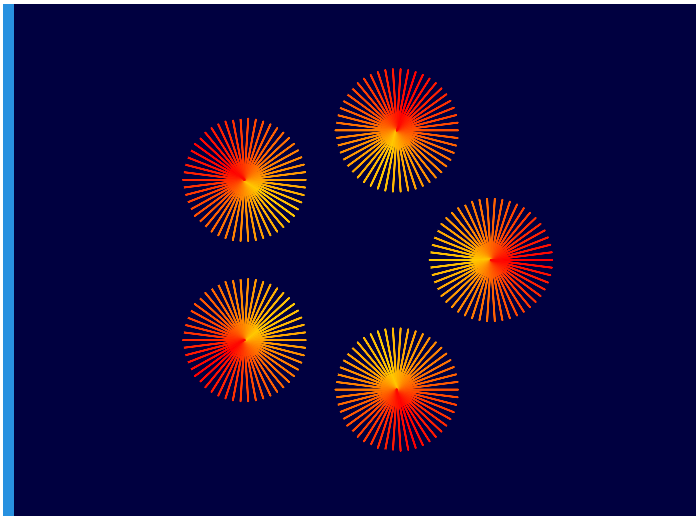
8.3. Fireworks Display

8.3.1. Problem: Fireworks display

The final problem for this project module is to put together everything you've learned up to now to create a fireworks display! In this problem you'll match the fireworks scene shown below.

You'll be able to generate different fireworks displays that vary based on the number of fireworks. The example below shows what the fireworks display will look like when you choose 5 fireworks:

```
Number of fireworks: 5
```



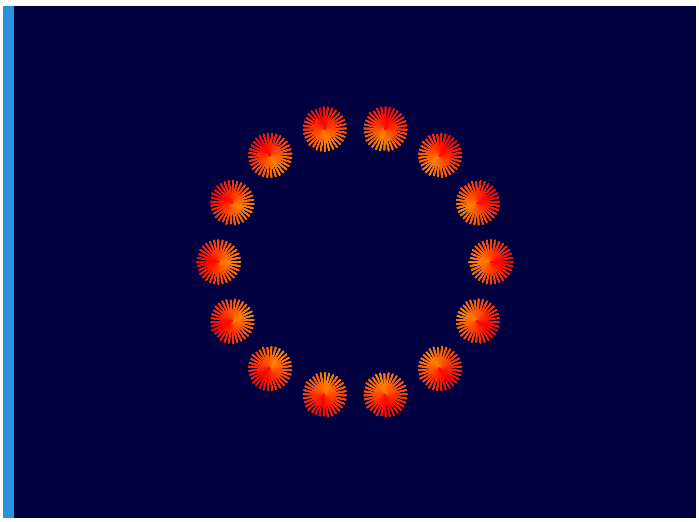
To get you started, we've provided some template code that includes some setup steps for the input and the variables you will need in your program. You don't need to change any of the existing code - you just need to fill in the bits that are missing inside the loop.

Here is a complete description of the details you need to build this scene:

- The background colour is rgb(0,0,64).
- The program will ask the user to enter `"Number of fireworks:"`.
- The number of spokes on each firework is (60 - (fireworks × 2)).
- The angle between each firework is 360 ÷ fireworks (so they end up in a circle).
- The angle between each firework spoke is 360 ÷ spokes (so the firework is complete).
- The length of each firework spoke is 180 ÷ fireworks (so they shrink as you have more fireworks).
- The centre of each firework is 80 turtle steps from the turtle's starting location.
- Each firework draws its first spoke red - rgb(255,0,0).
- After each spoke, the turtle turns left the calculated spoke angle.
- For the first half of the spokes in the firework, the amount of green increases by 8 after each spoke.
- For the second half of the spokes in the firework, the amount of green decreases by 8 after each spoke.
- After completing a firework, the turtle turns left the calculated firework angle.

And another example, but this time with 14 fireworks:

```
Number of fireworks: 14
```

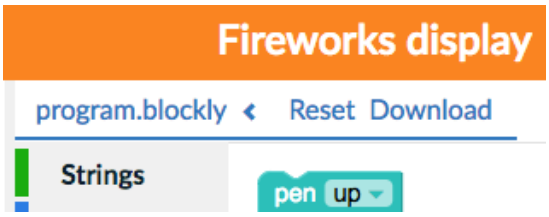


Hint

A good strategy might be to get your code working without the gradient first - start by drawing the fireworks correctly, then add the colours at the end.

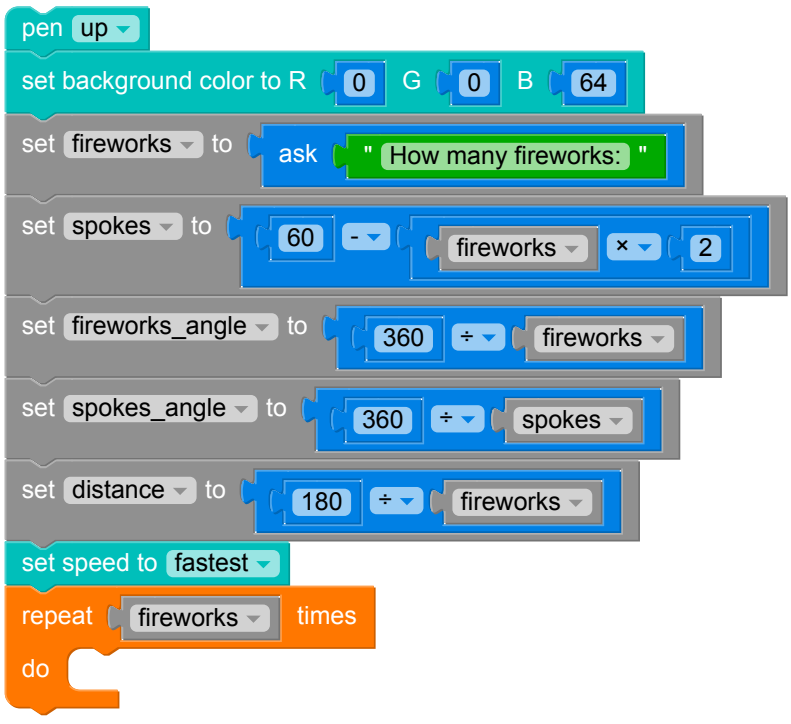
Resetting your code

If your code ends up in a state where you can't fix it anymore, you can always reset your code to the original template provided by using the Reset option in the code editor. You can find this by clicking on the little arrow near the program.blockly filename.



You'll need

program.blockly



Testing

https://aca.edu.au/challenges/7-maths-blockly.html

- [ ] Testing the first example in the question.
- [ ] Testing the first example with no extra lines.
- [ ] Testing the second example in the question.
- [ ] Testing two fireworks.
- [ ] Testing twenty-five fireworks.
- [ ] Testing a hidden case.

8.3.2. Congratulations!

Excellent work on finishing the project, and the course! You've learnt how you can use code to solve a range of maths problems, and how maths helps computers determine things like positioning and colour on a screen.

We hope you enjoyed this course, and can't wait to see what else you create as you learn more through our other code challenges.

8.4. Extension: Randomness

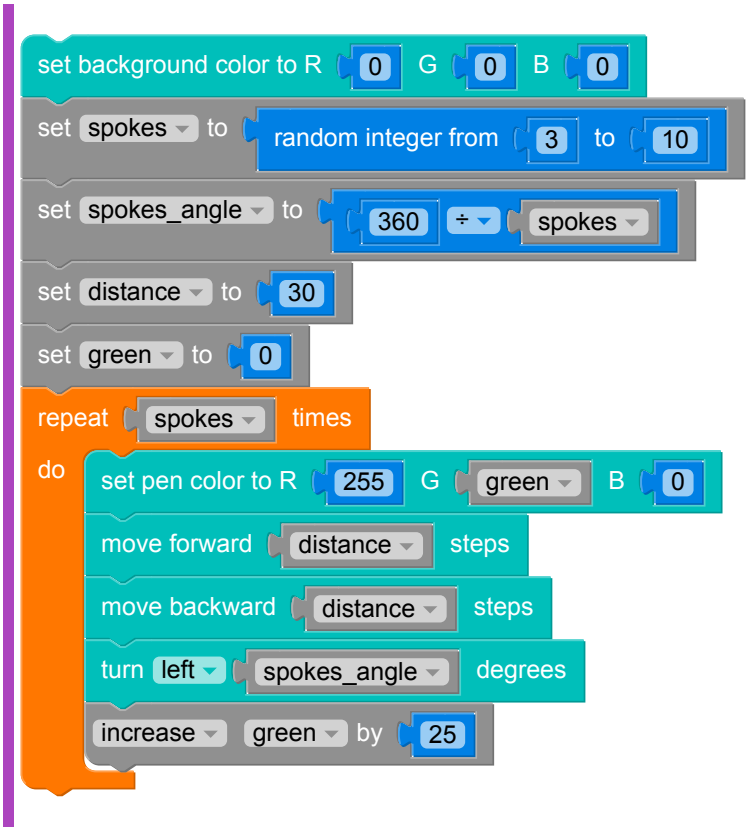
8.4.1. The random block

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

You may have finished the project, but if you're looking for something else to experiment with, we thought it would be fun to learn about the random block.

The random block allows you to have the computer generate a random number from a range of values. This is a fun way to make your program automatically change each time you run it - a fun way to make your fireworks different each time!

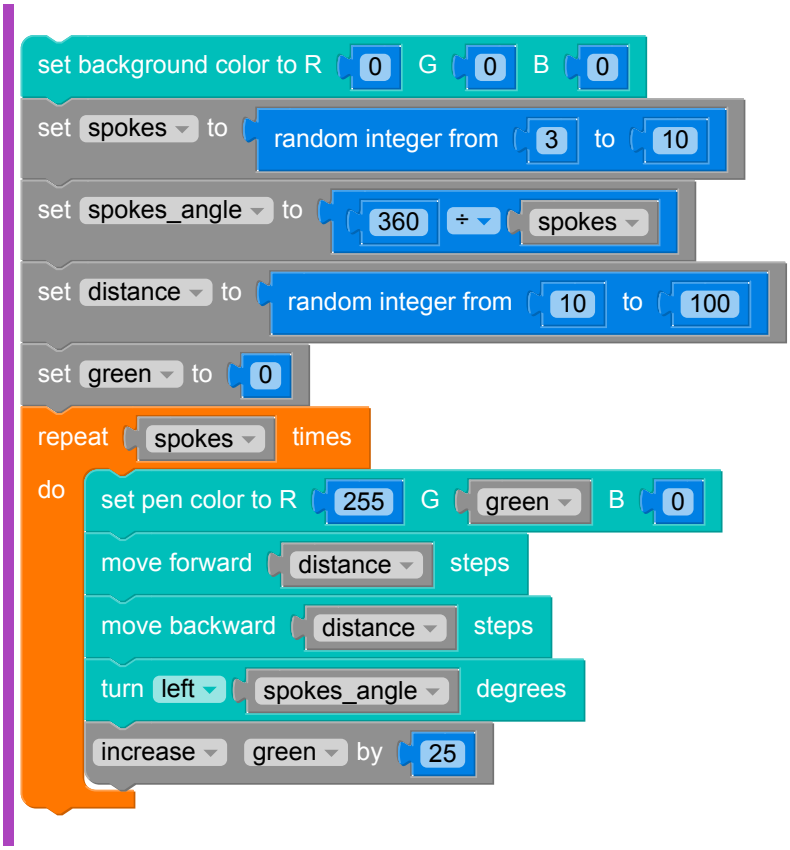
The example below draws a single firework, but the number of spokes on the firework changes to a random number between 3 and 10 each time the program is run.



Run the example program a few times to see how it works!

8.4.2. Lots of randomness

You can use the random block as many times as you like in your program. Here's the same program again, but this time we randomly generate the size of the firework too!



By adding a bit of randomness to your program, you could have fireworks in your fireworks display that have different numbers of spokes and can be a whole range of different sizes!

8.4.3. Problem: Fireworks playground

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