



DT Challenge Python  
**Turtle**

1. Introducing the Turtle
2. Angles with the Turtle
3. Looping with the Turtle
4. Add a Dash of Colour
5. Asking Questions
6. Making Decisions
7. Putting it all together!
8. Playground!



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

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

© Australian Government Department of Education and Training.

# 1

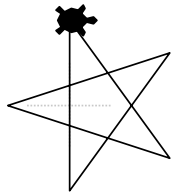
## INTRODUCING THE TURTLE

### 1.1. Turtle

---

#### 1.1.1. Introducing the turtle

Meet the [turtle](https://en.wikipedia.org/wiki/Turtle_graphics) ([https://en.wikipedia.org/wiki/Turtle\\_graphics](https://en.wikipedia.org/wiki/Turtle_graphics))!



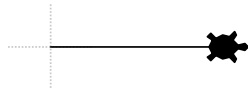
In this course, you'll be using the programming language Python to drive the turtle. It's fun, and what you'll learn is the basis for all [vector graphics](https://en.wikipedia.org/wiki/Vector_graphics) ([https://en.wikipedia.org/wiki/Vector\\_graphics](https://en.wikipedia.org/wiki/Vector_graphics)) in computers.

Your job is to write Python code to drive the turtle. What you'll learn is the basis for all [vector graphics](https://en.wikipedia.org/wiki/Vector_graphics) ([https://en.wikipedia.org/wiki/Vector\\_graphics](https://en.wikipedia.org/wiki/Vector_graphics)).

#### 1.1.2. Turtle Modules

`turtle` is a Python *module*. It's an extra part of the Python language, so we need to import its functions by putting an `import` statement at the top of each program:

```
from turtle import *  
forward(100)
```



The `*` means *everything*, so this imports all of the functions from the `turtle` module. Run this example to see what it does!

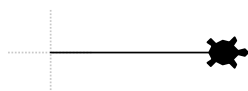
### 💡 Where is the turtle?

The turtle module is only enabled on turtle questions. If you want to start playing around, skip ahead to the next question before coming back to read the notes.

## 1.1.3. Move forward

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

```
from turtle import *  
forward(100)
```



When you run the Python 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.

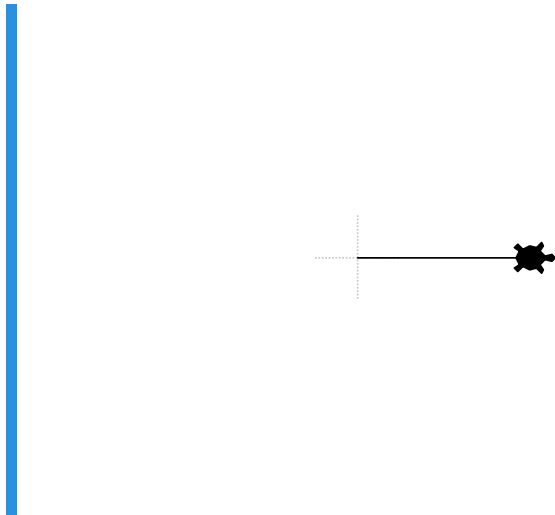
Try changing the `100` to `30` in the code snippet above and see how far the turtle moves.

## 1.1.4. Problem: Make a move!





Now it's your turn to write your own Turtle program! Write a program to make the turtle move **forward 100 turtle steps**.

It should look like this when you run it:



If you're not sure how to start writing the program, go back a few pages and take another look at the notes.

### 💡 How do I submit?

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

### You'll need

 `program.py`

```
from turtle import *
```

### Testing

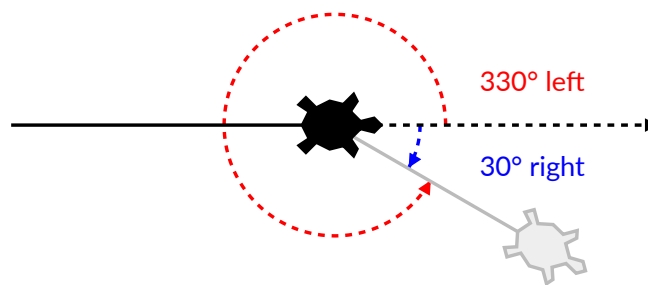
- Testing that there's a 100 turtle step line.
- Testing that the line is in the right place.
- Testing that there are no extra lines.
- Well done! You wrote your first turtle program. Keep moving!**

### 1.1.5. What do these ° signs mean?

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  $360^\circ \div 4 = 90^\circ$ .

This is called a *right angle* (don't confuse it with turning right!)

Turning half a circle (to face backwards) is  $360^\circ \div 2 = 180^\circ$ .

This is called a *straight angle* (it looks like a straight line).

### 1.1.6. Turning corners

The turtle always starts off facing to the right.

If you want to change which way the turtle is facing, you can turn `left` or turn `right`. These functions need the angle to turn in degrees. Here we're turning left by 90° (a right angle):

```
from turtle import *
left(90)
```



Now the turtle is facing the top of the screen.

If you turn a total of  $360^\circ$  then the turtle will end up facing the same way as how it started, because  $360^\circ$  is a full circle:

```
from turtle import *  
right(90)  
right(180)  
right(90)
```

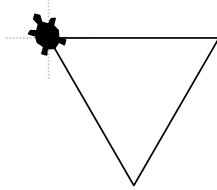


### 1.1.7. Drawing a shape

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

You can combine turtle instructions to draw shapes:

```
from turtle import *  
forward(100)  
right(120)  
forward(100)  
right(120)  
forward(100)
```



Here we've drawn a triangle with  $60^\circ$  angles, and 100 turtle steps on each side. Since the sides are equal length, we've drawn an [equilateral triangle](https://en.wikipedia.org/wiki/Equilateral_triangle). ([https://en.wikipedia.org/wiki/Equilateral\\_triangle](https://en.wikipedia.org/wiki/Equilateral_triangle))

([https://en.wikipedia.org/wiki/Equilateral\\_triangle](https://en.wikipedia.org/wiki/Equilateral_triangle))

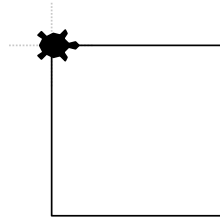


## 1.1.8. Problem: Starting from Square One



We've given you part of a program which tells the turtle to draw a square, your task is to finish it!

When it's finished, the turtle should draw a square like this:



All the sides of the square should be 100 turtle steps long.

### You'll need

 program.py

```
from turtle import *  
  
forward(100)  
right(90)  
forward(100)
```

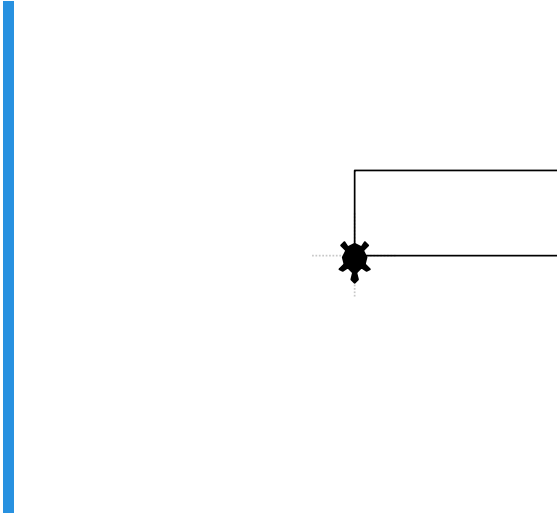
### Testing

- [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 another turtle problem!](#)

## 1.1.9. 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.

### You'll need

program.py

```
from turtle import *
```

### Testing

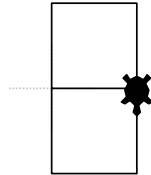
- [Testing the top of the rectangle.](#)
- [Testing the left side of the rectangle.](#)
- [Testing the right side of the rectangle.](#)
- [Testing the bottom of the rectangle.](#)
- [Testing the whole rectangle.](#)
- [Testing for no extra lines.](#)
- [Fantastic, you've solved this turtle problem!](#)

## 1.1.10. Problem: Domino



Write a program to draw a domino, made up of two squares on top of the other.

The turtle should start on the left in the middle of the domino:



Each side of each square should be 50 turtle steps long.

### You'll need

 program.py

```
from turtle import *
```

### Testing

- [Testing the middle of the domino.](#)
- [Testing the bottom of the domino.](#)
- [Testing the top of the domino.](#)
- [Testing the whole domino.](#)
- [Testing for no extra lines.](#)
- [Great work!](#)

[https://en.wikipedia.org/wiki/Equilateral\\_triangle](https://en.wikipedia.org/wiki/Equilateral_triangle)

[https://en.wikipedia.org/wiki/Equilateral\\_triangle](https://en.wikipedia.org/wiki/Equilateral_triangle)

# 2

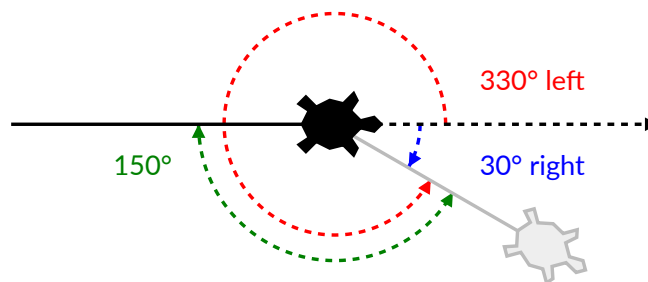
## ANGLES WITH THE TURTLE

### 2.1. Angles

#### 2.1.1. Spin me right round!

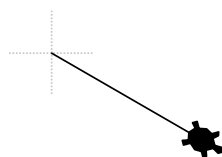
Remember that you can think of an *angle* as a *change of direction*. The angle between two lines is the turn you'd make to go from one line to the other. Here, the lines are the turtle's old and new directions.

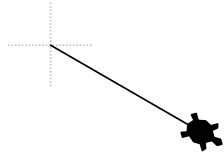
We've added a green *internal* angle between the two lines in our diagram. You can see the  $180^\circ - 150^\circ = 30^\circ$  turn you need:



If you get stuck with angle calculations, use the diagram!

```
from turtle import *
right(30)
forward(100)
```







## 2.1.2. Problem: Plate instead of bowl

Write a Turtle program to turn a very steep bowl shape into a plate.

The lines should all remain the length they are (i.e. 50 turtle steps, 100 turtle steps and 50 turtle steps), but you need to change the angles.

Calculate the size of the turns you need to make.

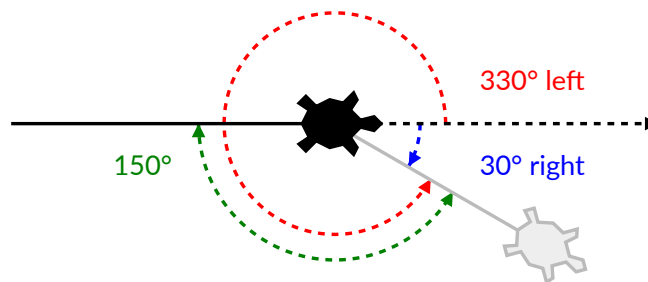
The angle between the plate side and the plate base is  $135^\circ$ .

The turtle will 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, and use the diagram below to help!



### You'll need



 program.py

```
from turtle import *  
  
right(90)  
forward(50)  
left(90)  
forward(100)  
left(90)  
forward(50)
```

### Testing

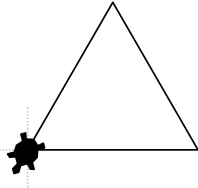
- [Testing the left side of the plate.](#)
- [Testing the base line of the plate.](#)
- [Testing the right side of the plate.](#)
- [Testing the whole plate.](#)
- [Testing for no extra lines.](#)

## 2.1.3. 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  $60^\circ$ .

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!

### You'll need

program.py

```
from turtle import *
```

### 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!](#)

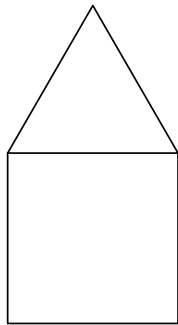
## 2.1.4. Problem: Get your house in order



[Use the turtle to draw a house!](#)

The triangle at the top should have angles that are all  $60^\circ$  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.

### **You'll need**

 program.py

```
from turtle import *
```

### **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!

# 3

## LOOPING WITH THE TURTLE

### 3.1. Loops with the turtle

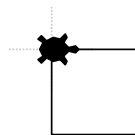
---

#### 3.1.1. Drawing shapes with loops

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

Drawing a square the long way:

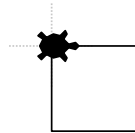
```
from turtle import *  
  
forward(50)  
right(90)  
forward(50)  
right(90)  
forward(50)  
right(90)  
forward(50)  
right(90)
```



We can use a loop to repeat certain instructions. Then, we don't have to repeat the same two instructions over and over again.

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

```
from turtle import *  
  
for count in range(4):  
    forward(50)  
    right(90)
```



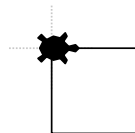
### 3.1.2. A closer look at for loops

[Let's look at what's happening here.](#)

[range\(4\)](#) is acting like a **repeat**, counting up four times – **sneakily**, starting at 0, and counting up 4 times, so 0, 1, 2 and 3.

[For each different value \(which is saved in \*\*count\*\* each time\), the \*\*body\*\* of the loop is run. That is, the \*\*forward\(50\)\*\* and \*\*right\(90\)\*\* commands are run. Then, the loop restarts!](#)

```
from turtle import *  
  
for count in range(4):  
    forward(50)  
    right(90)
```



[The \*\*body\*\* of the loop is all the lines that are \*indented\*, - that is, that start with two spaces in the example above.](#)

[Make sure all the instructions in your body are indented to the same amount! We recommend you try two spaces.](#)

### 3.1.3. Looping with numbers

[Just to see what's happening, we can print out some information to the screen. This program will print out Hello! to the screen four times:](#)

```
for count in range(4):
    print('Hello!')
```

```
Hello!
Hello!
Hello!
Hello!
```

[And this loop will print out the value that we're saving in count. See how it gets set to a different number each time, first 0, then 1, 2, and 3:](#)

```
for count in range(4):
    print(count)
```

```
0
1
2
3
```

[You don't have to call it count, either! You can call it anything you like!](#)

```
for side_number in range(4):
    print(side_number)
```

```
0
1
2
3
```

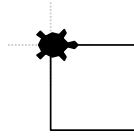
### 3.1.4. Looping and drawing

[Let's see how this works with the turtle!](#)

```
from turtle import *

for side_number in range(4):
    print('Now drawing side number: ')
    print(side_number)
    forward(50)
    right(90)
```

```
Now drawing side number:
0
Now drawing side number:
1
Now drawing side number:
2
Now drawing side number:
3
```

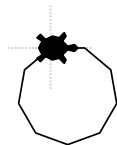


### 3.1.5. Looping lookouts!

[There are a few gotchas with loops to look out for!](#)

[Don't forget the colon! The for line has to end with a :, or else the loop won't work!](#)

```
from turtle import *  
  
for count in range(9):  
    forward(20)  
    right(40)
```



[Make sure all the instructions in your body are indented to the same amount! You can use two spaces, four spaces, or a tab, but they all have to be the same!](#)

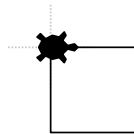
### 3.1.6. Problem: Back to Square One



Write a program to draw a square below the turtle.

Try using a `for` loop to make life easier!

Each side should be 50 turtle steps long.



#### Hint

Look for patterns that repeat themselves, and put those inside (*indented in*) the a `for` loop!

#### You'll need

`program.py`

```
from turtle import *
```

#### Testing

- Testing the top of the square.
- Testing the right side of the square.
- Testing the bottom of the square.
- Testing the left side of the square.
- Testing the whole square.
- Testing for no extra lines.
- Fantastic, you've drawn the *right* (pun intended) square!



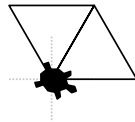
## 3.2. Loops and Movement

---

### 3.2.1. Repeat Block

We can use `for` loops to make some great patterns!

```
from turtle import *  
  
for count in range(6):  
    forward(50)  
    left(120)  
    forward(50)  
    left(120)  
    forward(50)  
    left(120)  
    left(60)
```



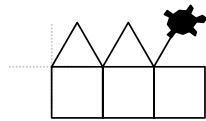


### 3.2.2. Problem: Row of terraces

[We've already drawn one house - now let's draw a whole row of houses!](#)

[We've given you the code to draw one house, and angled the turtle to the right spot for drawing the next house. Your task is to change the program so it draws 5 houses in a row!](#)

[The result should look like this:](#)



#### Hint

You only need to add one **for** loop and fix the indentation to solve this question!

#### You'll need

program.py

```
from turtle import *

forward(30).
right(90).
forward(30).
right(90).
forward(30).
right(90).
forward(30).
right(30).
forward(30).
right(120).
forward(30).
left(60).
```

#### Testing

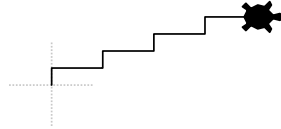
- [Testing the whole house.](#)
- [Testing for no extra lines.](#)
- [Well done! That's quite a row of houses!](#)

### 3.2.3. Problem: Staircase



Write a program that draws a staircase of 4 steps that are 10 steps high and 30 steps wide.

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



#### Hint

Think about what direction the turtle needs to face before you start drawing your steps.

#### You'll need

`program.py`

```
from turtle import *
```

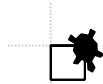
#### Testing

- Testing the bottom step.
- Testing the whole staircase.
- Testing there are no extra lines.

### 3.2.4. Squares on squares on squares

When we use a `for` loop, it repeats everything indented inside of it, the so-called *body* of the loop. Let's say we wanted to draw 6 squares. We could try something like this:

```
from turtle import *  
  
for count in range(6):  
    forward(20)  
    right(90)  
    forward(20)  
    right(90)  
    forward(20)  
    right(90)  
    forward(20)  
    right(90)
```

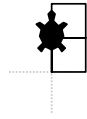


Six squares are drawn, but **on top of each other!** To draw more squares side by side, we'll need to add in an extra step to **move forward** between each square.

### 3.2.5. Moving between loops

Here, we've added a `left(90)` at the beginning, and a `forward(20)` at the end of the instructions within the `for` loop, so that each time the square is drawn in a new position, pointing the right way.

```
from turtle import *  
  
left(90)  
for count in range(6):  
    forward(20)  
    right(90)  
    forward(20)  
    right(90)  
    forward(20)  
    right(90)  
    forward(20)  
    right(90)  
    forward(20)
```

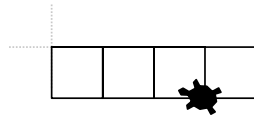


### 3.2.6. Problem: Blocks in a Row



[Let's draw a row of blocks!](#)

The row should be 6 blocks long, and each square should have sides **30 turtle steps** long.



#### You'll need

 `program.py`

`from turtle import *`

#### Testing

- [Testing the right side of the first square.](#)
- [Testing the left side of the first square.](#)
- [Testing the whole square.](#)
- [Testing for no extra lines.](#)
- [Great work!](#)

([https://en.wikipedia.org/wiki/Equilateral\\_triangle](https://en.wikipedia.org/wiki/Equilateral_triangle))

# 4

## ADD A DASH OF COLOUR

([https://en.wikipedia.org/wiki/Equilateral\\_triangle](https://en.wikipedia.org/wiki/Equilateral_triangle))

### 4.1. Turtle colours

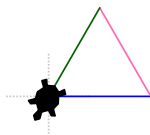
---

#### 4.1.1. Now with colour!

Let's add some colour! We can set the colour that we're drawing with using `pencolor`:

```
from turtle import *
pencolor('blue')

forward(60)
left(120)
pencolor('hotpink')
forward(60)
left(120)
pencolor('darkgreen')
forward(60)
```



#### **💡 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!

([https://en.wikipedia.org/wiki/Equilateral\\_triangle](https://en.wikipedia.org/wiki/Equilateral_triangle))

### 4.1.2. All the colours of the rainbow!

Try some other colour names! But **be careful**, you have to use a colour that the turtle knows about, and you must spell it correctly.

Here are some of our favourite colours:

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

([https://en.wikipedia.org/wiki/Equilateral\\_triangle](https://en.wikipedia.org/wiki/Equilateral_triangle))

You can see the whole list of colour names that Turtle understands

([https://en.wikipedia.org/wiki/Equilateral\\_triangle](https://en.wikipedia.org/wiki/Equilateral_triangle))here (<http://wiki.tcl.tk/37701>).



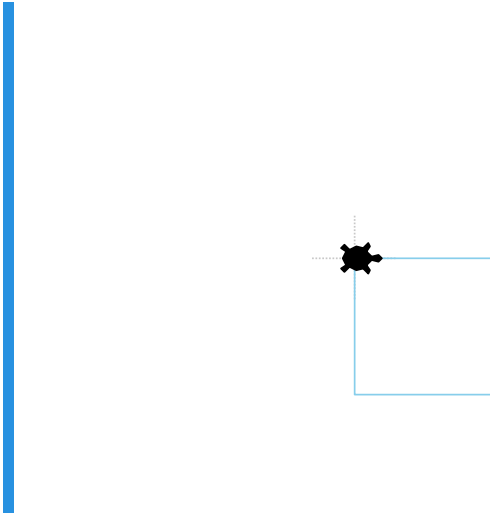
### 4.1.3. Problem: Skyblue Square



Let's add some colour to our square!

Write a program to draw a skyblue square using the turtle.

Each side should be **80 turtle steps** long.



#### You'll need

program.py

```
from turtle import *
```

#### Testing

- Testing the top of the square.
- Testing the right side of the square.
- Testing the bottom of the square.
- Testing the left side of the square.
- Testing the whole square.
- Testing for no extra lines.
- Great work! Blue skies (and boxes) from here!**

## 4.2. Turtle lines

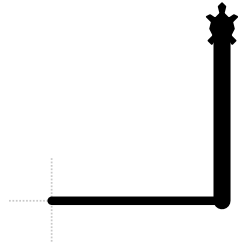
---

### 4.2.1. Drawing thicker lines

Those lines are looking nice and colourful, but they're a bit thin! We can change how thick the pen is using `pensize`.

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

```
from turtle import *  
  
pensize(5)  
forward(100)  
left(90)  
  
pensize(10)  
forward(100)
```



Try it out with different sizes!

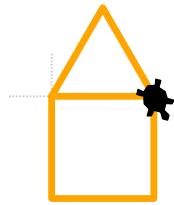
## 4.2.2. Problem: Build a sturdy house



The house we drew earlier looks a bit flimsy. Let's draw one that looks more sturdy!

Draw an **orange** house with walls of **pen size 4!**

Each side should be **60 turtle steps** long.



### You'll need

program.py

```
from turtle import *
```

### Testing

- Testing the top of the square of the house.
- Testing the right side of the house.
- Testing the bottom of the house.
- Testing the left side of the house.
- Testing the left side of the roof.
- Testing the right side of the roof.
- Testing the whole house.
- Testing for no extra lines.
- Nice building work!**

## 4.3. Filled shapes with the turtle

---

### 4.3.1. Filling with colour

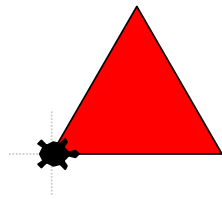
As well as changing the colour of lines, we can also fill shapes with colour.

To fill a shape, we need to tell the turtle when to **begin filling** and when to **end filling**.

We use `begin_fill` at the beginning of the shape we want to fill, and `end_fill` at the end.

We also need to set the `fillcolor` to the colour we want!

```
from turtle import *  
  
fillcolor('red')  
begin_fill()  
forward(100)  
left(120)  
forward(100)  
left(120)  
forward(100)  
left(120)  
end_fill()
```

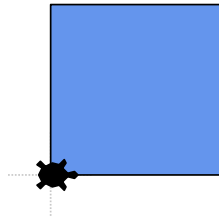


Try guessing what this program will draw before running the example! Then try changing the fill colour!

### 4.3.2. Fills and loops

We've already seen how drawing shapes is easier with loops, let's draw some shapes with both loops and fills!

```
from turtle import *  
  
fillcolor('cornflowerblue')  
begin_fill()  
for count in range(4):  
    forward(100)  
    left(90)  
end_fill()
```

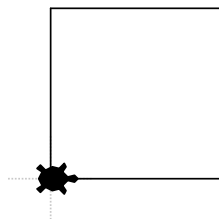


### 4.3.3. Filling flat out

The `begin_fill` and `end_fill` have to go **outside** the whole shape. You can't fill in a single line – there's nothing to fill in!

This example is trying to fill in each individual line, but not the whole completed shape:

```
from turtle import *  
  
fillcolor('orange')  
for count in range(4):  
    begin_fill()  
    forward(100)  
    left(90)  
    end_fill()
```



See how the shape isn't coloured in?

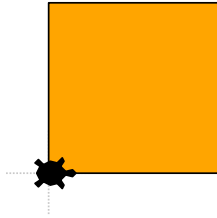
If we put the fill *inside* the loop, then the turtle tries to separately fill each side of the shape, which doesn't work!

### 4.3.4. Functional filling

The `begin_fill` and `end_fill` need to be wrapped around the whole shape you want to fill in.

In this case, that means putting them **outside** the `for` loop:

```
from turtle import *  
  
fillcolor('orange')  
begin_fill()  
for count in range(4):  
    forward(100)  
    left(90)  
end_fill()
```



## 4.3.5. Problem: Square Pennant Flags

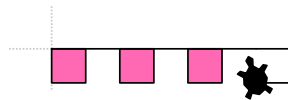


### Papel Picado

In Mexico decorations for parties and festivals often include [Papel picado](https://en.wikipedia.org/wiki/Papel_picado) ([https://en.wikipedia.org/wiki/Papel\\_picado](https://en.wikipedia.org/wiki/Papel_picado)). These are very colourful squares of paper which have beautiful cut-out designs.

You're going to make the turtle draw a string of 5 simple colourful Papel picado.

The **five** flags should be squares that are **20 turtle steps long** on each side, there will need to be an **extra line** to join the flags together, this will also be 20 turtle steps long. To make it super colourful you need to get the turtle to fill with **hot pink**.



### You'll need

program.py

```
from turtle import *
```

### Testing

- Testing the right side of the first flag.
- Testing the bottom of the first flag.
- Testing the left side of the first flag.
- Testing the drawing of all the outsides of the flags.
- Testing the fill colour.
- Testing for no extra lines.
- Time to party!**

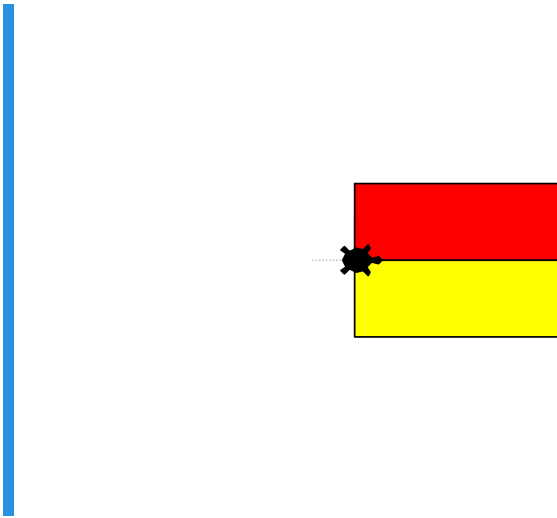
### 4.3.6. Problem: Swim between the flags!



In Australia it's very important when we go to the beach to swim between the flags. The flags are a very specific colour pattern to make them easy to recognise. They have a red rectangle on the top and a golden yellow colour on the bottom.

Get the turtle to draw a lifesaving flag. It should have one rectangle on the top that is **red** and one rectangle on the bottom that is **yellow**. The rectangles should be **120 turtle steps** long and **45 turtle steps** high.

The turtle should start on the left in the middle of the flag:



#### You'll need

program.py

```
from turtle import *
```

#### Testing

- Testing the middle of the flag.
- Testing the top of the flag.
- Testing the bottom of the flag.
- Testing the whole flag.
- Testing for no extra lines.
- Great work!



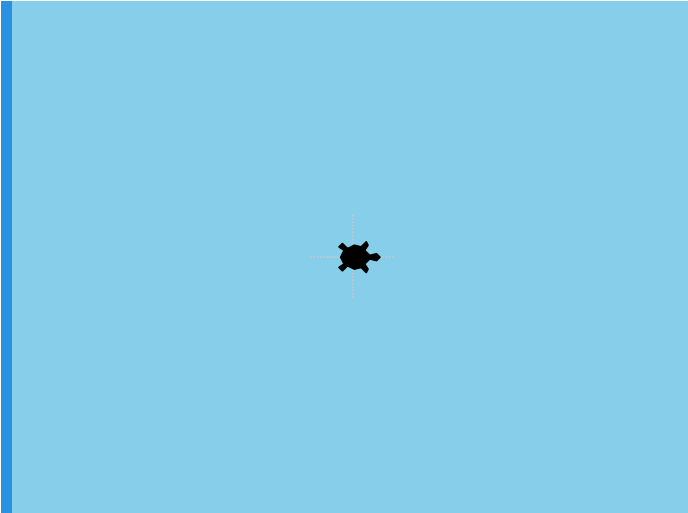
## 4.4. Advanced Turtle-fu!

---

### 4.4.1. Background colours

It's boring to always have a plain white background. That's why we have a way to set the `bgcolor` - short for **background color**.

```
from turtle import *  
bgcolor('skyblue')
```



Any of the colour names we've seen before will work here. (You can see the full list [here](http://wiki.tcl.tk/37701) (<http://wiki.tcl.tk/37701>)).

## 4.4.2. Problem: Fallout Shelter



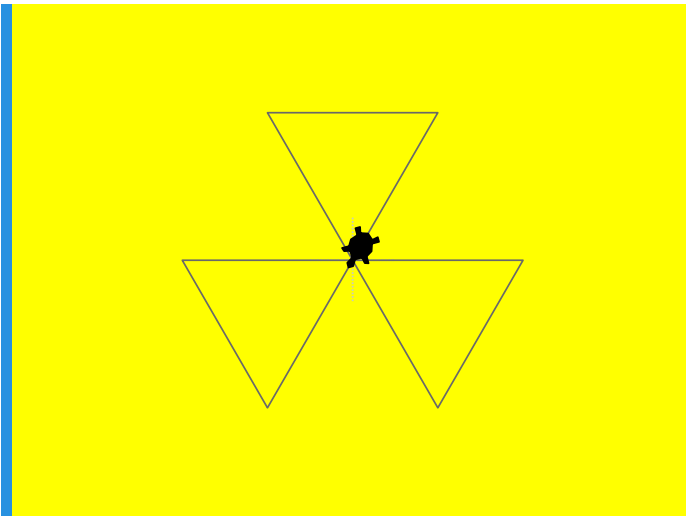
The symbol for a fallout shelter is three **black** triangles on a **yellow** background.

Let's draw this symbol with the turtle!

Set the **pencolor** to **dimgrey** so the edges stand out.

Each side of each triangle is **100 turtle steps** long and the triangles have all angles of **60 degrees**. The angle between each triangle is also **60 degrees**.

The output of your program should look like this:



The *centre* of the symbol is where the turtle starts.

### Hint

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

### You'll need

[program.py](#)

```
from turtle import *
```

### Testing

- Testing the outside of the first triangle.
- Testing the outside of the second triangle.
- Testing the outside of the third triangle.
- Testing all three triangles.
- Testing the fill of the triangles.
- Testing for no extra lines.
- Fantastic, but better stay away from radioactive materials anyway!**

# 5

## ASKING QUESTIONS

### 5.1. Variables and input

---

#### 5.1.1. Python the Calculator

If there's one thing computers are really good at, it's working with numbers. They can do billions of calculations per second!

In fact, Python is much better than a calculator, because it has variables!

We can use variables to store numbers for later, and we can do calculations on them, too!

Python can do all the operations you expect from a calculator:

Name	Calculator	Python
add	+	+
subtract	-	-
multiply	×	*
divide	÷	/

Python uses / for division because ÷ isn't a key on most keyboards!

#### 5.1.2. Remembering things

When drawing our square, we did the exact same thing 4 times: draw a line, and turn right. But if we made a typo and one side was longer than the others, it wouldn't make a square!

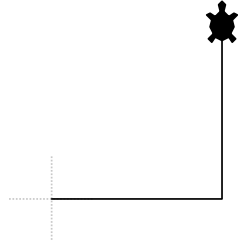
We need a **better way to remember and reuse** things like the side length.

We can use a *variable*! Each variable has a *name* which we use to set and get our value:

```
from turtle import *

line_length = 100

forward(line_length)
left(90)
forward(line_length)
```



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

Change the value of `line_length` to something else, and see what happens!

### 5.1.3. Reusing variables

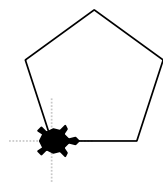
Once we've got values saved in variables, it's really easy to change how things work in our programs.

```
from turtle import *

sides = 5
side_length = 50

ext_angle = 360/sides

for i in range(sides):
    forward(side_length)
    left(ext_angle)
```



Try changing the value saved in the `sides` variable to a different number!

### 5.1.4. Asking Questions

We can use `input` to get the computer to ask a question. We might want to know what the user's favourite colour is. We could ask like this:

```
fave_colour = input("What is your favourite colour? ")  
print("You answered: " + fave_colour)
```

Try running this a few times and saying different colours!

## 5.1.5. Problem: Coloured Cards

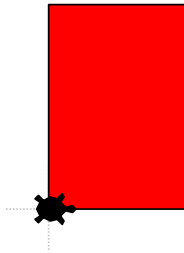


In many sports, coloured cards are used to let players know if a penalty has been awarded. For example, yellow cards are often a warning, and red cards send a player off.

Write a program which asks the user what colour card should be displayed, and draws it. The card should be **80 Turtle steps** wide, and **120 Turtle steps** tall.

We'll only test your program using valid `turtle` colours.

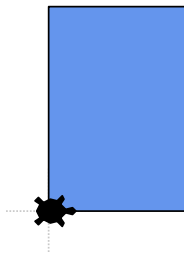
`Colour: red`



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

Here's an example with an unusual colour.

`Colour: cornflowerblue`



### Hint

We will only test your code with colours that Turtle understands!

### You'll need

`program.py`

```
from turtle import *
```

<https://aca.edu.au/challenges/78-python-turtle.html>

## 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 fill colour of the square.
- 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 yellow card.
- Testing a hotpink card.
- Testing a hidden test case.

## 5.1.6. Numbers and strings are different things!

Computers understand information in a different way to how people understand information.

Humans can easily understand that the spoken word "dog", a picture of a dog, or the letters *d*, *o* and *g* all represent the same thing. Computers don't see these similarities - they only see differences.

This is especially important when dealing with numbers!

If we ask a question with the computer using `input`, the answer always comes back as a *word* - what we call a **string** (short for *string of characters*).

If we want to use the answer that the user tells us as a number, we need to make sure Python understands it as a *number*.

We can do this using `int`:

```
from turtle import *
side_length = input("How long should the square sides be? ")

side_length = int(side_length)

for i in range(4):
    forward(side_length)
    right(90)
```

## 5.1.7. Asking the user for information

The `turtle` functions, like `forward`, only work with numbers. So if we ask the user for information using `input`, if it's a number, we need to make sure we use `int` to save it as a number.

```
from turtle import *
distance = input('How far? ')
forward(distance)
```

Because we're calling `forward` with `'50'` (a string), instead of `50` (an integer), it complains with a **TypeError**:

```
How far? 50
Traceback (most recent call last):
  File "program.py", line 3, in <module>
    forward(distance)
  File "/tmp/tmpbgmiSJ/turtle.py", line 632, in forward
    raise TypeError('Expected integer or float argument')
TypeError: Expected integer or float argument
```

Wow, this looks scary, *but it isn't!* Look at the last line. It says the **TypeError** occurred in our call to `forward` on **line 3** of `program.py` (our code). The `forward` function is defined in `turtle.py` (the `turtle` module file).

Notice that **where the error is detected isn't where the mistake was made** (the missing `int`). This is very common in programming.

We need to convert the string into an integer, like this:

```
from turtle import *
distance = int(input('How far? '))
forward(distance)
```

and now it works, because `forward` is given an integer, not a string.



### 5.1.8. Using user input

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

Now that we can ask the user for information using `input`, let's ask them how high our top hat should be:

```
from turtle import *

hat_height = int(input('What height should the hat be? '))
forward(30)
left(90)
forward(hat_height)
right(90)
forward(30)
right(90)
forward(hat_height)
left(90)
forward(30)
```



## 5.1.9. Problem: Purple square

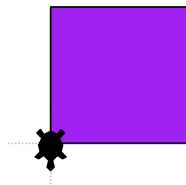
Write a program which asks the user what size square to draw, then draws it and fills it with purple.

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

We've started you off with some code that will read in the user's answer, and convert it to an integer using `int`.

The square should be filled with the colour `'purple'`.

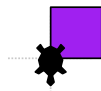
Side length: 80



The bottom left corner should be in the centre 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?

### You'll need

 program.py

```
from turtle import *
side_length = int(input('Side length: '))
```

### 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 fill colour of the square.
- 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.



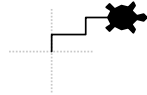
## 5.1.10. Problem: Step up

Write a program that draws steps. How many steps? That's the question!

Your program should ask the user how many steps to draw, and then draw them. Each step should be **10 Turtle steps tall**, and **20 Turtle steps wide**.

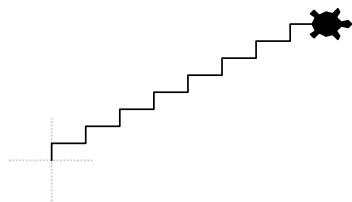
The steps will go up and to the right of the screen, as shown in the example below.

Steps: 2



Here's another example.

Steps: 8



### 💡 Hint

How many steps you draw depends on the number you read in and save in a variable!

Don't forget to save that number as an **integer**!

### You'll need

program.py

```
from turtle import *
```

## 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 a small set of steps.
- Testing a big set of steps.
- Testing a hidden test case.
- Testing another hidden test case.

# 6

## MAKING DECISIONS

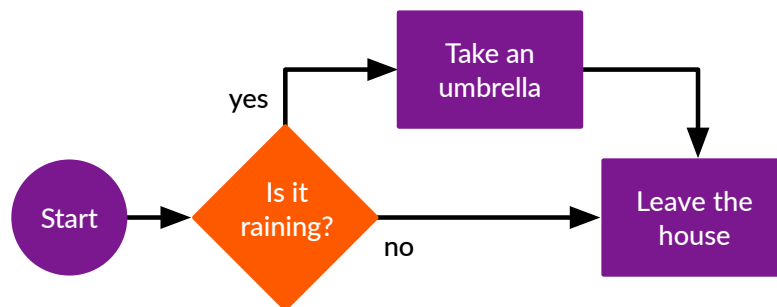
### 6.1. Making decisions

#### 6.1.1. Why do we need decisions?

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

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

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



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

We can write this in Python using an **if** statement.

#### 6.1.2. What if it is raining?

Let's write our flowchart as a Python program:

```

raining = input('Is it raining (yes/no)? ')
if raining == 'yes':
    print('Take an umbrella.')
print('Leave the house.')
  
```

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

Notice that the first **print** is *indented* (by two spaces). The **print** instruction is used to display characters and text on the screen.

This indented **print** instruction is the *body* of the **if** statement. The body must be indented.

If the value stored in **raining** is equal to **'yes'** (because the user entered **yes**), then the body is run. Otherwise, it is skipped.

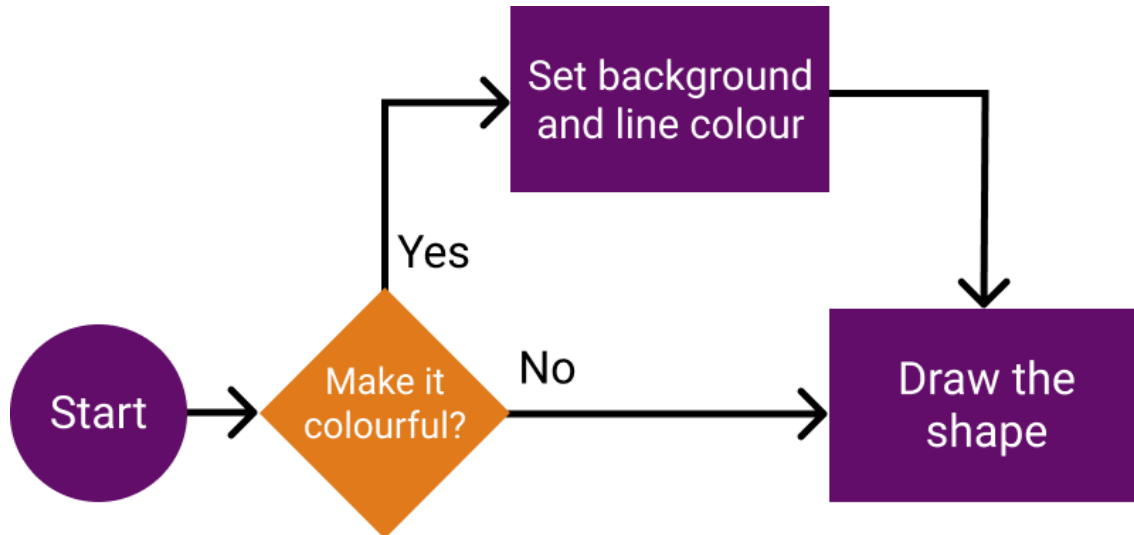
The second `print` always runs, because it is not indented, and isn't controlled by the `if` statement.

💡 **An `if` statement is a control structure**

The `if` statement *controls* how the program runs by deciding if the body is run or not.

### 6.1.3. Decisions with the Turtle

We can use `if` to make decisions with the Turtle, too!



Flowchart for making a colourful image with the turtle

This might look something like this in code:

```

from turtle import *
colourful = input('Make it colourful? ')
if colourful == 'yes':
    bgcolor('yellow')
    pencolor('darkgreen')
forward(50)
  
```

### 6.1.4. Controlling a block of code

An `if` statement can control more than one statement in its body.

These statements must have the same indentation, like this:

```

from turtle import *
colourful = input('Make it colourful? ')
if colourful == 'yes':
    bgcolor('yellow')
    pencolor('darkgreen')
    pensize(5)
forward(50)
  
```

If the user types in "yes" then all the indented lines (the *block*) will be executed first, then continue on with the rest of the program.

If it's anything else, those lines will be skipped and the next not-indented line (`forward(50)`) will be executed.

💡 **Careful with spaces!**

The number of spaces of indent must be to the same depth for every statement in the block. This example is broken because the indentation of the two lines is different:

```
food = input('What food do you like? ')
if food == 'cake':
    print('Wow, I love cake too!')
    print('Did I tell you I like cake?')
```

You can fix it by making both `print` statements indented by the same number of spaces (usually 2 or 4).

### 6.1.5. Assignment vs. comparison

You will notice in our examples that we are using two equals signs to check whether the variable is equal to a particular value:

```
colourful = input("Draw a background? ")
if colourful == 'yes':
    bgcolor('yellow')
```

This can seem quite confusing for beginner programmers.

A single `=` is used for *assignment*. This is what we do to set variables. The first line of the program above is setting the variable `colourful` to the value `"yes"` using a single equals sign.

A double `==` is used for *comparison*. This is what we do to check whether two things are equal. The second line of the program above is checking whether the variable `colourful` is equal to `"yes"` using a double equals sign.

If you do accidentally mix these up, Python will help by giving you a `SyntaxError`. For example, try running this program:

```
colourful = input("Draw a background? ")
if colourful = "yes":
    bgcolor("yellow")
```

```
File "program.py", line 2
    if colourful = "yes":
        ^
```

```
SyntaxError: invalid syntax
```

The second line only has one equals sign where it should have two.



## 6.1.6. Problem: Studybot - Highlighter or Pen



You're building a virtual robot to help you study. You've got it reading books, - now for highlighting and underlining notes!

Write a program that asks the user `Use a highlighter?` and either highlights or underlines for **100 Turtle steps**.

If the user types in `yes`, you should make the pen `"yellow"`, and pen thickness **15**.

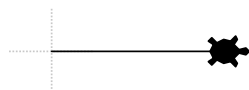
`Use a highlighter? yes`



Otherwise, no matter what the user types in, you should draw a black line 100 Turtle steps long.

Here's how the program should work if the user types in `no`:

`Use a highlighter? no`



### You'll need

`program.py`

```
from turtle import *
```

### Testing

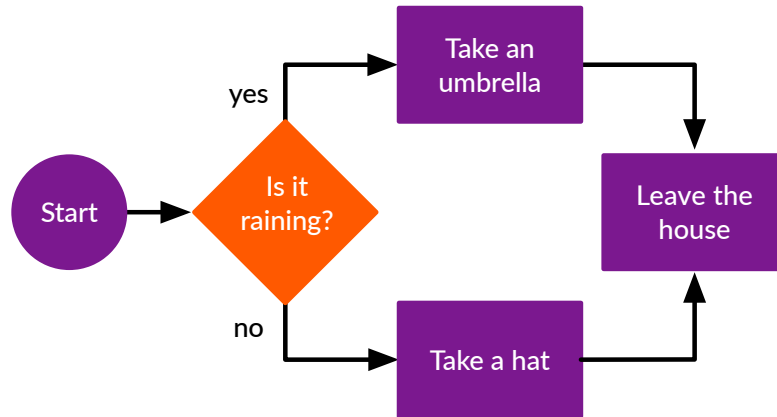
- Testing the highlighter.
- Testing the highlighter has no extra lines.
- Testing the pen.
- Testing the pen has no extra lines.
- Nice work! Now get that robot studying!

## 6.2. Decisions with two options

### 6.2.1. Decisions with two options

`if` statements allow you to make *yes* or *no* decisions. In Python these are called `True` and `False`.

Sometimes, we want an extra part to the `if` statement which is only run when a *condition* is `False`.



If the user says *yes* it is raining, then the program should say to take an umbrella, but otherwise it should say to put on a hat.

### 6.2.2. If it isn't raining...

In Python the `else` keyword specifies the steps to follow if the condition is `False` (in this case if it's *not* raining).

If the user says *yes* it is raining then the first *block* is executed, otherwise, the second *block* is executed instead:

```

raining = input('Is it raining? ')
if raining == 'yes':
    print('Pick up an umbrella.')
else:
    print('Put on a hat.')
  
```

The program will **never** say to pick up an umbrella *and* put on a hat!

**💡 Don't forget the colon!**

Notice the `else` keyword must be followed by a `:` character, just like the `if` statement.

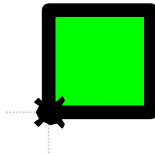
## 6.2.3. Problem: Traffic light



You're designing a traffic light system for robots! It's a square that is **60 Turtle steps** on each side.

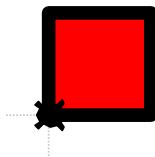
Write a program that reads in whether it is safe to go. If it is safe, you should draw a square with **green fill** and lines of thickness **8**:

Is it safe to go? yes



If it's not safe, you should draw a **red** square with lines of thickness **8**:

Is it safe to go? no



### 💡 Hint

One way to solve this question is to use two variables! One for the user's answer, and one for the colour you should use to draw!

### You'll need

`program.py`

```
from turtle import *
```

### Testing

Testing the bottom line from the first example in the question.

<https://aca.edu.au/challenges/78-python-turtle.html>

- 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 fill colour of the square.
- 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.
- Celebrate! All signals are go!**

## 6.3. Decisions about numbers

### 6.3.1. How do we compare things?

So far we have only checked whether two things are equal. However, there are other ways to compare things (particularly when we are using numbers). We can use the following *comparison operators* in *if* statements:

Operation	Operator
equal to	==
not equal to	!=
less than	<
less than or equal to	<=
greater than	>
greater than or equal to	>=

You can use a *print* statement to test these operators in conditional expressions:

```
x = 3
print(x < 10)
True
```

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

```
x = 3
print(x > 10)
False
```

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

### 6.3.2. Experimenting with comparison

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

```
x = 5
print(x <= 10)
True
```

Any value of *x* up to and including 10 will result in *True*. Any value of *x* greater than 10 will result in *False*. The opposite is true for greater than or equal to (>=).

Another important operator is not equal to (!=):

```
x = 5
print(x != 10)
True
```

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

### 6.3.3. Making decisions with numbers

Now we can bring together everything we've learned in this section, and write programs that make decisions based on numerical input. The example below makes two decisions based on the value in `x`:

```
x = 3
if x <= 3:
    print('x is less than or equal to three')
else:
    print('x is greater than three')
x is less than or equal to three
```

Try assigning different values to `x` to change how the program executes.

We could do something similar, but with user input:

```
height = int(input('How many cm tall are you? '))
if height == 157:
    print('You are the same height as Kylie Minogue.')
else:
    print('You are not the same height as Kylie Minogue.')
```

Remember you have to convert input to an integer using the `int` function if you want to do numerical comparisons.

## 6.4. Making complex decisions

---

### 6.4.1. Interplanetary visitor

Sometimes, we want to make decisions with more than two options. We can use `elif` short for *else if* to compare multiple things in the same check:

```
planet = input('What planet are you from? ')
if planet == 'Earth':
    print('Hello Earthling friend.')
elif planet == 'Mars':
    print('Hello Martian friend.')
elif planet == 'Jupiter':
    print('Hello Jovian friend.')
elif planet == 'Pluto':
    print('Pluto is not a planet!')
else:
    print('I do not know your planet.')
```

You can add as many `elif` clauses as you like, to deal with different cases.



## 6.4.2. Problem: The Three Little Pigs



In the fairytale [The Three Little Pigs](https://en.wikipedia.org/wiki/The_Three_Little_Pigs) ([https://en.wikipedia.org/wiki/The\\_Three\\_Little\\_Pigs](https://en.wikipedia.org/wiki/The_Three_Little_Pigs)), the first pig builds a house out of straw, the second sticks, and the third bricks.

Write a program which asks the user whether they want to build using `straw`, `sticks`, or `bricks`, and then draws the house in the appropriate colour. A straw house should be `orange`, a stick house should be `black`, and a brick house should be `slategray`.

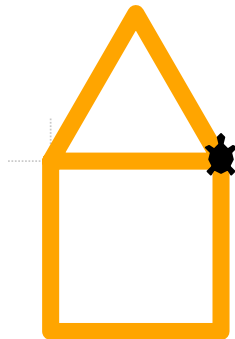
Material	Colour
<code>straw</code>	<code>orange</code>
<code>sticks</code>	<code>black</code>
<code>bricks</code>	<code>slategray</code>

The triangle at the top should have angles that are all 60°.

The triangle and square sides should all be **100 turtle steps** long and the `pensize` should be **10** for thick walls.

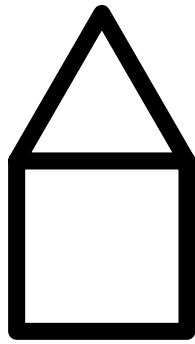
Here's a straw house:

```
Material: straw
```



Here's a stick house:

```
Material: sticks
```



And here's a brick house:

**Material:** bricks



### You'll need

 `program.py`

```
from turtle import *
```

### Testing

- Testing a house made of straw.
- Testing a house made of sticks.
- Testing a house made of bricks.



## PUTTING IT ALL TOGETHER!

### 7.1. Congratulations!

---

#### 7.1.1. Congratulations!

**Congratulations on making it through the course!** You've learnt so much!

We've put together a few extension questions and some advanced Turtle tips and tricks to challenge you even more. Have a go, and also try out your skills in our Turtle Playground question at the end where you can draw whatever you like!



## 7.1.2. Problem: Heartbeat

If you've ever seen a movie where someone is in a hospital bed, you would have seen their heartbeat appear as a line on a heart monitor. When a heartbeat is even, it repeats the same pattern over and over again like this:



Create a heartbeat pattern of 3 pulses that:

- Moves 20 steps forward before the start of the pulse;
- Turns 80° left to draw the start of the pulse;
- Moves 20 steps up to draw the start of the pulse;
- Turns 160° right at the top of the pulse;
- Moves 40 steps to draw the main part of the pulse;
- Turns 160° left at the bottom of the pulse;
- Moves another 20 steps to draw the last part of the pulse;
- Has a gap of 40 steps between each pulse (the hint will help with this).

### 💡 Hint

For this to work your turtle will need to face the same direction at the start and end of the loop, and the first and last thing it does is move forward 20 steps.

### You'll need

`program.py`

```
from turtle import *
```

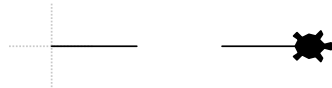
### Testing

- Testing the shape of the heartbeat.
- Testing the gap between each beat.
- Testing the whole heartbeat.
- Testing there are no extra lines.

### 7.1.3. Pen up and down

Sometimes you'll need to move the turtle around into the right position without drawing anything. The `penup` and `pendown` functions let you control this:

```
from turtle import *  
forward(50)  
penup()  
forward(50)  
pendown()  
forward(50)
```



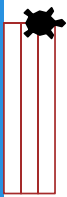
Imagine the turtle is holding a pen, after you've lifted up the pen with `penup` the pen is off the paper and the turtle won't draw anything as it moves around. After you've put the pen back down with `pendown`, the turtle will draw as it moves again.

## 7.1.4. 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



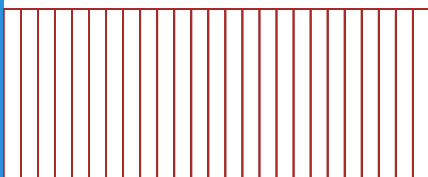
Your program should start off by moving 200 turtle steps to the left so that it starts off at the left-edge of the screen. Your program should work for any number up to 40 planks.

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



### You'll need

program.py

```
from turtle import *
```

## 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.



## 7.1.5. Problem: Scaleable House

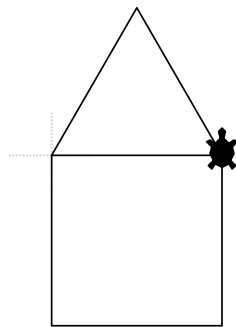
Let's draw a house for ants, or for giants!

We can take any shape and 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  $60^\circ$  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  $50 \times 2 = 100$  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





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

### You'll need

 `program.py`

`from turtle import *`

### 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  $10 \times 3$  turtle steps).
- Testing a house where the size is  $60 \times 1$  turtle steps.
- Testing a very very tiny house ( $3 \times 5$  turtle steps).
- Testing a hidden test case.
- Testing another hidden test case.



## PLAYGROUND!

### 8.1. Turtle Playground

---

#### 8.1.1. Playground Awaits!

This is the Playground Module. You can use these questions to draw whatever you'd like.

These questions aren't for points, just for playing, so have a go!

Make a work of art! Draw to your heart's content, and get those Turtles moving!

## 8.1.2. Problem: Turtle Playground!



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

### You'll need

 `program.py`

```
from turtle import *
```

### Testing

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

### 8.1.3. Problem: Turtle Playground!



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

#### You'll need

 `program.py`

```
from turtle import *
```

#### Testing

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

## 8.1.4. Problem: Turtle Playground!



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

### You'll need

 `program.py`

```
from turtle import *
```

### Testing

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