A workbook for young coders

# From Scratch to Python

Using Google Colab



This book belongs to:

_____

START HERE

# Hello, coder!

You already know how to code. Every time you snap Scratch blocks together, you are writing a program. You tell the computer what to do, step by step.

In this book you will learn **Python**. Python is a real programming language. Grown-up programmers use it to build websites, games, robots and even space missions. The great news is that Python uses the **same ideas** as Scratch:

- a **move 100 steps** block becomes `forward(100)`
- a **repeat 4** block becomes `for i in range(4):`
- a **say Hello!** block becomes `print("Hello!")`

The difference is that in Python you **type** the words instead of dragging blocks. That is the only big new thing. Everything else, like loops, variables and making your own blocks, you already understand.

## How to use this book

Go through the book in order. Each **mission** has five steps:

1. **Build it in Scratch.** Make the program in Scratch first, so you know what it should do.
2. **Type it in Colab.** Copy the Python code into Colab, letter by letter.
3. **Read it line by line.** Find out which Scratch block each line matches.
4. **Run it.** Press the play button and check that it worked.
5. **Make it yours.** Change the program and see what happens.

## Look out for these boxes

**Type this**

```
print("Hi!")
```

**Watch out!**

A mistake lots of people make, so you can avoid it.

**Did you notice?**

Something clever to spot in the code.

**Make it yours**

Ideas for changing the program. Tick each one you finish!

**Two-line puzzle**

A tiny puzzle at the end of each Part. The answer is two lines of Python.

Answers to the puzzles, and to the "Translate it yourself" pages, are at the back of the book. Try first, then check!

## Map of the book

The book has seven **Parts**. Each Part takes one family of Scratch blocks and shows you the Python for it. After the Parts come four special sections at the back.

| | Part | What you will do |
|---|---|---|
| 1 | **Motion and Pen** | Move the turtle and draw shapes. |
| 2 | **Loops** | Repeat things without typing them again. |
| 3 | **Variables and Lists** | Keep scores, names and lists of things. |
| 4 | **Operators** | Do maths, pick random numbers, join words. |
| 5 | **Sensing and Decisions** | Ask questions and make the program choose. |
| 6 | **My Blocks (functions)** | Make your own super-blocks. |
| 7 | **Looks, Sound and Events** | Shapes, colours, sounds and reacting. |
| | **Big Projects** | Bigger programs that use everything. |
| | **Bug Zoo** | Python error messages, and how to catch each bug. |
| | **Block Dictionary** | Look up any Scratch block and find its Python. |
| | **For grown-ups** | Notes for the person helping you. |

GETTING READY

# Meet Google Colab

**Google Colab** is a website where you can write Python and run it straight away. You don't have to install anything. Colab saves your work in Google Drive, so it will be there next time.

## Step 1: Open Colab

Ask a grown-up to open this website and sign in with a Google account:

`colab.research.google.com`

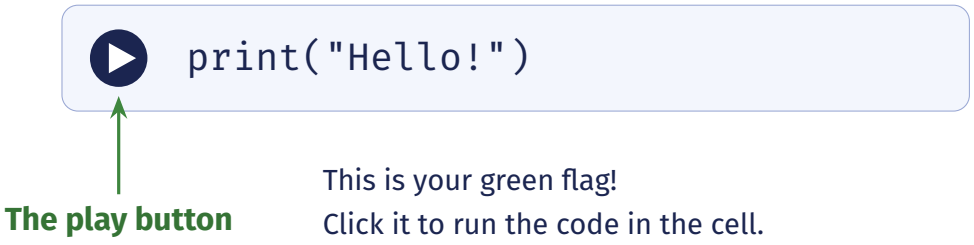
## Step 2: Make a new notebook

Click **New notebook**. A notebook is like a Scratch project: it is where your code lives.

At the top it says `Untitled0.ipynb`. Click on that name and change it to **My Python Adventures**.

## Step 3: Find the code cell

You will see a grey box with a round button on the left. This box is called a **code cell**. You type Python into it.



### Step 4: Say hello

Click inside the cell and type this exactly:

**Type this into a code cell**

```
print("Hello!")
```

Now click the play button . You can also hold **Shift** and press **Enter**. The first time, it may take a few seconds to start. Then you will see this underneath:

**You should see**

```
Hello!
```

**You just ran your first Python program!**

### Step 5: Add more cells

Click **+ Code** at the top to get a new empty cell. Use a new cell for each program. You can run any cell again whenever you like.

## The "In the beginning" cell: getting the turtle

In Scratch, you get the pen blocks by clicking the **Extensions** button in the bottom corner. Python works the same way: we add the drawing tools we need. In Python, the sprite that draws is called a **turtle**. It walks around the stage and leaves a line behind it, just like a Scratch sprite after **pen down**.

Each time you open your notebook, run this cell first. We call it the **"In the beginning" cell**. The Bible starts with "In the beginning God created the heavens and the earth" (Genesis 1:1), and every Colab session starts with this cell:

**The "In the beginning" cell: run it once every time you open Colab**

```
!pip install ColabTurtlePlus
```

The `!` at the start means: "this isn't Python. It is an instruction for Colab to download the turtle tools." Lots of text will scroll past. That is fine. Wait until it stops.

**Watch out: Plus!**

It must say **ColabTurtlePlus**, with **Plus** on the end. There is a different, older library called `ColabTurtle`. It has a black stage, a different size, and (0, 0) in the wrong corner, so none of the programs in this book would work with it.

## The three starting lines

Every program that draws starts with these three lines. They do the same job as this Scratch block:




**Start every drawing with these lines**

```
1 from ColabTurtlePlus.Turtle import *
2 clearscreen()
3 initializeTurtle((480, 360))
```

| Line | What it does |
|---|---|
| 1 | **Bring in the turtle tools.** This is like clicking Extensions and choosing Pen. The `*` means "bring in all of them". |
| 2 | **Clear the stage.** It wipes away the old drawing and the old turtles, so you can start fresh every time you run the cell. |
| 3 | **Make the stage 480 wide and 360 tall, and get the turtle ready.** That is exactly the same size as the Scratch stage! The two numbers sit in their own extra pair of brackets, `(480, 360)`, because the width and the height travel together as one thing. |

After these lines, you can just tell the turtle what to do. The turtle is like the **selected sprite** in Scratch: when you click a sprite, the blocks you snap together belong to it. In Python, you simply write `forward(100)`, and the turtle knows you mean it.

Try it! Type the three starting lines, then one more:

**Type this into a code cell**

```
1 from ColabTurtlePlus.Turtle import *
2 clearscreen()
3 initializeTurtle((480, 360))
4 forward(100)
```

The turtle starts in the middle of the stage, at (0, 0), facing **right**, just like a new Scratch sprite. When you run the cell, it walks 100 steps to the right and draws a line.

When you run the starting lines, Colab may also print a short note about `clearscreen()`. That is fine. It is just a reminder, and you can ignore it.

**If Colab gets stuck or forgets**

- **A cell keeps running forever?** Click the  button where the play button was. This is just like the red stop sign in Scratch.
- **You see `No module named 'ColabTurtlePlus'`?** Colab forgets everything when you close it or leave it alone for a while. Run the "In the beginning" cell again.
- **You see a `NameError`?** Python doesn't know a word yet, because a cell above has not run. Run the cells above it first, starting from the top. Or click **Runtime**, then **Run all**, to run every cell in order.

**Tip: switch on line numbers**

The line-by-line tables in this book talk about "line 3" and "line 4". Colab can number the lines in your cells too. Click **Tools**, then **Settings**, then **Editor**, and tick **Show line numbers**.

THE BIG PICTURE

# Scratch words, Python words

Here are the Scratch things you know, and what Python calls them.

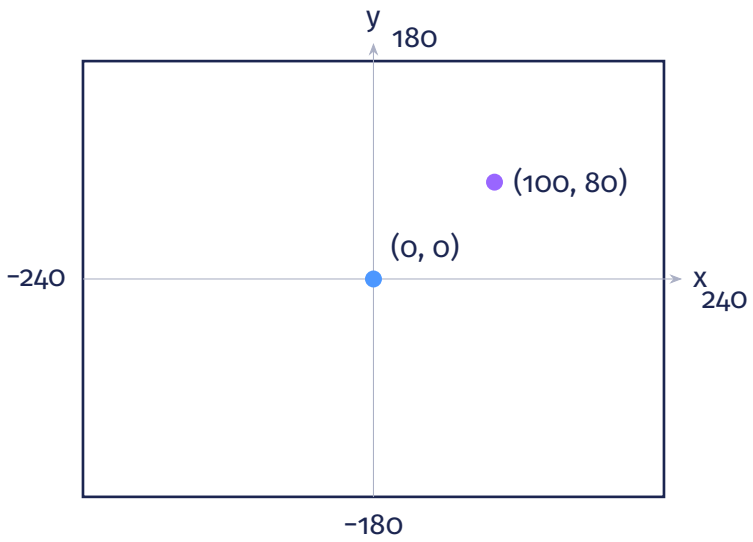
| In Scratch | In Python with Colab | Example |
|---|---|---|
| The selected sprite | The turtle | `forward(100)` |
| Another sprite | A new turtle with its own name | `cat = Turtle()` |
| The stage | The drawing area | `initializeTurtle((480, 360))` |
| The green flag | The play button  | or Shift + Enter |
| The stop sign | The stop button  | |
| A block | A line of code | `forward(10)` |
| Extensions button | `import` | `import random` |
| A white input bubble | Brackets ( ) | `right(90)` |
| Words in a bubble | Words in quote marks | `"Hello!"` |
| Make a Variable | Just give it a value | `score = 0` |
| My Blocks | A function | `def jump():` |

**More than one sprite**

Most programs only need the one turtle, so you just write `forward(100)`. When you want a second sprite, you make a new turtle and give it a name: `cat = Turtle()`. Then you tell that one what to do by putting its name and a dot first: `cat.forward(10)` means “cat, move 10 steps”.

## The stage is the same

Good news: the turtle's stage has the same coordinates as the Scratch stage. The middle is (0, 0). **x** goes from -240 on the left to 240 on the right. **y** goes from -180 at the bottom to 180 at the top.

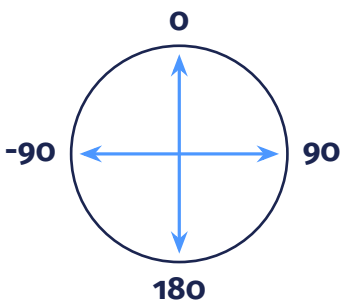



So **go to x: 100 y: 80** is `goto(100, 80)`, and the turtle lands on the purple dot, the very same spot as in Scratch.

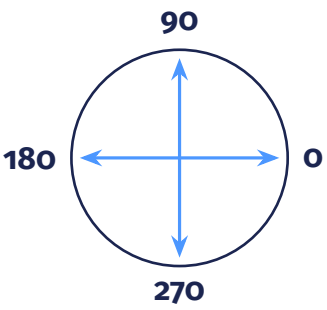
## But the compass is turned!

**Watch out: the compass is turned**

In Scratch, **point in direction 90** points to the **right**, and 0 points **up**.
In Python, `setheading(0)` points to the **right**, and 90 points **up**.




**Scratch**



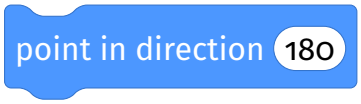

**Python**

**The secret rule**

To turn any Scratch direction into a Python direction, take it away from 90:

**Python direction = 90 – Scratch direction**

You don't even have to do the sum yourself. Python can do it for you!




```
setheading(90 - 180)
```

90 - 180 is -90. Python understands -90 too: it points **down**, the same as Scratch's 180.

| Scratch direction | Python | Points |
|---|---|---|
| 0 | `setheading(90 - 0)` | up |
| 90 | `setheading(90 - 90)` | right |
| 180 | `setheading(90 - 180)` | down |
| -90 | `setheading(90 - -90)` | left |

The good news is that **turning** works the same. **turn right 90 degrees** is `right(90)`, and **turn left 90 degrees** is `left(90)`. Most of the time you will use turning, not directions.

THE BIG PICTURE

# Four new habits

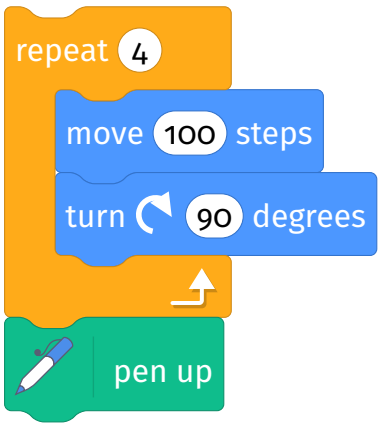
Python understands the same ideas as Scratch. But because you type instead of snapping blocks, there are four new habits to learn.

## Habit 1: Spaces show what is inside

In Scratch, a block goes **inside** a loop when you drop it into the C-shaped mouth.

In Python, two things do that job:

1. a **colon** : at the end of the line opens the mouth.
2. **four spaces** at the start of a line put it inside.




```
1 for␣i␣in␣range(4):
2 ␣␣␣␣forward(100)
3 ␣␣␣␣right(90)
4 penup()
```

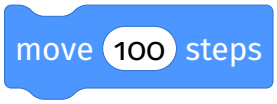
The little ␣ marks show the spaces, which are normally invisible. Lines 2 and 3 have four spaces, so they are **inside** the loop and repeat 4 times. Line 4 has no spaces, so it is **outside** the loop. It happens once, after the loop has finished.

**Colab helps you**

When you type a colon : and press Enter, Colab adds the four spaces for you. To come back out of the loop, press **Backspace** once.

## Habit 2: Spell it exactly

Blocks are already spelled for you. In Python, you do the spelling, and Python is very fussy. Capital letters, brackets and quote marks must all be just right.




```
1 forward(100)
```

| | |
|---|---|
| `forward(100)` | **Works!** |
| `Forward(100)` | **Error:** capital F. Python thinks `Forward` and `forward` are different words. |
| `foward(100)` | **Error:** missing r. Python can't guess what you meant. |
| `forward 100` | **Error:** no brackets around 100. |
| `forward(100` | **Error:** the closing bracket is missing. |
| `print("Hi)` | **Error:** the closing quote mark is missing. |

**Rule of pairs**

Brackets and quotes always come in pairs, just like socks! Every `(` needs a `)`, and every `"` needs another `"`.

## Habit 3: One line at a time, from the top

In Scratch, you can have lots of scripts that all start when you click the green flag. They all run at the same time.

Python is different. It reads your cell like a book: line 1, then line 2, then line 3, all the way to the bottom. It only jumps around when a loop or one of your own functions tells it to.

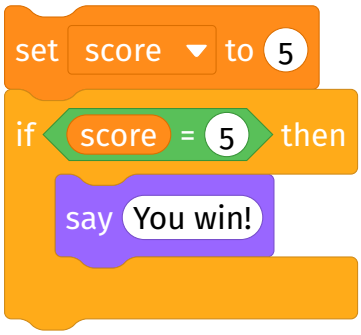



```
print("first")
print("second")
print("third")
```

## Habit 4: One = sets, two == checks

Scratch has two different blocks: one to **set** a variable and one to **check** whether it equals something. Python uses equals signs for both, so it needs a way to tell them apart:

- **One** equals sign = means “put this value in the box”. It is like .
- **Two** equals signs == mean “are these the same?”. It is like the green  block.




```
score = 5
if score == 5:
    print("You win!")
```

THE BIG PICTURE

# How Colab is different from Scratch

## Colab asks instead of listening

Colab can't hear key presses or mouse clicks. So instead of **when space key pressed**, your program **asks** you with `input()`, and you type your answer in a box under the cell.

**Type this into a code cell**

```
answer = input("Do you like cats? ")
if answer == "yes":
    print("Me too!")
```

Type `yes`, press Enter, and Python answers you. Later in the book you will meet a program that can hear keys, for real key-press games.

## One loop moves all the sprites

**Type this**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
cat = Turtle()
dog = Turtle()
dog.left(90)
for i in range(10):
    cat.forward(10)
    dog.forward(10)
```

Python does one line at a time, so **one loop moves every sprite** a little bit each time round.
Here the cat walks right and the dog walks up. In Scratch, this would be two scripts.

## Shapes and colours instead of costumes

There are no costume pictures in Colab, but the turtle can wear lots of **shapes**, like `"turtle"`, `"arrow"` or `"circle"`, in any colour you like: `shape("turtle")` and `color("gold")`. (Python spells colour `color`, with no u, and colour names use small letters.) And it can draw anything you can imagine, line by line. **You are ready. Let's go!**

TRACK YOUR PROGRESS

# My missions

Colour in a star each time you finish a mission: one star for making it work, two for doing one challenge, and three for doing them all!

| Mission | What you make | Done? |
|---|---|---|
| **Part 1: Motion and Pen** | | |
| 1 | A square the long way | ★ ★ ★ |
| 2 | Draw a house | ★ ★ ★ |
| 3 | Robot face | ★ ★ ★ |
| **Part 2: Loops** | | |
| 4 | Draw shapes the short way | ★ ★ ★ |
| 5 | Stars | ★ ★ ★ |
| 6 | Square spiral | ★ ★ ★ |
| 7 | Flower of squares | ★ ★ ★ |
| **Part 3: Variables and Lists** | | |
| 8 | Score counter | ★ ★ ★ |
| 9 | Growing spiral | ★ ★ ★ |
| 10 | Rainbow from a colour list | ★ ★ ★ |
| 11 | Random story maker | ★ ★ ★ |
| **Part 4: Operators** | | |
| 12 | Dice roller | ★ ★ ★ |
| 13 | Times-table printer | ★ ★ ★ |
| 14 | Random dot art | ★ ★ ★ |
| 15 | Name machine | ★ ★ ★ |
| **Part 5: Sensing and Decisions** | | |
| 16 | Say hello | ★ ★ ★ |
| 17 | Guess my number | ★ ★ ★ |
| 18 | Quiz with a score | ★ ★ ★ |
| 19 | Reaction timer | ★ ★ ★ |
| **Part 6: My Blocks** | | |
| 20 | Make your own block | ★ ★ ★ |
| 21 | Shape machine | ★ ★ ★ |
| 22 | Answer machine | ★ ★ ★ |

| Mission | What you make | Done? |
|---|---|---|
| 23 | Sprite army | ⋆ ⋆ ⋆ |
| **Part 7: Looks, Sound and Events** | | |
| 24 | Costume parade | ⋆ ⋆ ⋆ |
| 25 | Make music | ⋆ ⋆ ⋆ |
| 26 | Turtle race | ⋆ ⋆ ⋆ |
| 27 | Story with messages | ⋆ ⋆ ⋆ |
| **Big Projects** | | |
| 28 | Animate your name | ⋆ ⋆ ⋆ |
| 29 | Pen flower | ⋆ ⋆ ⋆ |
| 30 | Maths quiz | ⋆ ⋆ ⋆ |
| 31 | Guess the number deluxe | ⋆ ⋆ ⋆ |
| 32 | Choose-your-own story | ⋆ ⋆ ⋆ |
| 33 | Virtual pet | ⋆ ⋆ ⋆ |
| 34 | Chase game | ⋆ ⋆ ⋆ |
| 35 | Catch game | ⋆ ⋆ ⋆ |

PART 1

# Motion and Pen

Make the turtle walk, turn and draw, just like a sprite with the Pen extension.

In Scratch, the blue **Motion** blocks move your sprite around the stage, and the green **Pen** blocks let it draw. In Python, the **turtle** does both jobs. It is a sprite that always carries a pen.

In this part you will learn the Python for blocks like these:

**move 10 steps** **turn right 15 degrees** **go to x: 0 y: 0** **glide 1 secs to x: 0 y: 0** **point in direction 90** **change x by 10** **x position** **pen down** **pen up** **set pen color to** **set pen size to 1** **erase all** **stamp**

And you will learn some Python tricks that Scratch **can't** do, like circles, dots, filled-in shapes and writing words on the stage.

## Before you start

Every drawing program in this part begins with the **three starting lines**. They are like **when green flag clicked**. They bring in the turtle, wipe the stage and make it the same size as the Scratch stage.

**The three starting lines**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
```

On the concept pages below, the boxes show only the new lines. Type the three starting lines first, then the new lines underneath, and press ▶.

**Where does the turtle start?**

- In the middle of the stage, at $(0, 0)$, just like a new sprite.
- Pointing to the **right**, just like a new sprite (Scratch direction 90).
- With its pen **down**. This is different! A Scratch sprite only draws after **pen down**, but the turtle draws a line every time it moves.

## Move and turn




```
forward(100)
right(90)
forward(100)
left(45)
backward(50)
```

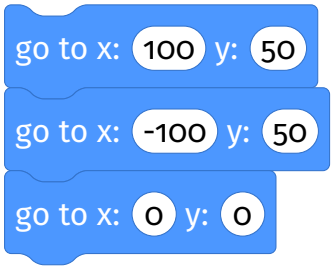
**How it works:** `forward(100)` is **move 100 steps**. The number goes inside the round brackets ( ), just like it goes inside the white bubble on the block. One step is one pixel in both, so your Scratch numbers work in Python too. To walk backwards, Scratch uses a minus number, but Python has its own word, `backward`.

`right(90)` turns clockwise, like the ↻ arrow, and `left(45)` turns anticlockwise, like the ↺ arrow. The good news: turning works **exactly** the same in Scratch and Python.

**Watch out: lowercase only**

Python is fussy about capital letters. `forward(100)` works, but `Forward(100)` gives an error. Every command in this part is written in lowercase letters.

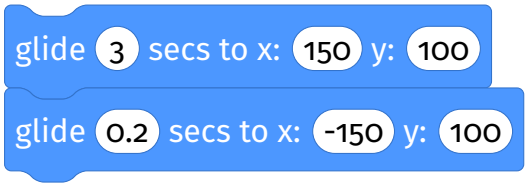
## Go to a spot




```
goto(100, 50)
goto(-100, 50)
goto(0, 0)
```

**How it works:** `goto(100, 50)` means "go to x: 100, y: 50". The x number always comes first, then a comma, then the y number. The stage is the same as in Scratch: $(0, 0)$ is the middle, x goes from $-240$ to $240$ and y goes from $-180$ to $180$. Because the pen is down, this program draws a triangle!

## Glide slowly



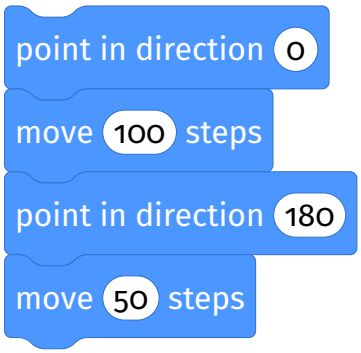

```
speed(1)
goto(150, 100)
speed(13)
goto(-150, 100)
```

**How it works:** The turtle always glides when it moves. You choose how fast with `speed()`. It takes a number from **1** (slowest, like a snail) to **13** (fastest, like a rocket). Python doesn't use seconds, so you pick the speed that looks right. The speed stays the same until you change it, so put `speed(13)` near the top when you want a big drawing to finish quickly.

**Watch out: don't use speed 0**

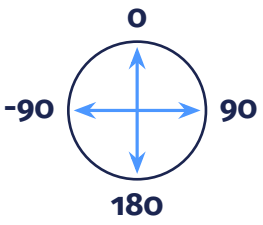
You might think `speed(0)` means "stand still". It doesn't! In Colab it means "don't show anything until the end", so your stage stays empty. Stick to 1 to 13.

## Point in a direction

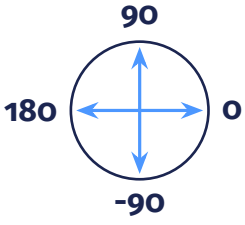



```
setheading(90 - 0)
forward(100)
setheading(90 - 180)
forward(50)
```

**How it works:** `setheading` means "point in direction". But Python's compass is turned round! In Scratch, 0 points **up** and the numbers go clockwise. In Python, 0 points **right** and the numbers go anticlockwise. Python can do sums inside the brackets, so we let it do the translating for us.




**Scratch**




**Python**

**The secret rule**

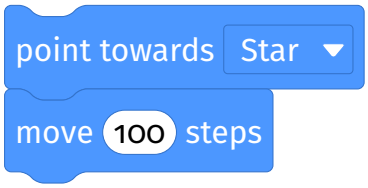
**Python direction = 90 − Scratch direction**

So **point in direction 30** becomes `setheading(90 - 30)`. Write the Scratch number after the `90 -` and Python works out the rest. You never have to do the sum in your head!

**Fill in the table.** Work out each Python direction. Then try it in Colab with `forward(100)` after it, and check the turtle goes the way you expected.

| Scratch direction | Python code | Python direction | Which way? |
|---|---|---|---|
| 90 | `setheading(90 - 90)` | 0 | right |
| 0 | `setheading(90 - 0)` | 90 | up |
| -90 | `setheading(90 - -90)` | _____ | _____ |
| 45 | `setheading(90 - 45)` | _____ | _____ |
| 135 | `setheading(90 - 135)` | _____ | _____ |

## Point towards something



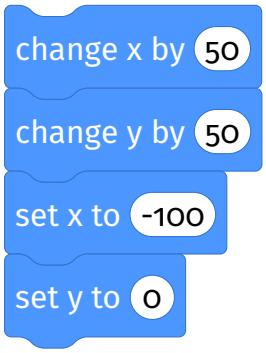

```
setheading(towards(100, 100))
forward(100)
```

**How it works:** In Scratch you point towards another sprite, like a Star sitting at x: 100, y: 100. In Python you point towards a **spot**. `towards(100, 100)` works out which direction the spot is in, and `setheading( )` turns the turtle to face it. One command sits inside the other, like a round block dropped into a white bubble.

**No 90 minus this time**

`towards` already gives the answer in Python's directions, so you don't need the secret rule here. You only need `90 -` when **you** type a Scratch direction number.

## Change x and y, set x and y



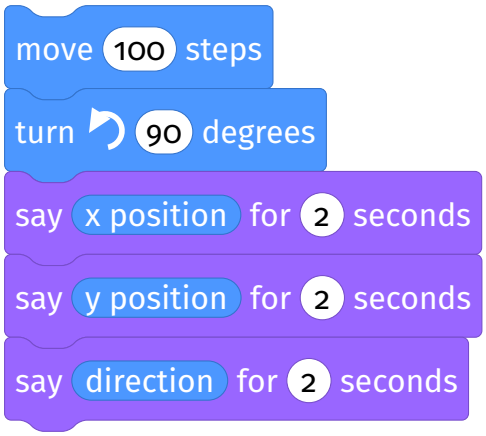

```
setx(xcor() + 50)
sety(ycor() + 50)
setx(-100)
sety(0)
```

**How it works:** `setx(-100)` is **set x to -100**, and `sety(0)` is **set y to 0**. Python has no "change x" command, so we build one. `xcor()` means "my x position". So `setx(xcor() + 50)` says: "set x to (my x position + 50)". That moves the turtle 50 steps to the right, which is exactly what **change x by 50** does!

**Watch out!**

`xcor()` needs its empty brackets. They tell Python "go and get the answer now". Without them, Python gets confused.

## Where am I? Which way am I facing?




```
forward(100)
left(90)
print(xcor())
print(ycor())
print(heading())
```

**How it works:** The round blocks **x position**, **y position** and **direction** become `xcor()`, `ycor()` and `heading()`. `print()` shows an answer underneath the cell, a bit like a sprite saying it.

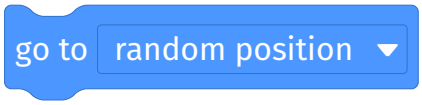
**You should see**

```
100.0
0.0
90
```

**Two things to spot**

- `100.0` is just 100 written with a decimal point. Python likes to show positions that way.
- Scratch would say the direction is **0**, because the sprite points up. Python says **90**, because `heading()` answers in **Python** directions. To get the Scratch number, use the secret rule again: `print(90 - heading())`.

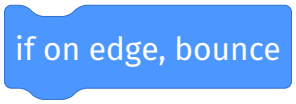
## Sneak peek: go to a random spot




```
import random
goto(random.randint(-240, 240),
     random.randint(-180, 180))
```

**How it works:** `import random` brings in Python's dice. `random.randint(-240, 240)` picks a whole number from $-240$ to $240$, so it is a surprise x. The second one picks a surprise y. The line was too long, so it carries on to the next line. Python allows that while a bracket is still open. You will meet `random` properly in Part 4.

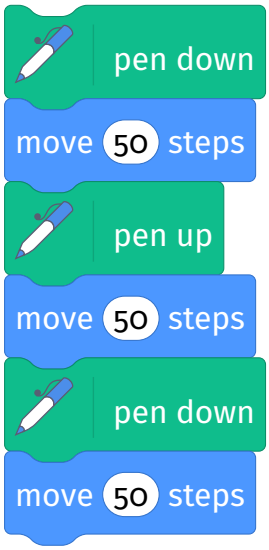
## If on edge, bounce




Python has no command for this one.

There is no wall around the turtle's stage. Try `forward(1000)`: the turtle walks right off the edge and keeps going! In **Part 5** you will build your own "if on edge, bounce" with an `if`. Then you will know the secret of what Scratch's block does.

## Pen up, pen down




```
forward(50)
penup()
forward(50)
pendown()
forward(50)
```

**How it works:** `penup()` lifts the pen off the paper, so the turtle moves without drawing. `pendown()` puts it back. The brackets are empty because these commands need no number, but Python still wants the brackets. They mean "do it now!". The turtle starts with its pen down, so the first **pen down** needs no line of Python.

### Moving without a line

To jump to a new place without drawing, use three lines: `penup()`, then `goto(x, y)`, then `pendown()`. You will use this trick a lot in the missions.

## Choose a pen colour




```
pencolor("red")
forward(100)
pencolor("#ff8800")
forward(100)
```

**How it works:** In Scratch you click a colour. In Python you type its name in **quote marks**, all in lowercase: `"red"`, `"gold"`, `"skyblue"`, `"hotpink"`. The quote marks tell Python "this is a word, not a command". There are 140 colour names to try!

For any colour at all, use a **hex code**, like `"#ff8800"` (orange). It is a `#` and six digits or letters. The first two say how much red, the middle two how much green and the last two how much blue. `00` is none and `ff` is the most.

**Watch out: American spelling**

In this book we write "colour", but Python spells it the American way: `pencolor`, with no u. If you type `pencolour`, Python won't know the word.

## Pen size and erase all

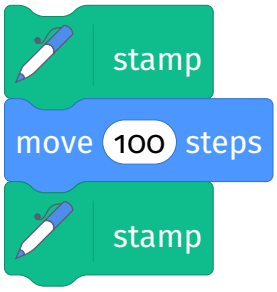



```
pensize(10)
forward(100)
clear()
```

**How it works:** A bigger number in `pensize` gives a fatter line. The turtle starts at size 1, just like Scratch. The size must be more than 0: `pensize(0)` gives an error.

`clear()` rubs out everything the turtle has drawn. The turtle stays where it is, and keeps its colour and size, just like **erase all**. (The starting line `clearscreen()` is the bigger eraser that wipes the whole stage and starts again.)

## Stamp




```
shape("turtle")
stamp()
forward(100)
stamp()
```

**How it works:** `stamp()` prints a copy of the turtle onto the stage, like a rubber stamp. In Scratch the stamp looks like your costume. In Python it looks like the turtle's **shape**. `shape("turtle")` makes it look like a real turtle. You can also try the shapes `arrow`, `circle`, `square`, `triangle` and `classic` (the arrow it starts with). Remember the quote marks!

## Python superpowers that Scratch doesn't have

Scratch's Pen can only draw lines. The turtle can do more!

| Python | What it does |
|---|---|
| `circle(50)` | Draws a whole circle with a radius of 50 (that is the distance from the middle to the edge). The middle of the circle is on the turtle's **left**. A minus number, `circle(-50)`, puts it on the right. |
| `dot(30, "gold")` | Draws a filled-in dot, 30 across, right where the turtle is. It works even when the pen is up. |
| `fillcolor("skyblue")`<br>`begin_fill()`<br>…<br>`end_fill()` | Colours in a shape. Choose the colour, say `begin_fill()`, draw the shape, then say `end_fill()`. The inside gets painted. |
| `write("Hi!")` | Writes words on the stage where the turtle is. |
| `shape("turtle")` | Changes how the turtle looks. A bit like choosing a costume. |

Try them all at once:

```
shape("turtle")
circle(50)
dot(30, "gold")
fillcolor("skyblue")
begin_fill()
circle(-40)
end_fill()
write("Hi!")
```

You should see an empty circle above the turtle, a blue filled-in circle below it, a gold dot where they touch, and the word `Hi!` on top. It looks a bit like a snowman!

MISSION 1

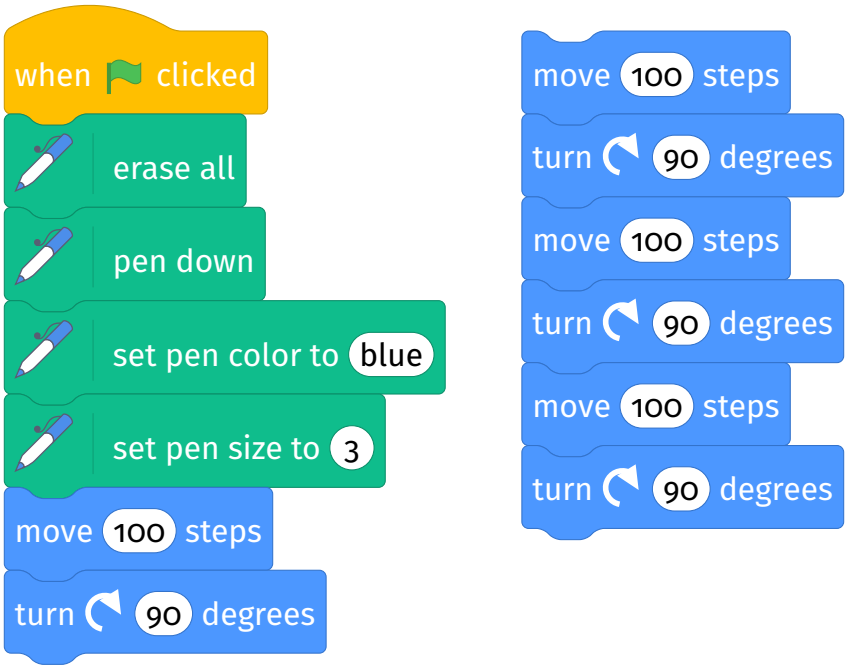
# A square the long way

**You will learn:** `forward`, `right`, `pencolor`, `pensize` and `shape`, all in one program

Time to put the blocks together. You will draw a blue square, one side at a time.

## Step 1: Build it in Scratch

Add the **Pen** extension, then build:




The right-hand blocks join on underneath the left-hand ones.

## Step 2: Type it in Colab

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
shape("turtle")
pencolor("blue")
pensize(3)
forward(100)
right(90)
forward(100)
right(90)
forward(100)
right(90)
forward(100)
right(90)
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | **when green flag clicked** **erase all** | The three starting lines. `clearscreen()` wipes the stage, so it does the job of **erase all** too. |
| — | **pen down** | No line needed. The turtle's pen starts down. |
| 4 | (choosing a costume) | `shape("turtle")` makes the turtle look like a turtle. |
| 5 | **set pen color to blue** | The colour name goes in quote marks. |
| 6 | **set pen size to 3** | A slightly fatter line. |
| 7–8 | **move 100 steps** **turn right 90 degrees** | The first side, then a turn at the corner. |
| 9–10 | the same again | The second side. |
| 11–12 | the same again | The third side. |
| 13–14 | the same again | The fourth side. The last turn makes the turtle face the way it started. |

### Step 4: Run it

Press . The turtle walks right, turns, walks down, and so on. You should see a blue square below the middle of the stage, with the turtle back where it started, facing right.

**Phew, that was long!**

Lines 7 and 8 are typed **four times**. In Scratch you would use **repeat 4** so you only build them once. Python has a shortcut just like that, and **Part 2** shows you how. For now, typing it the long way helps you see every step.

### Step 5: Make it yours

**Make it yours**

☐ Change `"blue"` to your favourite colour, and the pen size to 8.
☐ Make a bigger square: change all four `100`s to `150`.
☐ Make each side a different colour. Add a `pencolor(...)` line before each `forward`.
☐ Change every `right` to `left`. Where does the square go now? Predict first, then run it.

MISSION 2

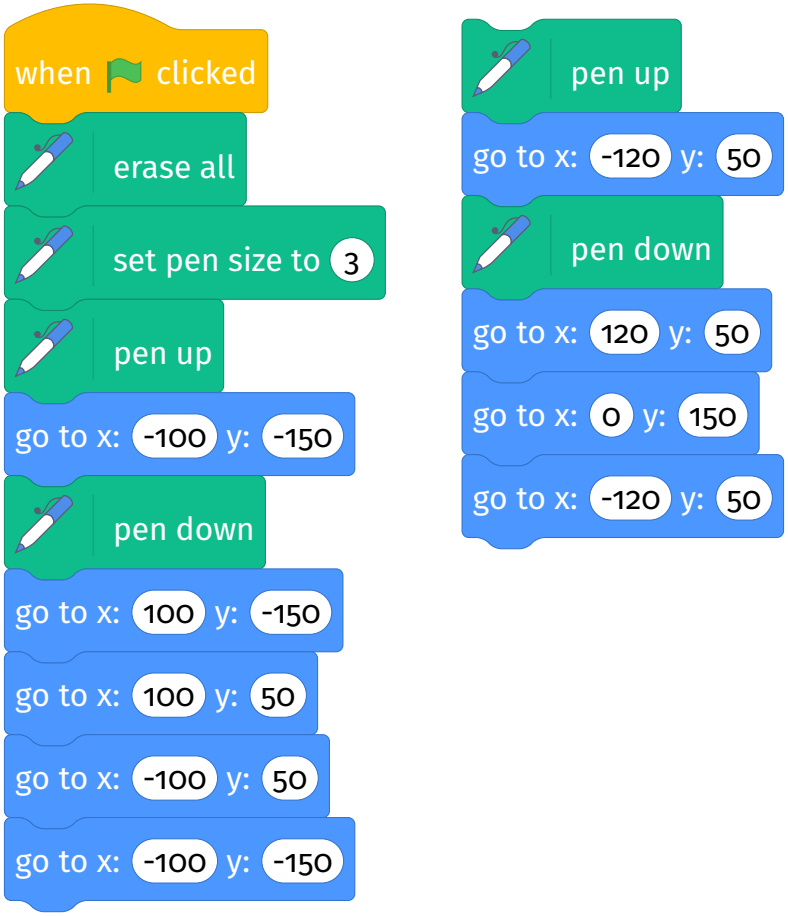
# Draw a house

**You will learn:** moving without drawing with `penup` and `goto`, and filling shapes with colour

Now you will draw a house with a roof, a door and a round window. You will use `goto` for every corner, and `penup()` to move between the parts without leaving a line.

## Step 1: Build it in Scratch

Scratch can't fill shapes with colour, so the Scratch house is just outlines. Here are the walls and the roof:




The right-hand blocks join on underneath the left-hand ones. The door works the same way.

## Step 2: Type it in Colab

The empty lines split the program into parts: walls, roof, door and window. Python doesn't mind empty lines, and they make it easier to read.

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
speed(8)
pensize(3)
```

**...and carry on in the same cell**

```
penup()
goto(-100, -150)
pendown()
fillcolor("tan")
begin_fill()
goto(100, -150)
goto(100, 50)
goto(-100, 50)
goto(-100, -150)
end_fill()

penup()
goto(-120, 50)
pendown()
fillcolor("firebrick")
begin_fill()
goto(120, 50)
goto(0, 150)
goto(-120, 50)
end_fill()
```

**...and carry on in the same cell**

```
penup()
goto(-25, -150)
pendown()
fillcolor("saddlebrown")
begin_fill()
goto(-25, -70)
goto(25, -70)
goto(25, -150)
end_fill()

penup()
goto(55, -10)
dot(40, "skyblue")
hideturtle()
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | **when green flag clicked** | The three starting lines. |
| 4 | (none) | `speed(8)` makes the turtle quicker, so you don't wait too long. |
| 5 | **set pen size to 3** | Fatter lines for the outline. |
| 7–9 | **pen up**<br>**go to x: -100 y: -150**<br>**pen down** | The "moving without a line" trick. The turtle hops to the bottom-left corner of the walls. |
| 10–16 | **go to x: 100 y: -150** and the other corners | Choose the paint colour and say `begin_fill()`: "start remembering this shape". Walk round the four corners and back to the start. Then `end_fill()` paints the inside tan. Scratch can't fill! |
| 18–26 | the roof | The same pattern again: hop, choose a colour, draw three corners, fill. A triangle only needs three corners! |
| 28–36 | the door | The same pattern a third time. The door starts at the bottom of the walls. |
| 38–41 | **hide** | Hop to the window spot and drop a big blue dot. Then `hideturtle()` hides the turtle so you can admire your house. |

## Step 4: Run it

You should see a tan house with a red roof, a brown door and a round blue window.

**If a line goes across your house**

You probably forgot a `penup()`, so the turtle drew a line as it moved. Check lines 7, 18, 28 and 38.

## Step 5: Make it yours

**Make it yours**

☐ Paint your house in your own colours.
☐ Add a second window on the left. Copy lines 38 to 40 to the end, and change the x number.
☐ Add a sun in the top corner: hop to `goto(-190, 140)` and use a big `dot(60, "gold")`.
☐ Write your house number on the door. Hop to the door, then use `write("12")`.

MISSION 3

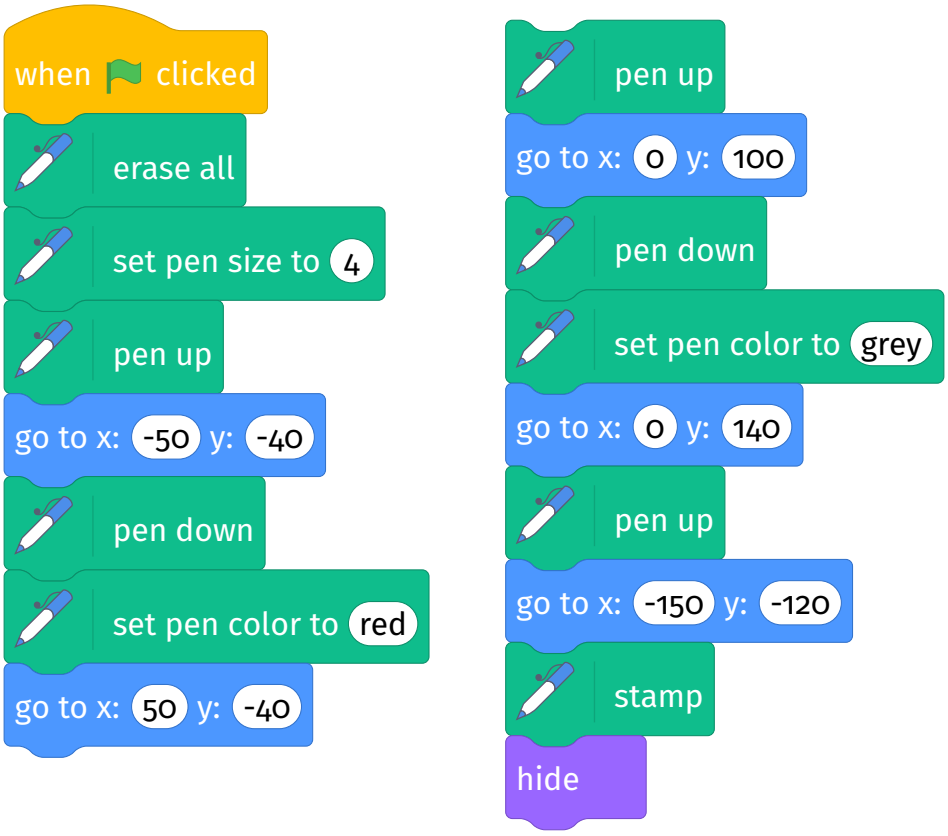
# Robot face

**You will learn:** `circle`, `dot`, `write` and `stamp`: the Python superpowers

Meet Robo! You will draw a robot head with a round face, big eyes, a mouth and an antenna. Then Robo gets a little pet turtle.

## Step 1: Build it in Scratch

Scratch has no circle block, so the head and eyes are hard to make in Scratch. Here is the part Scratch **can** do: the mouth, the antenna and the pet.



## Step 2: Type it in Colab

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
speed(10)
pensize(4)
pencolor("slategray")
```

**...and carry on in the same cell**

```
penup()
goto(0, -100)
pendown()
fillcolor("lightgray")
begin_fill()
circle(100)
end_fill()

penup()
goto(-40, 30)
dot(40, "white")
dot(16, "black")
goto(40, 30)
dot(40, "white")
dot(16, "black")
```

**...and carry on in the same cell**

```
goto(-50, -40)
pendown()
pencolor("red")
goto(50, -40)

penup()
goto(0, 100)
pendown()
pencolor("slategray")
goto(0, 140)
dot(20, "gold")

penup()
goto(120, -150)
write("I am Robo!")
shape("turtle")
color("seagreen")
goto(-150, -120)
stamp()
hideturtle()
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | **when green flag clicked** | The three starting lines. |
| 4–6 | **set pen size to 4**<br>**set pen color to** | Fast drawing, a thick line and a grey-blue colour called `"slategray"`. |
| 8–10 | **pen up** **go to x: 0 y: -100**<br>**pen down** | Hop to the bottom of where the face will be. |
| 11–14 | (Scratch can't do this) | `circle(100)` draws the face. The middle of the circle is 100 steps to the turtle's left. The turtle faces right, so its left is **up**, and the middle lands at $(0, 0)$. The fill paints it light grey. |
| 16–22 | **pen up** **go to x: -40 y: 30** | Hop to the left eye. A big white dot with a small black dot on top makes an eyeball. Then the same for the right eye. Dots work with the pen up. |
| 24–27 | **go to x: -50 y: -40** … | The mouth: pen down, red, and a straight line across. |
| 29–34 | **go to x: 0 y: 100** … | The antenna: a short line up from the top of the head, and a gold dot on the end. |
| 36–38 | (Scratch can't do this) | Hop below the face and `write` a message. Words go in quote marks. |
| 39–40 | (choosing a costume) | Turn into a turtle shape. `color("seagreen")` sets the pen colour **and** the fill colour at once, so the turtle becomes green. |
| 41–42 | **go to x: -150 y: -120** **stamp** | Walk to the bottom-left and stamp a copy of the turtle: Robo's pet! |
| 43 | **hide** | Hide the real turtle. The stamp stays behind. |

## Step 4: Run it

You should see a grey round robot face with two big eyes, a straight red mouth and an antenna with a gold ball on top. "I am Robo!" is written in the bottom-right, and a green turtle stamp sits in the bottom-left.

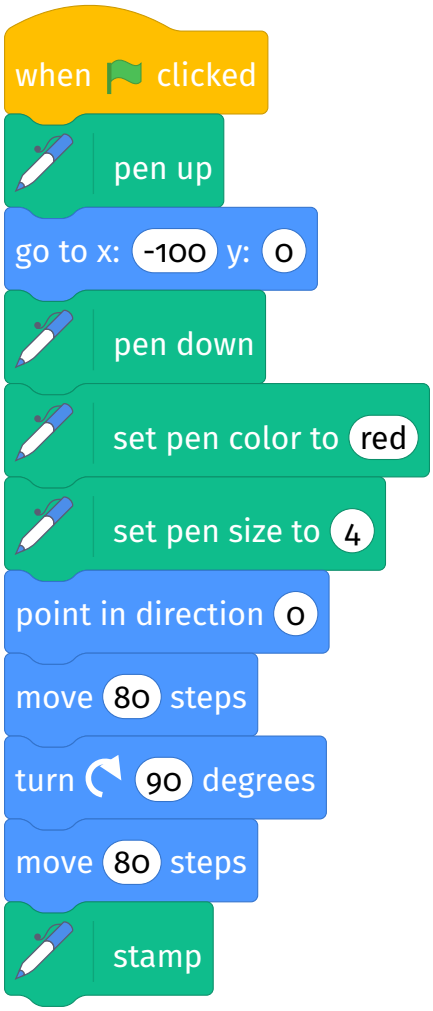
## Step 5: Make it yours

**Make it yours**

☐ Give Robo different coloured eyes, like `"lime"` or `"hotpink"`.
☐ Change the message to your own robot's name.
☐ Make a smile: replace line 27 with `goto(0, -60)` and add `goto(50, -40)` after it. Now the mouth is a V shape.
☐ Stamp a whole family of pets. After line 42, add more `goto(...)` and `stamp()` pairs. Try changing `color(...)` between stamps.

## Translate it yourself

Here is a Scratch script you haven't seen. Write the Python for it on the lines below. Start with the three starting lines. Then type it into Colab to check. Hints: remember the secret rule for **point in direction**, and remember that the turtle's pen starts down. (The answer is at the back of the book.)




**My Python**

**Two-line puzzle**

The turtle is in the middle, facing right, as always. Make it point **straight down** and walk **100 steps**. You may use only **two lines** of Python after the three starting lines. (Hint: Scratch's direction for down is 180.)

## Part 1 checklist

Tick each one you can do.

☐ I can move the turtle with `forward` and `backward`, and turn it with `right` and `left`.

☐ I can send the turtle to a spot with `goto(x, y)`, and choose how fast it goes with `speed`.

☐ I can turn a Scratch direction into Python with `setheading(90 - d)`.

☐ I can move without drawing using `penup()` and `pendown()`.

☐ I can choose a pen colour and size, `clear()` the drawing and `stamp()` the turtle.

☐ I can use a Python superpower: `circle`, `dot`, `begin_fill` or `write`.

PART 2

# Loops

Make the computer do the repeating for you.

In Part 1 you drew pictures one command at a time. To draw a square you typed `forward(100)` and `right(90)` four times each. That is a lot of typing! In Scratch you would never do that. You would grab a **repeat 4** block and let it do the work.

In this part you will learn the Python for the orange **Control** blocks that repeat and wait:

**repeat 10** **forever** **wait 1 seconds** **stop this script**

By the end you will draw stars, spirals and flowers with just a few lines of code.

## Repeat becomes `for`

Here is the square from Part 1, but with a loop. Look at the two sides and match each block to its line. (In this part, the side-by-side pictures leave out the three starting lines to save space. When you type a drawing program into Colab, always start with them.)




```
1 for i in range(4):
2     forward(100)
3     right(90)
```

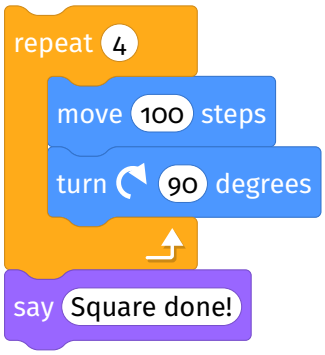
**How it works.** Line 1, `for i in range(4):`, is the **repeat 4** block. You can read it as "for each turn, 4 times". The word `range(4)` means "count 4 times". Lines 2 and 3 are the blocks inside the loop, so they happen 4 times.

## The mouth of the C: a colon and four spaces

In Scratch, a block goes **inside** a loop when you drop it into the C-shaped mouth. Python has no C shape, so it uses two things instead:

1. A **colon** `:` at the end of the `for` line. The colon **opens the mouth**. It says "the inside starts on the next line".
2. **Four spaces** at the start of a line. Those lines are **inside** the mouth.

The first line that goes back to the left edge, with no spaces, is the block snapped **under** the C. It runs once, after the loop has finished.




```
1 for␣i␣in␣range(4):
2 ␣␣␣␣forward(100)
3 ␣␣␣␣right(90)
4 print("Square␣done!")
```

The little ␣ marks show the spaces, which are normally invisible. Lines 2 and 3 have four spaces, so they repeat 4 times. Line 4 has no spaces, so it is outside the loop. You see `Square done!` only once, when the square is finished.

**Colab helps you**

When you type the colon and press **Enter**, Colab puts the four spaces in for you. To come back out of the loop, press **Backspace** once at the start of the line.

## What does `i` count?

The `i` in `for i in range(4):` is a **counter**. Each time the loop goes round, Python changes `i` by itself. Scratch's **repeat** block has no counter you can see. In Scratch you would have to make one yourself.

Let's peek at `i`. This program doesn't draw, so it doesn't need the three starting lines.

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

**You should see**

```
0
1
2
3
```

Python starts counting at **0**, so `i` is 0, then 1, then 2, then 3. That is still 4 times round! Notice that `range(4)` stops **before** it gets to 4. You don't have to use `i`, but it is always there if you want it.

## Using `i` inside the loop

Because `i` changes every time round, you can use it to make each turn of the loop a little different. The star `*` means **times** in Python (there is no × key on the keyboard).

```
for i in range(5):
    print(i * 10)
```

**You should see**

```
0
10
20
30
40
```

Now imagine `forward(i * 5)` instead of `print(i * 10)`. The turtle would walk a bit further each time round. That is how you will draw a spiral in a mission soon! A name like `i` that holds a number is called a **variable**. You will make your own variables in Part 3.

## When the spaces go wrong

Python is fussy about the colon and the spaces. Here are the two red messages you will see most often. Read them like a detective's clue note: the last line tells you what is wrong.

### Forgot the colon?

```
for i in range(3)
    print("hi")
```

```
SyntaxError: expected ':'
```

Python wants the colon that opens the mouth. Add `:` to the end of the **`for`** line.

### Forgot the spaces?

```
for i in range(3):
print("hi")
```

```
IndentationError: expected an indented block after 'for' statement on
line 1
```

This means "the C has an empty mouth". Nothing is inside the loop. Put four spaces in front of `print("hi")`.

**Watch out: the sneaky bug with no red writing**

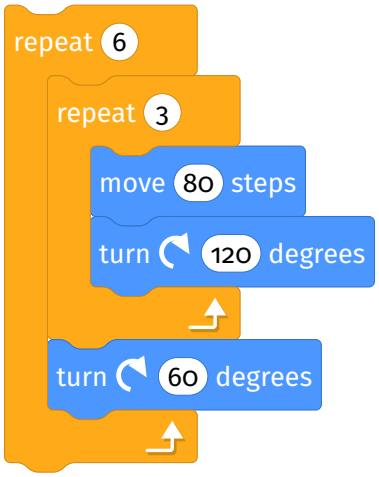
What if only **one** line loses its spaces?

```
for i in range(4):
    forward(100)
right(90)
```

Python is happy, so there is **no error**. But `right(90)` is now outside the loop, like a block dropped **under** the C instead of inside it. The turtle walks forward 4 times in a row (one long line that runs off the stage) and turns only once, at the very end. If your drawing looks wrong, check the spaces first!

## A loop inside a loop

In Scratch you can put a **repeat** block inside another **repeat** block. Python can do the same. The inside loop gets **more** spaces.




```
for i in range(6):
    for j in range(3):
        forward(80)
        right(120)
    right(60)
```

**How it works.** Count the spaces at the start of each line:

- **0 spaces** (line 1): the outside loop. It goes round 6 times.
- **4 spaces** (lines 2 and 5): inside the outside loop.
- **8 spaces** (lines 3 and 4): inside **both** loops. They draw one triangle.

Line 5 has 4 spaces, so it is not part of the triangle. It turns the turtle a little after each triangle. The inside loop uses a different counter letter, `j`, so the two counters don't get mixed up. The result is 6 triangles spinning round the middle, 18 lines in all.

## Forever becomes `while True`

The **forever** block repeats with no end. In Python it is `while True:`. Read it as "keep going while true is true", and true is always true!




```
while True:
    forward(100)
    right(90)
```

**How it works.** The colon and the four spaces work just like in a `for` loop. The difference is that this loop **never stops on its own**. The turtle draws the same square again and again,

and the cell keeps running. While a cell is running, its play button turns into a spinning circle. Click it ((■)) to stop the cell, just like the red stop sign in Scratch. You can also use the menu: **Runtime**, then **Interrupt execution**.

**Watch out: forever means forever**

When you stop the cell, Colab shows some red writing that ends with `KeyboardInterrupt`. That is not a mistake. It just means "you pressed stop". While a forever loop is running, no other cell can run, so remember to stop it. For drawings, a `for` loop with a number is usually better: it finishes by itself, and you still see the whole picture.

## Wait becomes `time.sleep`

The **wait 1 seconds** block pauses the script. Python keeps its clock in a toolbox called `time`. You open the toolbox once, at the top, with `import time`. Then `time.sleep(2)` means "sleep for 2 seconds".




```
1 import time
2 print("Ready...")
3 time.sleep(2)
4 print("Go!")
```

The dot in `time.sleep` means "the `sleep` tool from the `time` toolbox". If you forget line 1, Python gives a `NameError`, because it doesn't know what `time` is yet.

**Countdown rocket.** Here a loop and a wait work together. Each time round, `5 - i` works out the next number: $5 - 0 = 5$, then $5 - 1 = 4$, and so on.

```
1 import time
2 for i in range(5):
3     print(5 - i)
4     time.sleep(1)
5 print("Blast off!")
```

**You should see**

```
5
4
3
2
1
Blast off!
```

**Predict first**

What would happen if `print("Blast off!")` had four spaces in front of it? Say what you think, then try it. (Remember the sneaky bug!)

## Stop this script becomes break

The **stop this script** block ends the script. Inside a Python loop, the word **break** does the same job: it **breaks out** of the loop straight away.




```
while True:
    print("Hello!")
    break
print("Out of the loop")
```

This prints `Hello!` only once, then `Out of the loop`. Here is something Scratch can't do: in Scratch nothing can snap under a **forever**, but in Python the line after the loop runs as soon as **break** jumps out. Right now **break** looks a bit silly, because it always stops the loop the first time round. In Part 5 you will learn **if**, and then **break** becomes really useful: "if the guess is right, stop".

**What about repeat until?**

Scratch also has **repeat until**. In Python it becomes **while not** `...`:, where the dots are a question like "has the cat reached the edge?". You need a few more tools to ask those questions, so **repeat until** waits for Part 5.

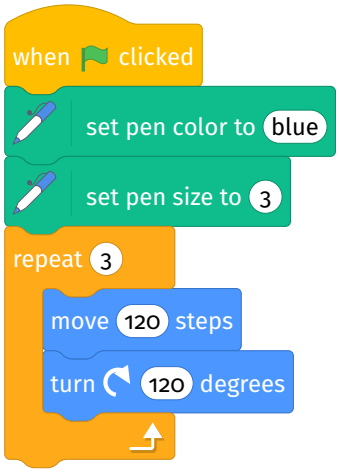
MISSION 4

# Draw shapes the short way

**You will learn:** `for` loops, the colon and four spaces, and the 360 trick

Now you can draw any shape with just three lines of loop. Let's start with a triangle. (In Scratch, add the **Pen** extension first.)

## Step 1: Build it in Scratch




## Step 2: Type it in Colab

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
pencolor("blue")
pensize(3)
for i in range(3):
    forward(120)
    right(120)
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | **when green flag clicked** | The three starting lines. They get the stage and the turtle ready. The pen starts down. |
| 4 | **set pen color to blue** | The colour is a word in quote marks, in small letters. |
| 5 | **set pen size to 3** | A thicker line. |
| 6 | **repeat 3** | `for i in range(3):` means "do this 3 times". The colon opens the mouth. |
| 7 | **move 120 steps** | Four spaces in front, so it is inside the loop. |
| 8 | **turn right 120 degrees** | Four spaces too, so this is inside the loop as well. |

### Step 4: Run it

Press . The turtle draws a blue triangle, then stops back where it started, pointing the same way as at the beginning.

### Step 5: Make it yours

Here is a trick: **when the turtle goes all the way round a shape, its turns add up to 360 degrees**, one full circle. So to find the turn, divide 360 by the number of sides. Fill in the table, then draw each shape by changing the numbers in lines 6, 7 and 8. Bigger shapes need a smaller move, or they run off the stage.

| Shape | Sides (range) | Move | Turn | Drawn? |
|---|---|---|---|---|
| Triangle | 3 | 120 | $360 \div 3 = 120$ | ☐ |
| Square | 4 | 120 | $360 \div 4 = 90$ | ☐ |
| Pentagon | 5 | 100 | _____ | ☐ |
| Hexagon | 6 | 80 | _____ | ☐ |
| Octagon | 8 | 60 | _____ | ☐ |
| Decagon | 10 | 50 | _____ | ☐ |
| Almost a circle | 36 | 10 | _____ | ☐ |

**Make it yours**

☐ Fill in the table and draw every shape. Which one looks most like a circle?
☐ Add `speed(13)` just after line 3 so the turtle draws faster.
☐ Put `pencolor("red")` **inside** the loop, with four spaces, just before `forward(120)`. Does anything change? Why?
☐ Add `print("Ta-da!")` at the very end with **no** spaces. How many times does it print? Now give it four spaces and run it again.

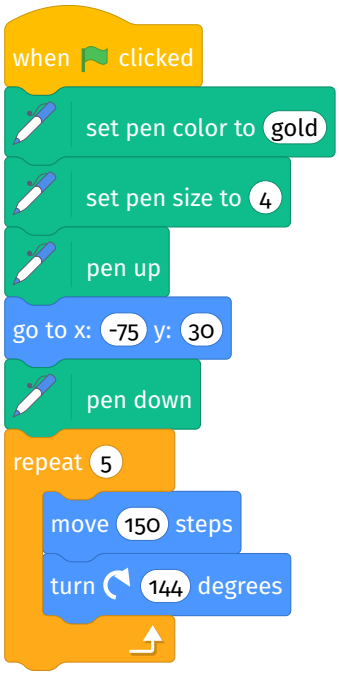
MISSION 5

# Stars

**You will learn:** turning more than once round, and moving before you draw

A star is a shape that turns **further** than a pentagon. Each point is a sharp turn of 144 degrees.

## Step 1: Build it in Scratch




## Step 2: Type it in Colab

**Type this into a code cell**

```
1  from ColabTurtlePlus.Turtle import *
2  clearscreen()
3  initializeTurtle((480, 360))
4  pencolor("gold")
5  pensize(4)
6  penup()
7  goto(-75, 30)
8  pendown()
9  for i in range(5):
10     forward(150)
11     right(144)
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | **when green flag clicked** | The three starting lines. |
| 4–5 | **set pen color to gold**<br>**set pen size to 4** | A thick gold pen. |
| 6 | **pen up** | Lift the pen so the next move leaves no line. |
| 7 | **go to x: -75 y: 30** | Jump a little to the left, so the star ends up in the middle of the stage. |
| 8 | **pen down** | Ready to draw again. |
| 9 | **repeat 5** | 5 points, so 5 times round. |
| 10–11 | **move 150 steps**<br>**turn right 144 degrees** | Inside the loop: one long line and one sharp turn for each point. |

## Step 4: Run it

Press . A big gold star appears in the middle of the stage, and the turtle ends up back at the start.

**Twice round!**

A pentagon turns $5 \times 72 = 360$ degrees: once round. The star turns $5 \times 144 = 720$ degrees. That is $360 + 360$, so the turtle spins round **twice** while it draws the star. That's why its lines cross over in the middle.

## Step 5: Make it yours

For a star with an **odd** number of points (5, 7, 9, ...), here is a recipe for the turn:

$$\text{turn} = 180 - (180 \div \text{points})$$

Check it for 5 points: $180 \div 5 = 36$, and $180 - 36 = 144$. It works! Fill in the table and draw each star. Use the number of points in `range( )` too.

| Points | $180 \div$ points | Turn | Drawn? |
|---|---|---|---|
| 5 | 36 | 144 | ☐ |
| 9 | _____ | _____ | ☐ |
| 15 | _____ | _____ | ☐ |
| 45 | _____ | _____ | ☐ |

**Make it yours**

☐ Fill in the star table. The 45-point star looks like a fuzzy sun! Add `speed(13)` after line 3 for that one.
☐ **Even stars** need a different turn. Try `range(8)` with `right(135)`, and `range(12)` with `right(150)`.
☐ Change the colour and the pen size. Try `"skyblue"` or `"hotpink"`.
☐ Change line 7 to put the star in a corner of the stage. Can you keep it all on the stage?

MISSION 6

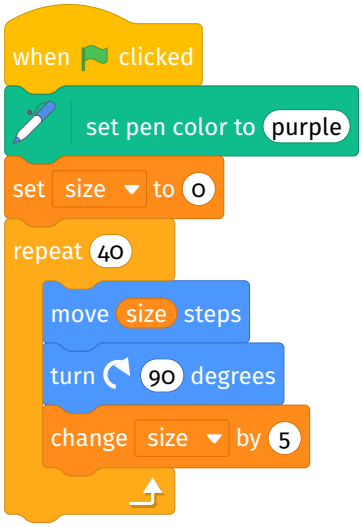
# Square spiral

**You will learn:** using the counter `i` to change each step

In this mission each line is a little longer than the last, so the turtle spirals outwards. In Scratch you need a variable for this. In Python the counter `i` does the job for free!

In Scratch, make a variable called **size** first.

## Step 1: Build it in Scratch




## Step 2: Type it in Colab

**Type this into a code cell**

```
1 from ColabTurtlePlus.Turtle import *
2 clearscreen()
3 initializeTurtle((480, 360))
4 pencolor("purple")
5 speed(13)
6 for i in range(40):
7     forward(i * 5)
8     right(90)
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | **when green flag clicked** | The three starting lines. |
| 4 | **set pen color to purple** | A purple pen. |
| 5 | (none) | Fastest drawing, so you don't wait long for 40 lines. |
| — | **set size to 0** | No line needed! The counter `i` starts at 0 by itself. |
| 6 | **repeat 40** | 40 times round. `i` counts 0, 1, 2, … up to 39. |
| 7 | **move size steps** | `i * 5` means "`i` times 5". The first line is $0 \times 5 = 0$ steps, then 5, then 10, and so on, up to $39 \times 5 = 195$ steps. |
| 8 | **turn right 90 degrees** | A square corner each time. |
| — | **change size by 5** | No line needed! `i` goes up by 1 every time round, so `i * 5` goes up by 5. |

### Step 4: Run it

Press . A purple square spiral grows out from the middle of the stage. The lines get longer and longer, 5 steps at a time.

**A puzzle for you**

The loop goes round 40 times, but if you count carefully, the turtle draws only 39 lines. Where did one go? (Look at the first time round. The answer is at the back of the book.)

### Step 5: Make it yours

**Make it yours**

☐ Change `right(90)` to `right(91)`. Just one degree more makes the spiral twist round like a whirlpool!
☐ Change the turn to `right(120)` for a triangle spiral, or `right(60)` for a hexagon spiral.
☐ Try `range(60)` with `forward(i * 3)`. Why does the spiral get more lines but stay about the same size?
☐ **Super challenge:** what number in line 6 would make the spiral run off the stage? Guess first, then test it.

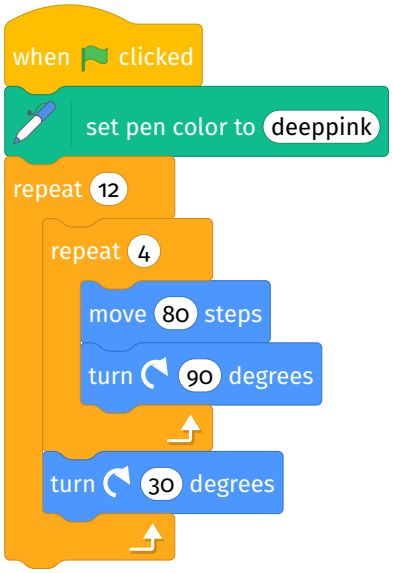
MISSION 7

# Flower of squares

**You will learn:** loops inside loops, with 8 spaces

One square is plain. Twelve squares, each turned a little, make a flower!

## Step 1: Build it in Scratch




## Step 2: Type it in Colab

**Type this into a code cell**

```
1  from ColabTurtlePlus.Turtle import *
2  clearscreen()
3  initializeTurtle((480, 360))
4  pencolor("deeppink")
5  speed(13)
6  for i in range(12):
7      for j in range(4):
8          forward(80)
9          right(90)
10     right(30)
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | **when green flag clicked** | The three starting lines. |
| 4–5 | **set pen color to deeppink** | A deep pink pen, drawing fast. |
| 6 | **repeat 12** | The outside loop: 12 squares. No spaces in front. |
| 7 | **repeat 4** | The inside loop: one square. It has 4 spaces, so it is inside the outside loop. It gets its own counter, `j`. |
| 8–9 | **move 80 steps**<br>**turn right 90 degrees** | 8 spaces: inside **both** loops. These two lines run $12 \times 4 = 48$ times. |
| 10 | **turn right 30 degrees** | Back to 4 spaces: after each square, turn a little. It runs 12 times. |

### Step 4: Run it

Press . The turtle draws a square, turns a little, draws another square, and so on. After 12 squares you have a pink flower. The turns add up to $12 \times 30 = 360$, so the flower goes exactly once round.

### Step 5: Make it yours

**Make it yours**

☐ Change line 6 to `range(36)` and line 10 to `right(10)`. Why do those two numbers belong together? (Hint: the 360 trick.)
☐ Make a flower of triangles: change the inside loop to `range(3)` and `right(120)`.
☐ Make a flower of stars: inside loop `range(5)` with `right(144)`, outside loop `range(8)` with `right(45)`.
☐ Add `hideturtle()` at the very end so only the flower shows. How many spaces should it have?

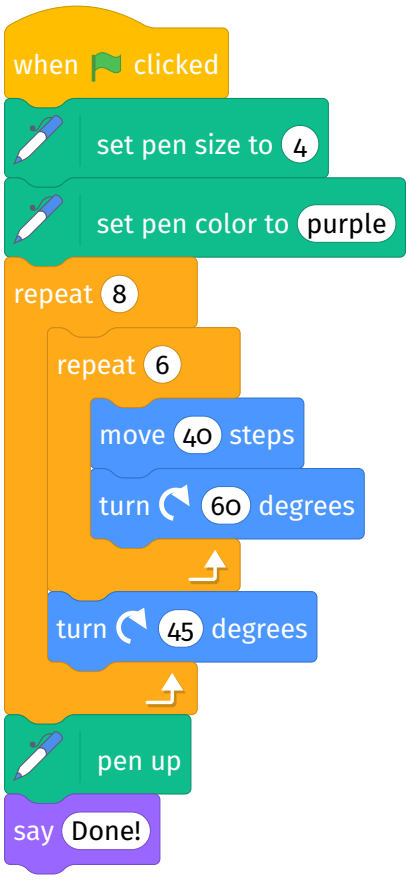
**Watch out!**

If you get one square and then a mess, check line 10. It needs exactly **4** spaces. With 8 spaces it is inside the square loop, and every corner turns an extra 30 degrees.

## Translate it yourself

Here is a Scratch script you haven't seen before. Write the Python on the lines. Start with the three starting lines.

1. **Draw a line** from each block to the Python word that does the same job.
2. **Write the Python** on the lines. Count your spaces carefully!
3. **Type it into Colab** and run it. Did the turtle draw the same thing as the Scratch sprite?




**My Python**

**Clues**

How many spaces does **move 40 steps** need? It is inside two C blocks. And where does **pen up** go: inside a loop, or after?

**Two-line puzzle**

Write a program that prints the numbers **1, 2, 3, 4, 5**, one on each line. You may only use **two lines** of Python, and one of them must be a `for` loop.
**Clue:** `i` starts at 0, but you want to start at 1. What can you do to `i` inside the loop?

## Part 2 checklist

Tick each one when you can do it without looking back.

☐ I can turn **repeat 10** into `for i in range(10):` with a colon and four spaces.

☐ I know that `i` counts 0, 1, 2, … and I can use it inside a loop, like `forward(i * 5)`.

☐ I can put a line **after** a loop by taking its spaces away.

☐ I can put a loop inside a loop, using 8 spaces for the innermost lines.

☐ I can use `while True:` and stop it with the stop button .

☐ I can make Python wait with `import time` and `time.sleep(1)`.

PART 3

# Variables and Lists

Give your programs a memory: boxes that remember one thing, and backpacks that carry lots.

In Scratch, the orange **Variables** blocks and the red-orange **List** blocks give your project a memory. A variable remembers **one** thing, like a score. A list remembers **lots** of things in order, like everything in a backpack.

In this part you will meet the Python for blocks like these:

**set score to 0** **change score by 1** **show variable score**

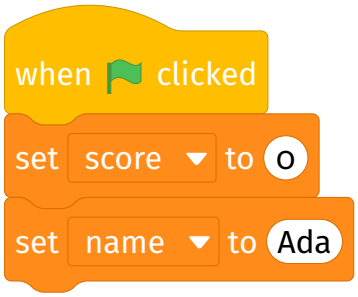
**add apple to backpack** **item 1 of backpack**

**length of backpack** **backpack contains sword?**

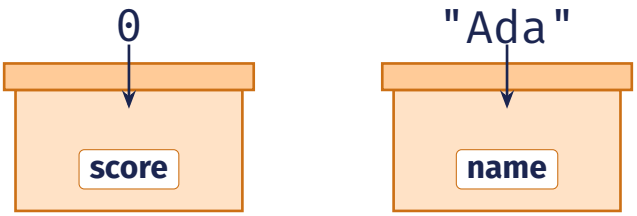
Good news: in Python, variables and lists need **no** starting lines. Programs that do not draw can start straight away.

## A variable is a labelled box

In Scratch, you click **Make a Variable**, type a name, and Scratch makes a box for you. In Python there is no button. You make a box just by putting something in it.




```
score = 0
name = "Ada"
```



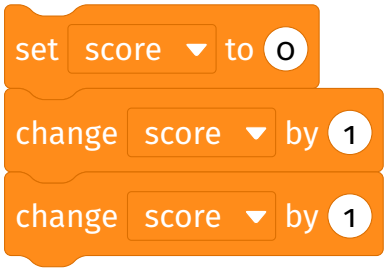

**How it works:** `score = 0` means "put 0 in the box labelled `score`". In Python, one = does **not** mean "equals". It means **put in the box**. Read it from right to left: the value on the

right goes into the box on the left. A box holds one thing at a time, so putting something new in throws the old thing out.

**Why does "Ada" have quote marks?**

Quote marks tell Python "this is writing". Without them, `name = Ada` makes Python look for a box called Ada, and it can't find one. Numbers like `0` don't need quote marks.

## Change the score




```
1 score = 0
2 score = score + 1
3 score += 1
```

**How it works:** Line 2 looks strange if you think `=` means "equals". But remember, it means **put in the box**. Python works out the right side first: "what's in `score`, plus 1". That is 1. Then it puts the 1 back in the box. Line 3 is a shortcut that does exactly the same thing: `score += 1` means "change score by 1". After both lines, `score` holds 2.

**Going down**

**change score by -1** is `score -= 1` (or `score = score - 1`). You can use `+=`, `-=`, `*=` and `/=`.

## Show the variable

In Scratch, ticking the box next to a variable shows a **monitor** on the stage that updates all by itself. Python has no monitor. Instead, you `print` the box whenever you want to look inside.




```
1 score = 7
2 print(score)
3 print("Score:", score)
```

**You should see**

```
7
Score: 7
```

**How it works:** `print(score)` has no quote marks around `score`, so Python shows what is **inside** the box, not the word "score". In line 3, the comma lets you print two things in a row, and Python puts a space between them for you. That is easier than Scratch's **join** block!

## Naming your boxes

Scratch lets you call a variable almost anything, even `my score` with a space. Python is pickier. A Python name must follow these rules:

- **No spaces.** Use an underscore _ instead: `my_score`.
- **Don't start with a digit.** `score2` is fine, `2score` is not.
- **Capitals matter.** `Score` and `score` are two different boxes.
- **Not a Python word.** You can't call a box `for` or `import`, because Python already uses those words.

| Name | OK? | Why |
|---|---|---|
| `score` | **Yes** | Small letters are perfect. |
| `my_score` | **Yes** | The underscore joins the words. |
| `score2` | **Yes** | Digits are fine after the first letter. |
| `my score` | **No** | A space. Python thinks these are two names. |
| `2score` | **No** | It starts with a digit. |
| `for` | **No** | Python already uses this word for loops. |

**Watch out: capitals make a different box**

```
Score = 10
print(score)
```

```
NameError: name 'score' is not defined
```

Python only knows the boxes you have made. Read the last line of the red error: it names the box Python can't find. Look closely: the box you made is `Score`, with a capital S. Use small letters for your names and you will rarely hit this bug.

**Quick quiz: box or bust?**

Tick the names Python will accept. (Answers at the back.)

☐ `lives` ☐ `high score` ☐ `player_1`
☐ `1st_place` ☐ `bestTime` ☐ `import`

## Text or number? Quotes matter!

Scratch quietly guesses whether something is a number or writing. Python never guesses. `5` is a number. `"5"` in quote marks is writing (Python calls writing a **string**).

```
1 print(5 + 5)
2 print("5" + "5")
```

**You should see**

```
10
55
```

**How it works:** Adding two numbers does maths: `10`. Adding two strings sticks them together, like the join block: `55`. In Scratch, both would give 10. And if you try to add a string and a number, Python stops and complains:

```
score = 5
print("Score: " + score)
```

```
TypeError: can only concatenate str (not "int") to str
```

This means "I can only join writing to writing, and you gave me a number". The easy fix is the comma you already know: `print("Score:", score)`.

**For this sprite only, or for all sprites?**

When you make a Scratch variable, Scratch asks: **For all sprites** or **For this sprite only**? In this part, all your Python boxes are like "for all sprites": any line in your program can use them. In a later part you will learn how to give each sprite its own boxes, like a cat with its own `health` and a dog with its own `health`.

## Variables can draw too

A box can hold the size of a shape. Change the box, and every line that uses it changes too.



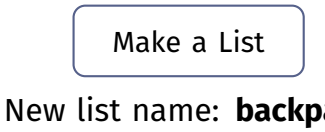

```
1  from ColabTurtlePlus.Turtle import *
2  clearscreen()
3  initializeTurtle((480, 360))
4  size = 50
5  for i in range(4):
6      forward(size)
7      right(90)
8  size = size + 50
9  for i in range(4):
10     forward(size)
11     right(90)
```

**How it works:** `forward(size)` has no quote marks, so the turtle walks however many steps are in the `size` box. The first square uses 50. Line 8 makes the box hold 100, so the second square is twice as big. Both squares start from the same corner.

What if you put `size = size + 50` **inside** the loop, so the size grows after every line? You will find out in the **Growing spiral** mission...

## A list is a backpack

A variable holds one thing. A **list** holds lots of things, in order, under one name. In Scratch you click **Make a List**. In Python you write square brackets.

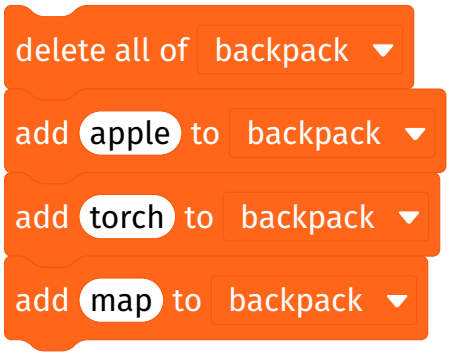



New list name: **backpack**

```
backpack = []
print(backpack)
```

**How it works:** The square brackets `[ ]` are the backpack. With nothing between them, the backpack is empty, and `print(backpack)` shows `[]`. Printing a list is Python's **show list backpack**.

## Put things in the backpack




```
backpack = []
backpack.append("apple")
backpack.append("torch")
backpack.append("map")
print(backpack)
```

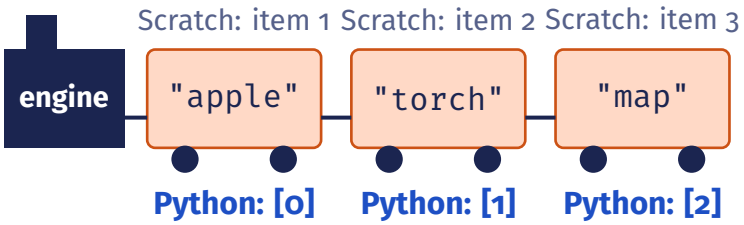
**You should see**

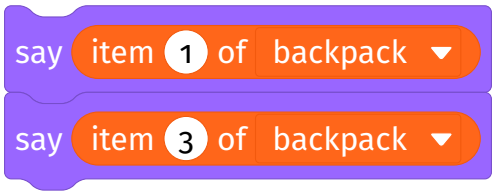
```
['apple', 'torch', 'map']
```

**How it works:** `append` means "add to the end". The dot joins the list's name to the job you want it to do: `backpack.append("torch")` means "backpack, add torch to your end". You can also fill a backpack in one go: `backpack = ["apple", "torch", "map"]`. Commas go between the items.

## Python counts from 0!

Think of a list as a train. Each carriage carries one item. Scratch numbers the carriages like seats: 1, 2, 3. Python counts **how many carriages away from the front** each one is, so the first carriage is number **0**.




```
print(backpack[0])
print(backpack[2])
```

**You should see**

```
apple
map
```

**How it works:** The square brackets after the name pick one carriage. Here is the rule to remember: **Scratch item $n$ is Python `[n - 1]`**. Item 1 is `backpack[0]`, item 3 is `backpack[2]`.

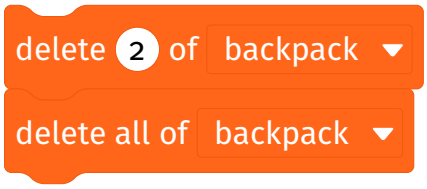
## How long is the list?




```
print(len(backpack))
```

**How it works:** `len` is short for **length**. It counts the carriages: for our backpack it gives 3. Notice that the length is 3 but the last item is `backpack[2]`. That is the count-from-0 rule again.

## Take things out

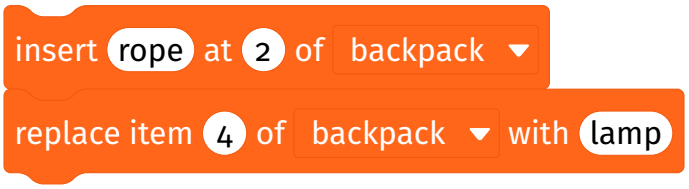



```
del backpack[1]
backpack.clear()
```

**How it works:** `del` is short for **delete**. Scratch's item 2 is Python's `[1]`, so `del backpack[1]` takes out `"torch"`, and the train closes the gap: `['apple', 'map']`. `backpack.clear()` empties the whole backpack, leaving `[]`.

## Squeeze in and swap

Start again with `backpack = ["apple", "torch", "map"]`.



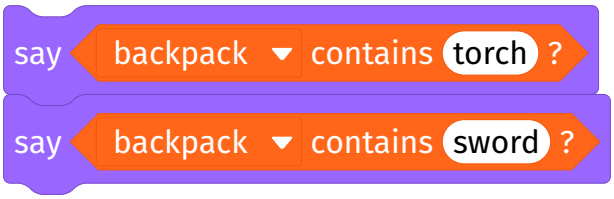

```
backpack.insert(1, "rope")
backpack[3] = "lamp"
```

**After line 1, then after line 2**

```
['apple', 'rope', 'torch', 'map']
['apple', 'rope', 'torch', 'lamp']
```

**How it works:** `insert(1, "rope")` squeezes a new carriage in at place 1 (Scratch's place 2), and the others shuffle back. `backpack[3] = "lamp"` uses the "put in the box" sign: carriage 3 (Scratch's item 4) gets a new item, and the old `"map"` is thrown out. Each carriage works like a little box!

## Is it in the backpack?




```
backpack = ["apple", "torch", "map"]
print("torch" in backpack)
print("sword" in backpack)
```

**You should see**

```
True
False
```

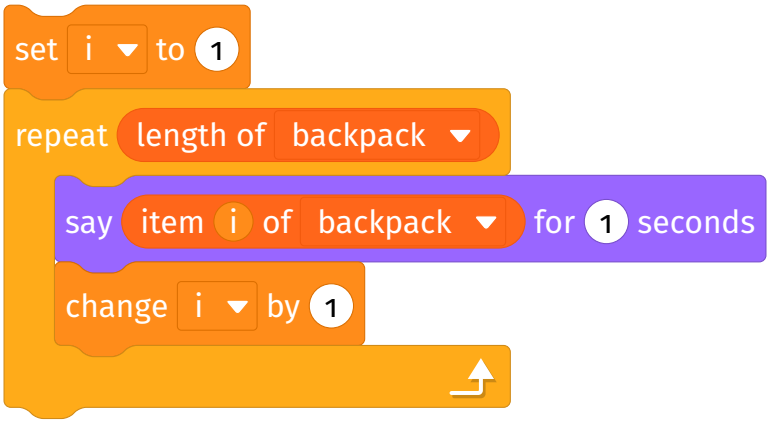
**How it works:** `"sword" in backpack` asks a yes-or-no question. Python answers `True` (yes) or `False` (no), where Scratch says `true` or `false`. In Part 5 you will use questions like this with **`if`**, to do something only when the sword is there.

**Watch out: capitals again**

Scratch thinks `Torch` and `torch` are the same. Python doesn't: `"Torch" in backpack` is `False`. Type your items exactly.

## A Python power: walk through the backpack

In Scratch, to visit every item you need a counter variable, **repeat (length of backpack)** and **item (i) of backpack**. Python has a much shorter way.




```
for thing in backpack:
    print("In my backpack:", thing)
```

**You should see**

```
In my backpack: apple
In my backpack: torch
In my backpack: map
```

**How it works:** `for thing in backpack:` means "take each item out of the backpack in turn, and call it `thing`". The first time round the loop, `thing` holds `"apple"`, then `"torch"`, then `"map"`. You get to choose the name: `for snack in snacks:` reads nicely. No counter, no `- 1`, no mistakes!

## Pick one at random




```
import random
snacks = ["crisps", "banana", "cookie"]
print(random.choice(snacks))
```

**How it works:** Did you know you can type `random` into Scratch's **item 1 of** block? Python's version is `random.choice(snacks)`: it reaches into the list without looking and pulls

out one item. Like `import time` in Part 2, `import random` brings in Python's random tools first. Run it a few times: you won't always get the same snack.

**Watch out: Python tells you when a carriage is missing**

Ask Scratch for **item 4 of backpack** when there are only 3 items, and Scratch quietly gives you nothing. Python stops and tells you:

```
backpack = ["apple", "torch", "map"]
print(backpack[3])
```

```
IndexError: list index out of range
```

**Index** is Python's word for a carriage number. "Out of range" means "there is no carriage with that number". A list of 3 has carriages 0, 1 and 2 only. This error is a clue note, not a telling-off: it shows you exactly which line to fix.

MISSION 8

# Score counter

**You will learn:** making a variable, changing it in a loop, and printing it

Every game needs a score. This program gives you 10 points every second, five times, and shows the score each time, just like a Scratch monitor ticking up.

## Step 1: Build it in Scratch




Click **Make a Variable** and call it **score**. Then build:

## Step 2: Type it in Colab

This one doesn't draw, so you don't need the three starting lines.

**Type this into a code cell**

```
import time
score = 0
print("Score:", score)
for i in range(5):
    time.sleep(1)
    score = score + 10
    print("Score:", score)
print("Game over! Final score:", score)
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| — | when green flag clicked | No code needed. The play button ▶ starts it. |
| 1 | (wait needs this) | Brings in Python's clock tools, so we can use `time.sleep`. |
| 2 | set score to 0 | Makes a box called `score` and puts 0 in it. |
| 3 | show variable score | Shows what is in the box: `Score: 0`. |
| 4 | repeat 5 | Do the indented lines 5 times. |
| 5 | wait 1 seconds | Wait one second. |
| 6 | change score by 10 | Take what's in the box, add 10, and put it back. |
| 7 | (the monitor updates) | Scratch's monitor changes by itself. In Python, we print again to see the new score. |
| 8 | say join Game over! … | No spaces in front, so it runs once, after the loop. |

## Step 4: Run it

Press ▶. A new line appears every second.

**You should see**

```
Score: 0
Score: 10
Score: 20
Score: 30
Score: 40
Score: 50
Game over! Final
score: 50
```

**What if you forget line 2?**

Delete line 2 and click **Runtime**, then **Restart session**, then run the cell again. Python says `NameError: name 'score' is not defined`. You can't add 10 to a box that doesn't exist yet! In Scratch, a new variable starts at 0 by itself. In Python, you always fill the box first.

**Make it yours**

☐ Change line 6 to `score += 10`. Does anything change when you run it?
☐ Make it count **down**: start the score at 50 and take away 10 each time. (Hint: `-=`.)
☐ Add a second box, `lives = 3`, before the loop. Inside the loop, take one life away and print both: `print("Score:", score, "Lives:", lives)`. What happens to the lives after 5 goes?
☐ **Super challenge:** make the points get bigger each time. Add `points = 10` before the loop. Inside it, add `points` to the score, then make `points` 10 bigger. What is the final score?

MISSION 9

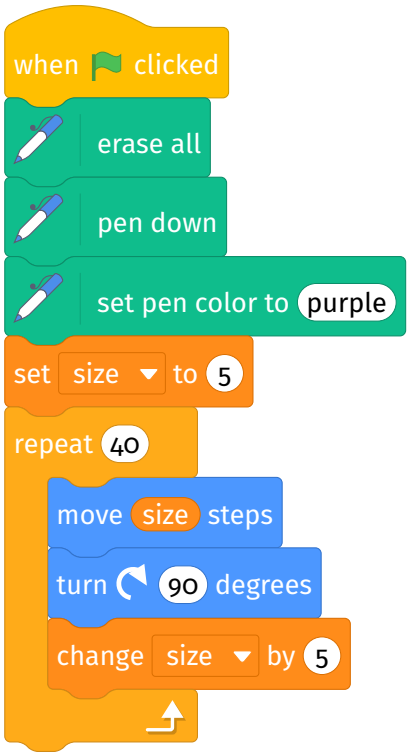
# Growing spiral

**You will learn:** using a variable for the size, and changing it inside a loop

On the “Variables can draw too” page, the size changed once, between two squares. Now the size grows a little after **every** line. Each line is longer than the one before, and the turtle spirals outwards.

## Step 1 and Step 2: Build it in Scratch, then type it in Colab

Make a variable called **size**. Add the **Pen** extension, then build the script on the left. Then type the Python on the right.




**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
speed(13)
pencolor("purple")
size = 5
for i in range(40):
    forward(size)
    right(90)
    size = size + 5
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | when green flag clicked<br>erase all | The three starting lines. `clearscreen()` wipes the stage, like erase all. The pen starts down. |
| 4 | | The fastest speed, so you don't wait too long. |
| 5 | set pen color to purple | Colour names go in quote marks, in small letters. |
| 6 | set size to 5 | The first line will be 5 steps long. |
| 7 | repeat 40 | Do the indented lines 40 times. |
| 8 | move (size) steps | Walk forward by whatever is in the `size` box right now. |
| 9 | turn right 90 degrees | Turn a corner. |
| 10 | change size by 5 | Make the box 5 bigger, **inside** the loop, so the next line is longer. |

## Step 4: Run it

A purple square spiral grows out from the middle of the stage. The lines go 5, 10, 15, 20 ... steps, and the last one is 200 steps long.

**Spaces matter**

Line 10 has four spaces in front, so it is inside the loop. Take the spaces away and run it. Now `size` only changes **once**, after the loop has finished, so every line is 5 steps long and the turtle just goes round a tiny square 40 times. No red error, but the wrong drawing!

**Make it yours**

☐ Change `right(90)` to `right(91)`. The spiral twists! Try 89, 120 and 144 too.
☐ Make the spiral tighter with `size = size + 2`, or looser with `size = size + 8`. With `+ 8`, does it still fit on the stage?
☐ Make the pen thicker as well: add `thick = 1` before the loop, then `pensize(thick)` and `thick = thick + 0.2` inside it.
☐ **Super challenge:** make a spiral that **shrinks**. Start at `size = 200` and take away 5 each time. (Hint: so it fits on the stage, jump to `goto(-100, 100)` with the pen up before you start.)

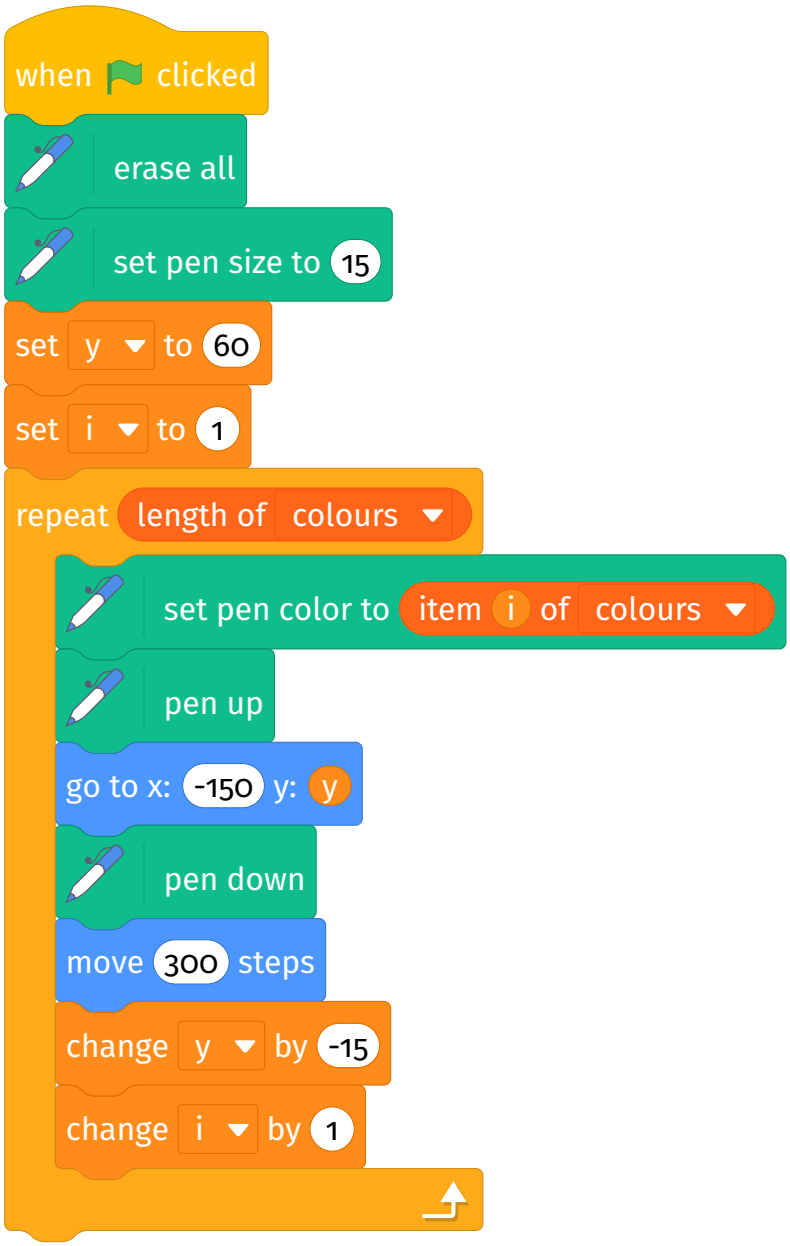
MISSION 10

# Rainbow from a colour list

**You will learn:** a list of colours, and a `for` loop that walks through it

A rainbow is a list of colours! This program keeps the six colours in a list, then draws one thick stripe for each colour, from top to bottom.

## Step 1: Build it in Scratch




Make a list called **colours** and two variables called **y** and **i**. Scratch's pen doesn't understand colour names, so fill the list on the stage with six colour codes: `#FF0000`, `#FFA500`, `#FFFF00`, `#008000`, `#0000FF`, `#800080`. Then build:

## Step 2: Type it in Colab

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
speed(13)
pensize(15)
colours = ["red", "orange", "yellow",
           "green", "blue", "purple"]
y = 60
for colour in colours:
    pencolor(colour)
    penup()
    goto(-150, y)
    pendown()
    forward(300)
    y = y - 15
hideturtle()
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | **when green flag clicked** | The three starting lines. They also do **erase all**. |
| 4–5 | **set pen size to 15** | The fastest speed, and thick stripes. |
| 6–7 | (the list on the stage) | Six colour names. Inside square brackets you may carry on to the next line. |
| 8 | **set y to 60** | The first stripe is at $y = 60$. |
| 9 | **repeat length of colours** | Take out each colour in turn, and call it `colour`. No counter `i`! |
| 10 | **set pen color to** **item i of colours** | Use the colour we just took out. `colour` has no quote marks: it is a box, not the word “colour”. Python understands colour **names**, so no codes are needed. |
| 11–13 | **pen up** **go to x: -150 y: (y)** **pen down** | Jump to the left end of this stripe without drawing. |
| 14 | **move 300 steps** | Draw the stripe. The turtle faces right. |

| | | |
|---|---|---|
| 15 | **change y by -15** | Move down 15 for the next stripe. Then line 16 hides the turtle. |

## Step 4: Run it

Six thick stripes appear one under another: red at the top, then orange, yellow, green, blue and purple at the bottom.

**Colour and `colour`**

In this book we spell it **colour**. Python was written in America, so its own words use **color**: `pencolor`. Our list and box names can use any spelling we like, as long as we always spell them the same way.

**Make it yours**

☐ Add a seventh stripe with `colours.append("pink")` on a new line after line 7.
☐ Leave gaps between the stripes: change `15` in line 15 to `20`.
☐ Make your own flag. Change the list to `["red", "white", "blue"]`. Which countries could you make?
☐ **Super challenge:** make every stripe shorter than the one above it. Add `length = 300` before the loop, use `forward(length)`, and take 30 off `length` each time round.

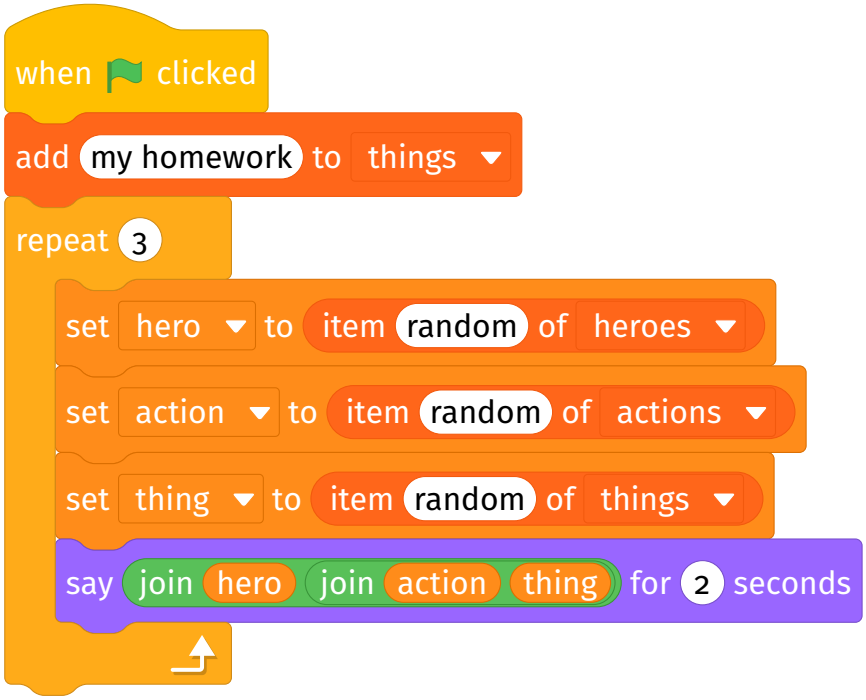
MISSION 11

# Random story maker

**You will learn:** filling lists, adding to them, and picking with `random.choice`

Three lists, one pinch of randomness, and you get a silly sentence every time. Will a dragon eat your homework?

## Step 1: Build it in Scratch




Make three lists, **heroes**, **actions** and **things**, and fill them on the stage. Make three variables, **hero**, **action** and **thing**. Then build:

## Step 2: Type it in Colab

**Type this into a code cell**

```
import random
heroes = ["A dragon", "My teacher", "A tiny robot"]
actions = ["ate", "sang to", "tickled"]
things = ["a banana", "the moon", "a sock"]
things.append("my homework")
for i in range(3):
    hero = random.choice(heroes)
    action = random.choice(actions)
    thing = random.choice(things)
    print(hero, action, thing)
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1 | | Brings in Python's random tools. |
| 2–4 | (lists filled on the stage) | Three lists, each with three items. Each item is a string, so each has quote marks. Commas go between items. |
| 5 | **add my homework to things** | Puts a fourth thing on the end of the `things` list. |
| 6 | **repeat 3** | Make three sentences. |
| 7–9 | **set hero to** **item random of heroes** … | Pick one item from each list, and put it in a box. |
| 10 | **say join hero …** | Print the three boxes in a row. The commas put spaces between them, so you don't need three **join** blocks. |

## Step 4: Run it

You will get three random sentences. Yours will be different!

**You might see**

```
A dragon ate a sock
A tiny robot tickled my homework
My teacher sang to the moon
```

**Click the flag twice**

In Scratch, lists remember their items even after the project stops. Click the green flag twice, and `my homework` is added **twice**! In Python, line 4 makes a fresh `things` list every time you run the cell, so it always starts with exactly three things.

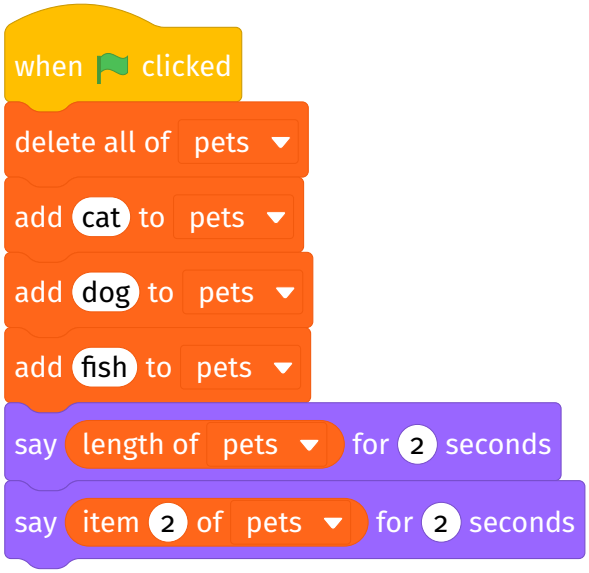
**Make it yours**

☐ Add your own words to each list. Don't forget the quote marks and commas!
☐ Make 5 sentences instead of 3.
☐ Add a fourth list, `places`, like `["in the bath", "on the bus"]`, and print a place at the end of each sentence.
☐ **Super challenge:** after the loop, print `len(things)` and the whole `things` list. How many things are there, and where did `"my homework"` go?

YOUR TURN

# Translate it yourself

You are the translator now! Write the Python for this new Scratch script, then run it in Colab.




**My Python**

**Hints:** What starts an empty list? Which word adds to the end? Item 2 is not `[2]`!

## Two-line puzzle

Can you count the items in a list **without** `len`? Write the **two** missing lines, using a `for` loop and `+=`. It should print 4.

```
words = ["tiger", "lion", "zebra", "hippo"]
count = 0
```

```
print(count)
```

## Part 3 checklist

- ☐ I can make a variable with = and explain that it means “put in the box”.
- ☐ I can change a variable with `score += 1` and show it with `print`.
- ☐ I know the naming rules, and that `5` and `"5"` are different.
- ☐ I can make a list and use `append`, `len`, `del`, `insert` and `clear`.
- ☐ I remember that Scratch item 1 is Python `[0]`, and I know what `IndexError` means.
- ☐ I can walk through a list with `for` and pick at random with `random.choice`.

PART 4

# Operators

Do maths, roll dice, compare things and take words apart.

The green **Operators** blocks are the thinking blocks of Scratch. They do sums, pick random numbers, check whether things are true, and play with words. In this part you will meet the Python version of every one of them:

- **7 + 2** becomes `7 + 2`
- **pick random 1 to 6** becomes `random.randint(1, 6)`
- **join Hello Sam** becomes `"Hello " + "Sam"`
- **letter 1 of banana** becomes `"banana"[0]`

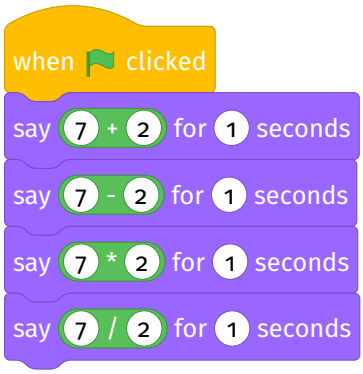
Most of the programs in this part **don't draw**, so you don't need the three starting lines. Just make a new cell with **+ Code**, type, and press ▶.

**A friendly waiter and a strict robot chef**

Scratch is like a friendly waiter. If you ask for something a bit muddled, it guesses what you meant. Python is like a strict robot chef. It does **exactly** what you say, and if your order doesn't make sense, it stops and tells you why. That sounds annoying, but it means Python never quietly guesses wrong. Look out for the **Watch out!** boxes in this part: they show the places where the robot chef is fussier than the waiter.

## Doing sums: + – * /

In Scratch you can drop a green sum block into a say block. In Python you put the sum inside `print( )`.




```
print(7 + 2)
print(7 - 2)
print(7 * 2)
print(7 / 2)
```

**You should see**

```
9
5
14
3.5
```

**How it works.** The signs are the same as in Scratch: `*` (a star) means times, and `/` (a slash) means divide. You don't need the green oval: the numbers and the sign just sit next to each other. Put spaces around the sign so it is easy to read.

**Watch out: divide always gives a decimal**

In Python, `/` always gives a number with a decimal point, even when it divides exactly. `print(6 / 2)` shows `3.0`, not `3`. It is still the right answer!
**Bonus:** two slashes, `//`, divide and throw away the leftover part. `print(7 // 2)` shows `3`. Scratch has no block for this.

**Brackets are the green ovals**

In Scratch, you choose which sum happens first by putting one block **inside** another. In Python you use brackets. Without brackets, Python does times and divide before plus and minus, just like in maths lessons:

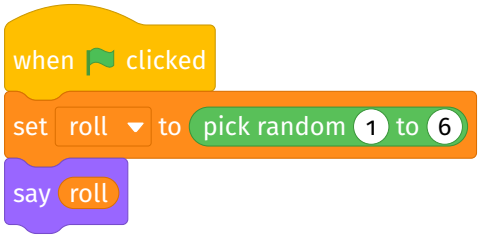



`2 + 3 * 4` gives `14`




`(2 + 3) * 4` gives `20`

## Pick random




```
import random
roll = random.randint(1, 6)
print(roll)
```

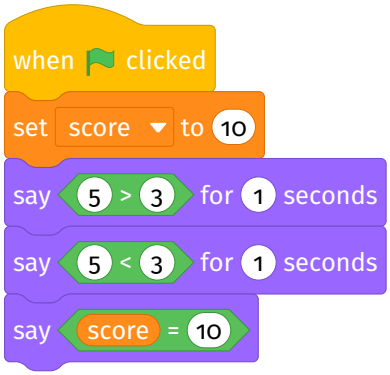
**How it works.** `import random` is like clicking the **Extensions** button: it brings in Python's random-number tools. You only need it once, at the top of the cell. `random.randint(1, 6)` means "from the random tools, pick a whole number (an **int**eger) from 1 to 6". The dot means "from inside". Just like Scratch, **both ends are included**, so you can get 1 and you can get 6. Run the cell a few times: you get a different number each time.

**Watch out: smallest number first**

Scratch doesn't mind **pick random 6 to 1**: it swaps the numbers for you. Python stops with `ValueError: empty range`. Always put the smaller number first.

## Bigger, smaller, the same: > < ==

The pointy green blocks answer a question with **true** or **false**. You can drop one into a **say** block to see the answer. In Python, the answer is **`True`** or **`False`**.




```
score = 10
print(5 > 3)
print(5 < 3)
print(score == 10)
```

**You should see**

```
True
False
True
```

**How it works.** `>` and `<` are the same as in Scratch. To check whether two things are the same, Python uses **two** equals signs, `==`. Remember from Part 3: **one** equals sign puts a value in a box (`score = 10`). Two equals signs ask a question: "is `score` the same as 10?" **`True`** and **`False`** always start with a capital letter.

In Part 5 you will put these questions inside **`if`**, to make the program decide what to do. For now, we just print the answers.

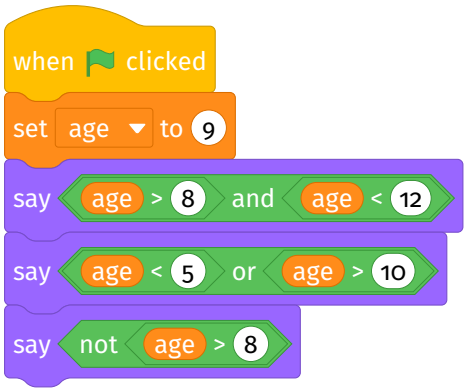
**Two blocks Scratch doesn't have**

Python can also ask "bigger **or the same**" with >=, and "smaller or the same" with <=. `print(10 >= 10)` shows `True`. And `!=` means "is **not** the same as".

**Watch out: capital letters count!**

In Scratch, Cat = cat is **true**. Scratch doesn't care about capital letters. Python does: `print("Cat" == "cat")` shows `False`.
The fix is `.lower()`, which turns writing into small letters before you compare: `print("Cat".lower() == "cat".lower())` shows `True`.

## and, or, not



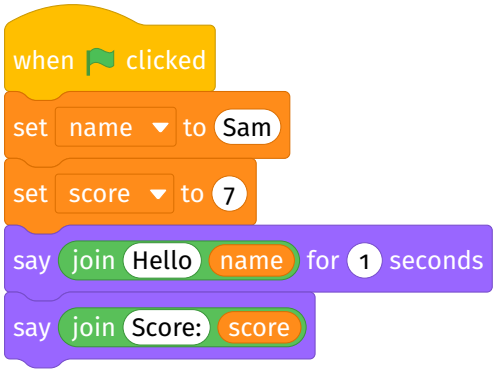

```
age = 9
print(age > 8 and age < 12)
print(age < 5 or age > 10)
print(not age > 8)
```

**You should see**

```
True
False
False
```

**How it works.** Python uses the same words as Scratch, written in small letters. **`and`** is **`True`** only when **both** sides are true. **`or`** is **`True`** when **at least one** side is true. **`not`** flips the answer: true becomes false and false becomes true. You don't need the extra hexagon shapes: Python works out the > and < questions first, then the **`and`**.

## Join: sticking words together




```
1 name = "Sam"
2 score = 7
3 print("Hello " + name)
4 print("Score: " + str(score))
```

**You should see**

```
Hello Sam
Score: 7
```

**How it works.** When `+` has writing on both sides, it joins them, just like **join**. Python calls a piece of writing a **string** (think of letters threaded on a string, like beads). Notice the space inside `"Hello "`: without it you would get `HelloSam`.

**Watch out: you can't join writing and a number**

`print("Score: " + score)` stops with this clue:

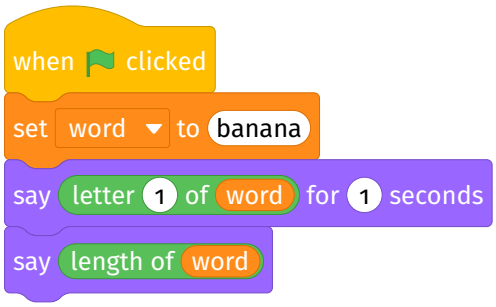
```
TypeError: can only concatenate str (not "int") to str
```

"Concatenate" is a long word for join. Python is saying: "I can only join a string to a string, and you gave me a number." Scratch would just guess. In Python, turn the number into a string first with `str(score)`, like line 4 above.

**Bonus: f-strings, the easy way to join**

Put an `f` in front of the quote marks, and anything inside curly brackets `{ }` is filled in for you, numbers too:
`print(f"Score: {score}")` shows `Score: 7`. No `str( )` and no `+` needed!
And one more trick: `print("ha" * 3)` shows `hahaha`.

## Letter of, and length of




```
word = "banana"
print(word[0])
print(len(word))
```

**You should see**

```
b
6
```

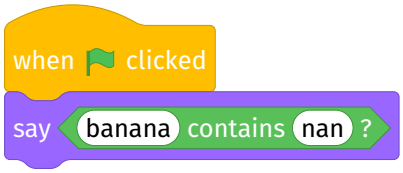
**How it works.** Square brackets after a string pick out one letter, just like picking an item from a list in Part 3. And just like lists, Python **counts from 0**. So Scratch's **letter 1 of banana** is `word[0]` in Python, letter 2 is `word[1]`, and so on. The rule is: **take away 1** from the Scratch number. **`len( )`** is short for *length*: it counts the letters. It works on lists too.

| | | | | | | |
|---|---|---|---|---|---|---|
| Scratch | 1 | 2 | 3 | 4 | 5 | 6 |
| | b | a | n | a | n | a |
| **Python** | **0** | **1** | **2** | **3** | **4** | **5** |

**Watch out: off the end**

`"banana"` has 6 letters, so the last one is `word[5]`. Asking for `word[6]` stops with `IndexError: string index out of range`. (Scratch just gives an empty answer.) **Bonus:** `word[-1]` is a shortcut for the last letter.

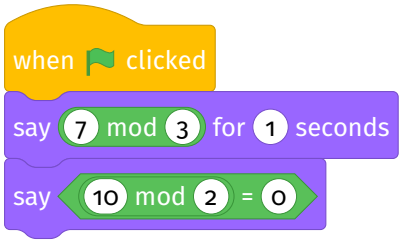
## Contains




```
print("nan" in "banana")
```

**How it works.** It shows `True`. Read it out loud: "is `nan` **in** `banana`?" The order is flipped from Scratch: the small piece comes first, then **`in`**, then the big word. Like `==`, **`in`** cares about capitals: `"A" in "banana"` is **`False`**. Use `.lower( )` again if you need it.

## Mod: the leftover




```
print(7 % 3)
print(10 % 2 == 0)
```

**You should see**

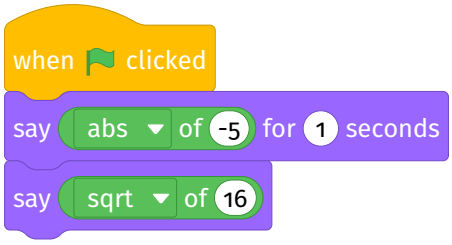
```
1
True
```

**How it works.** In Python, mod is the percent sign, %. It gives the **leftover** after dividing. Share 7 sweets between 3 friends: everyone gets 2, and 1 is left over. So `7 % 3` is 1.

**Even or odd?**

An even number divided by 2 has nothing left over. So `n % 2 == 0` is `True` for even numbers and `False` for odd ones. Programmers use this trick all the time.

## abs, square root and friends




```
import math
print(abs(-5))
print(math.sqrt(16))
```

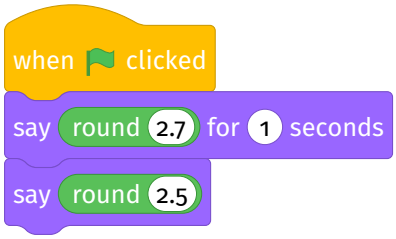
**You should see**

```
5
4.0
```

**How it works.** `abs( )` takes away the minus sign. It is built into Python, so you can use it straight away. The square root lives in Python's maths tools, so first `import math`, then use `math.sqrt( )`. The square root of 16 is 4, because $4 \times 4 = 16$. Like divide, it gives a decimal: `4.0`.

The other choices in that Scratch drop-down are in `math` too: **floor of** is `math.floor( )` (round down) and **ceiling of** is `math.ceil( )` (round up). The angle ones, like `sin`, work a bit differently in Python, so leave them for later.

## Round




```
print(round(2.7))
print(round(2.5))
```

**You should see**

```
3
2
```

**How it works.** `round( )` goes to the nearest whole number, so 2.7 becomes 3. But look at the second answer!

**Watch out: Python rounds 2.5 to 2**

Scratch rounds 2.5 **up** to 3. Python rounds a number that ends in .5 to the nearest **even** number. So `round(2.5)` is 2, but `round(3.5)` is 4. Grown-up programmers do this so that, over lots of numbers, rounding doesn't push the total up.
Want Scratch's way? Use `math.floor(2.5 + 0.5)`, which gives 3.
**Bonus:** `round(3.14159, 2)` keeps 2 decimal places and gives `3.14`.

## Python is fussier than Scratch

Here are all the robot-chef moments from this part in one table. Come back to it when Python shows you a red error.

| You write | Scratch says | Python says | Do this instead |
|---|---|---|---|
| `"Score: " + 10` | Score: 10 | `TypeError` | `"Score: " + str(10)` |
| `"Cat" == "cat"` | true | `False` | `.lower()` on both |
| letter 1 of `"cat"` | c | `"cat"[1]` is a | `"cat"[0]` |
| round 2.5 | 3 | 2 | `math.floor(2.5 + 0.5)` |
| `10 / 0` | Infinity | `ZeroDivisionError` | Never divide by 0 |
| pick random 6 to 1 | a number | `ValueError` | Smaller number first |

## Predict it!

Before you run each line, write down what you think Python will show. Then type all six lines into one cell and run it. How many did you get right? (Answers at the back.)

| Code | I think it shows |
|---|---|
| `print(9 / 3)` | _____ |
| `print("4" + "4")` | _____ |
| `print("Dog" == "dog")` | _____ |
| `print("pizza"[1])` | _____ |
| `print(round(4.5))` | _____ |
| `print(17 % 5)` | _____ |

MISSION 12

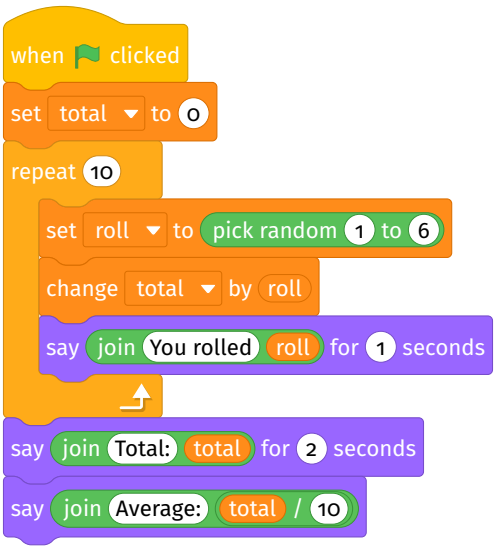
# Dice roller

**You will learn:** `random.randint`, adding up in a loop, and joining numbers with `str( )`

Roll a dice 10 times, add up all the rolls, and work out the average. Is the average close to what you expected?

## Step 1: Build it in Scratch

Make two variables, **roll** and **total**, then build:




## Step 2: Type it in Colab

**Type this into a code cell**

```
import random
total = 0
for i in range(10):
    roll = random.randint(1, 6)
    total = total + roll
    print("You rolled " + str(roll))
print("Total: " + str(total))
print("Average: " + str(total / 10))
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| — | when green flag clicked | No code needed: the play button starts it. |
| 1 | (Extensions) | Brings in the random tools. |
| 2 | set total to 0 | Make a box called `total` with 0 in it. |
| 3 | repeat 10 | Do the indented lines 10 times. |
| 4 | set roll to pick random | Roll the dice: a whole number from 1 to 6, both ends included. |
| 5 | change total by roll | Take what is in `total`, add the roll, and put the answer back in `total`. |
| 6 | say join You rolled roll | `roll` is a number, so `str(roll)` turns it into a string before joining. |
| 7 | say join Total: total | No spaces in front, so this runs once, after the loop. |
| 8 | say join Average: ... | `total / 10` is worked out first, then turned into a string. |

## Step 4: Run it

Press . Your rolls will be different, because they are random! Here is one real run:

**You might see**

```
You rolled 6
You rolled 2
... 8 more rolls: 6, 2, 1, 3, 3, 1, 4, 6 ...
Total: 34
Average: 3.4
```

The average is a decimal because `/` always gives a decimal. If you roll a fair dice lots and lots of times, the average gets close to 3.5, exactly half-way between 1 and 6.

**Make it yours**

☐ Roll 100 times instead of 10. Change line 3 **and** line 8. Is the average closer to 3.5 now? (Tip: take out line 6 first, or you will get 100 lines!)
☐ Make a 20-sided dice for a board game.
☐ Roll **two** dice each time: `roll = random.randint(1, 6) + random.randint(1, 6)`. What is the biggest total you could get?
☐ Add `print(roll == 6)` inside the loop. It shows `True` every time you roll a six.

MISSION 13

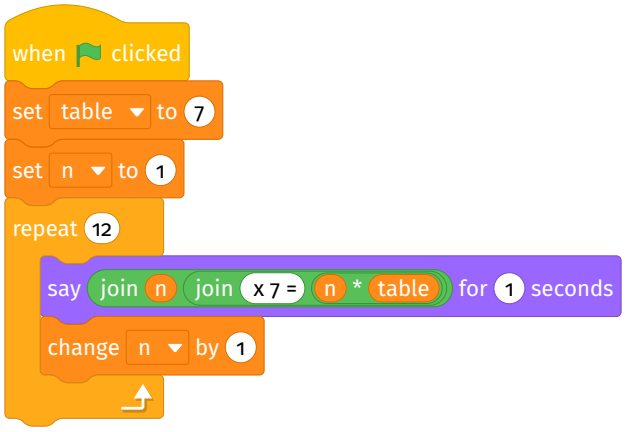
# Times-table printer

**You will learn:** multiplying in a loop, and f-strings

Let Python print a whole times table in a flash. Then change one number to get any table you like.

## Step 1: Build it in Scratch

Make two variables, **table** and **n**, then build:




That's a lot of joins! Python makes this much easier.

## Step 2: Type it in Colab

**Type this into a code cell**

```
table = 7
n = 1
for i in range(12):
    print(f"{n} x {table} = {n * table}")
    n = n + 1
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1 | **set table to 7** | Which times table to print. |
| 2 | **set n to 1** | n counts 1, 2, 3 … up to 12. |
| 3 | **repeat 12** | Do the indented lines 12 times. |
| 4 | **say join …** | An **f-string**. Each pair of curly brackets is a gap that Python fills in. `{n * table}` even does the sum first! The x is just a letter x in the writing. |
| 5 | **change n by 1** | Add 1 to n, ready for the next line. |

## Step 4: Run it

**You should see**

```
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
```

*… and so on, down to …*

```
12 x 7 = 84
```

**The same line without an f-string**

You could also join with `+` and `str( )`, like Mission 1. It works, but it is long, so it is easier in three lines:

```
a = str(n) + " x "
b = str(table) + " = "
print(a + b + str(n * table))
```

Both ways are right. Most Python coders choose the f-string.

**Make it yours**

☐ Print the 9 times table. Which line do you change?
☐ Go all the way up to 20 x 7.
☐ Print square numbers: change line 4 so it shows `5 x 5 = 25` and so on. (Hint: you only need n.)
☐ Add `{n * table % 2 == 0}` at the end of the f-string. What does the `True` or `False` tell you?

MISSION 14

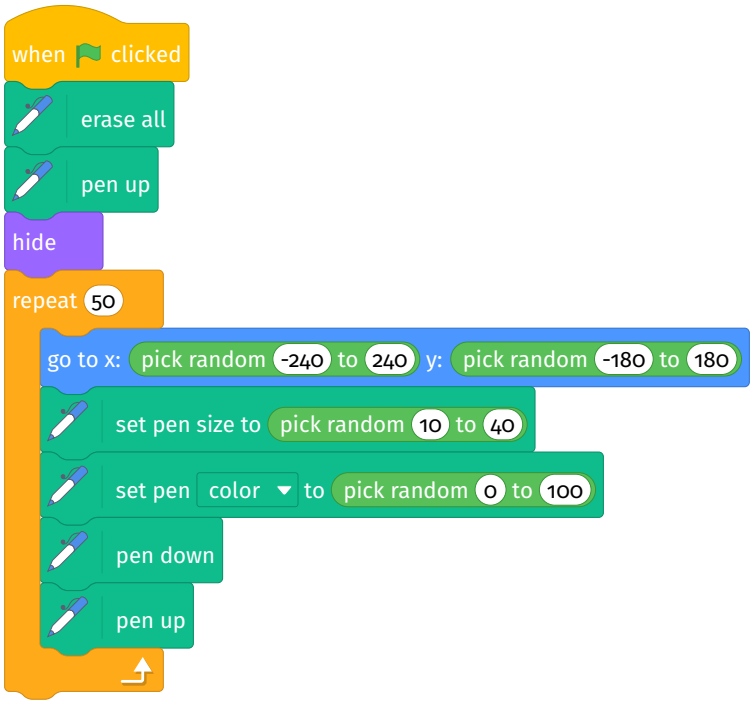
# Random dot art

**You will learn:** random positions with `goto`, `dot( )`, and picking a random colour from a list

The turtle jumps to 50 random places on the stage and leaves a coloured dot at each one. Every time you run it, you get a new picture!

## Step 1: Build it in Scratch

Add the **Pen** extension, then build:



### Step 2: Type it in Colab

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
import random

colours = ["red", "gold", "skyblue", "pink", "lime"]
speed(13)
penup()
hideturtle()
for i in range(50):
    x = random.randint(-240, 240)
    y = random.randint(-180, 180)
    goto(x, y)
    size = random.randint(10, 40)
    colour = colours[random.randint(0, 4)]
    dot(size, colour)
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | **when green flag clicked**<br>**erase all** | The three starting lines. They also wipe the stage clean. |
| 4 | (Extensions) | Bring in the random tools. |
| 6 | (a list) | A list of 5 colour names. Scratch pen colours are numbers from 0 to 100, but Python likes colour **names**, so we keep our favourites in a list. |
| 7 | | Draw as fast as the turtle can go. |
| 8–9 | **pen up**<br>**hide** | Don't draw lines between the dots, and hide the turtle. |
| 10 | **repeat 50** | Make 50 dots. |
| 11–13 | **go to x: ... y: ...** | Pick a random x from one side of the stage to the other, and a random y from bottom to top. Then go there. |
| 14 | **set pen size to pick random** | A random dot size from 10 to 40. |
| 15 | **set pen color to pick random** | Pick a random place in the list. The list has 5 colours, but Python counts from 0, so the places are 0, 1, 2, 3 and 4. That's why it is `randint(0, 4)` and not `randint(1, 5)`. |
| 16 | **pen down**<br>**pen up** | In Scratch, pen down then pen up makes a dot. `dot(size, colour)` draws a round dot right where the turtle is. |

## Step 4: Run it

You should see 50 dots of different sizes in red, gold, sky blue, pink and lime, scattered all over the stage. Run it again: the picture is different every time.

**Make it yours**

☐ Add `bgcolor("black")` after line 3 for a night sky. Make 200 dots.
☐ Add more colours to the list, such as `"orange"` and `"purple"`. Line 15 only picks places 0 to 4, so it will never pick them until you fix it. (Clever way: `random.randint(0, len(colours) - 1)` always fits the list.)
☐ Make all the dots fall in the top half of the stage. Which numbers in line 12 do you change?
☐ **Bonus:** `random.choice(colours)` picks a random item from a list in one go. Try it in line 15.

MISSION 15

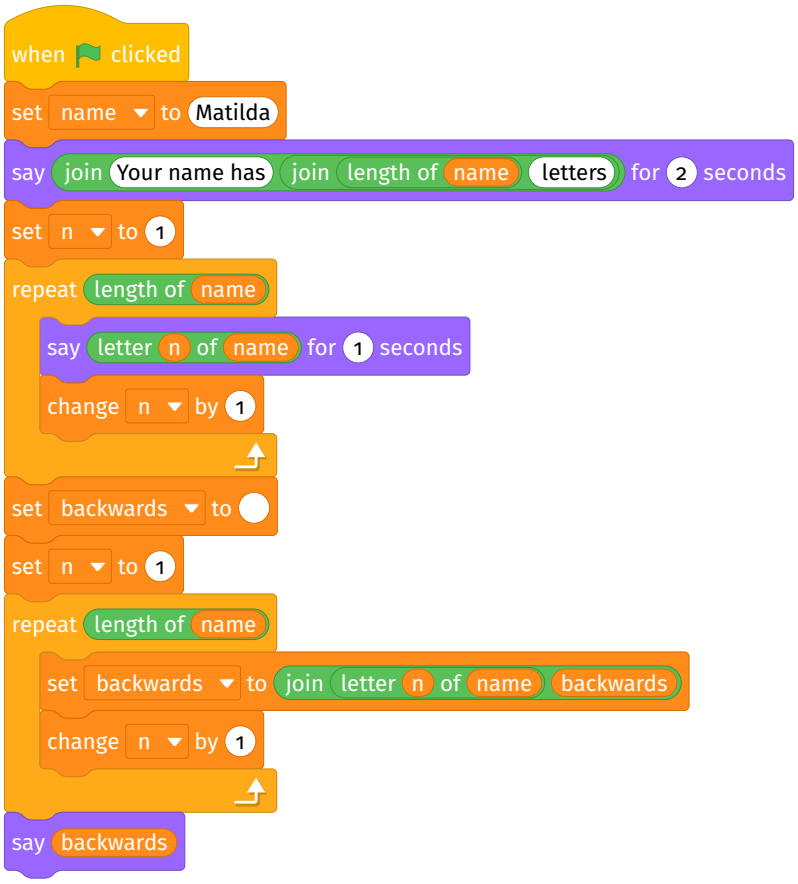
# Name machine

**You will learn:** taking words apart with letters and `len( )`, and building a word backwards

Python takes a name, counts its letters, spells it out one letter per line, and then writes it backwards.

## Step 1: Build it in Scratch

Make three variables, **name**, **n** and **backwards**, then build:



## Step 2: Type it in Colab

Put your own name in line 1!

**Type this into a code cell**

```
1 name = "Matilda"
2 print("Your name has " + str(len(name)) + " letters")
3 for i in range(len(name)):
4     print(name[i])
5
6 backwards = ""
7 for i in range(len(name)):
8     backwards = name[i] + backwards
9 print(backwards)
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1 | **set name to Matilda** | The name goes in quote marks, because it is a string. |
| 2 | **say join Your name has …** | `len(name)` counts the letters. `str( )` turns that number into a string, so it can be joined. |
| 3 | **repeat length of name** | Go round once for every letter. |
| 4 | **say letter n of name** | Here is the clever part. The loop's `i` counts 0, 1, 2, 3 … and those are exactly the numbers Python uses for letters! So Python doesn't need n or **change n by 1**. |
| 6 | **set backwards to (empty)** | Two quote marks with nothing between them make an empty string, like an empty box. |
| 7 | **repeat length of name** | Go through the letters again. |
| 8 | **set backwards to join …** | Stick each new letter on the **front**. M, then aM, then taM … until the whole name is backwards. |
| 9 | **say backwards** | Show the backwards name, once, after the loop. |

### Step 4: Run it

**You should see**

```
Your name has 7 letters
M
a
t
i
l
d
a
adlitaM
```

**Spaces are letters too**

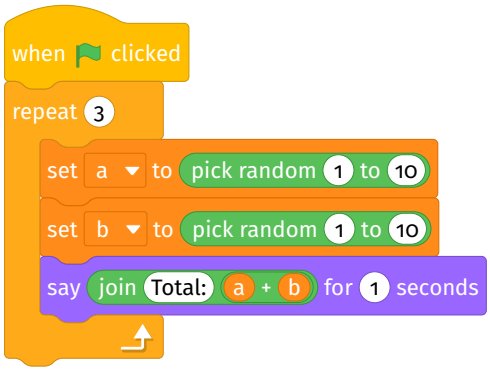
Try `name = "Ada Lovelace"`. Python says it has 12 letters, because it counts the space too. Scratch's **length of** does the same.

**Make it yours**

☐ Print the name in capitals with `print(name.upper())`.
☐ Print the first and the last letter: `name[0]` and `name[-1]`.
☐ Number the letters: change line 4 to `print(str(i + 1) + ": " + name[i])`. Why does it say `i + 1`?
☐ Does your name contain an a? Add `print("a" in name.lower())`.

## Part 4: Translate it yourself

Here is a Scratch script you haven't seen. Write it in Python on the lines, then type it into Colab and run it. Does it do the same as the Scratch script? (Answer at the back.)




**My Python**

**Two clues:** what do you need at the very top before you can pick random numbers? And can you join `"Total: "` straight to a number?

### Two-line puzzle

The turtle's secret password is the **first letter** of `Turtle` joined to the **number of letters** in `Turtle`. Write **two lines** of Python that work out the password and print it. Don't type the answer in yourself: make Python work it out!
When it works, you should see:

**You should see**

```
T6
```

## Part 4 checklist

☐ I can do sums with `+ - * /`, and I know `/` always gives a decimal.

☐ I can pick a random number with `random.randint( )`, after `import random`.

☐ I can compare with `> < ==` and **`and`**, **`or`**, **`not`**, and print **`True`** or **`False`**.

☐ I can join strings with `+`, and use **`str`**`( )` or an f-string for numbers.

☐ I can get a letter with `word[0]`, count letters with **`len`**`( )`, and check with **`in`**.

☐ I know where Python is fussier than Scratch: **`round`**`(2.5)`, capitals and counting from 0.

PART 5

# Sensing and Decisions

Your programs will listen to you, watch the clock and make choices.

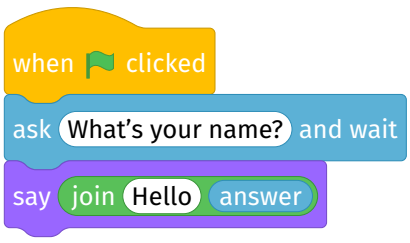
Until now your programs did the same thing every time you pressed ▶. In this part they start to **sense** things and **decide** what to do. Sensing blocks are the sprite's eyes, ears and watch. In Python, `input()` is the program's ears, `distance()` is its measuring tape, and `time` is its stopwatch.

You will meet these Scratch blocks and their Python twins:

- **ask and wait** and **answer** become `input()`
- **if then**, **if then else** and **repeat until** become `if`, `else` and `while`
- **timer** and **current year** become `time` and `datetime`
- **touching?** and **if on edge, bounce** become a little bit of maths with `distance()`, `xcor()` and `if`

None of these programs need the turtle until near the end, so most of them start straight away with no starting lines.

## Ask a question and wait




```
name = input("What's your name? ")
print("Hello " + name)
```

**How it works.** `input("...")` does the job of **ask and wait**. It shows the question, then a little box appears **under the cell**. Click in the box, type, and press **Enter**. Whatever you typed comes out of `input()` and is put into the variable on the left of the `=`. Scratch keeps the reply in a block called **answer**; in Python **you** choose the name of the box, so here it is called `name`.

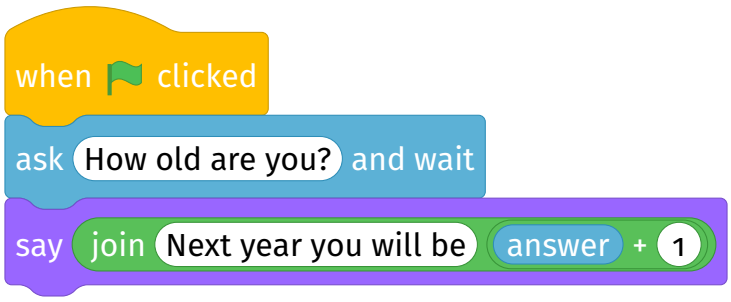
**You should see**

```
What's your name? Ada
Hello Ada
```

**Leave a space at the end**

Look at the space after the question mark in `"What's your name? "`. It stops your typing from sticking to the question, like `What's your name?Ada`.

## Input always gives you writing




```
age = input("How old are you? ")
age = int(age)
print("Next year you will be " + str(age + 1))
```

**How it works.** Whatever you type into `input()` is **writing** to Python, even if it looks like a number. `"9"` is writing; `9` is a number. Scratch quietly turns the writing into a number for you. Python wants you to say so: `int(age)` turns the writing `"9"` into the number 9. (`int` is short for *integer*, which means a whole number.) Then `str( )` turns the answer back into writing so it can be joined with `+`, just as you did in Part 4.

You can do both jobs in one line, and you will see this a lot:

```
age = int(input("How old are you? "))
```

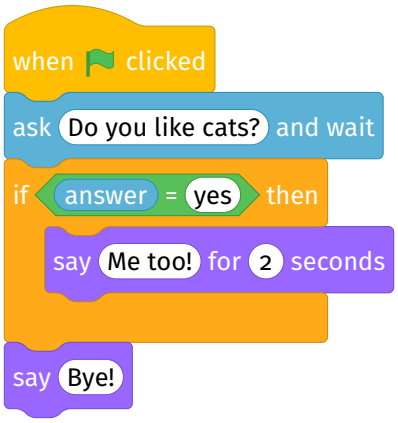
**What if you type a word?**

Run the program and type `nine` instead of 9. Python can't turn `nine` into a number, so it stops with a clue note. Read the **last line** first:

```
ValueError: invalid literal for int() with base 10: 'nine'
```

It means: "`int()` was given `'nine'`, and that isn't a number I can read." Nothing is broken. Press ▶ again and type digits.

## If this, then that



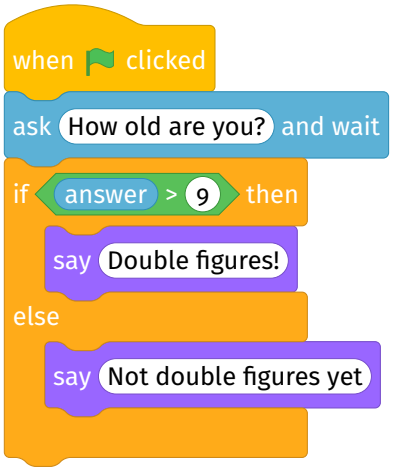

```
answer = input("Do you like cats? ")
if answer == "yes":
    print("Me too!")
print("Bye!")
```

**How it works.** `if` works just like the C-shaped **if then** block. The **colon** at the end opens the mouth of the C, and the lines with **four spaces** in front go inside it, exactly like a `for` loop. Python checks the question once. If it is true, it runs the inside lines; if not, it skips them. `print("Bye!")` is back at the left edge, so it is **under** the C and always runs. Remember: **two** equals signs `==` **check**; one `=` **sets**.

**Capitals count**

Type `Yes` with a capital Y. Python says only `Bye!`, because `"Yes"` and `"yes"` are different to Python. Scratch thinks they are the same! You will learn the fix, `.lower()`, in Mission 3.

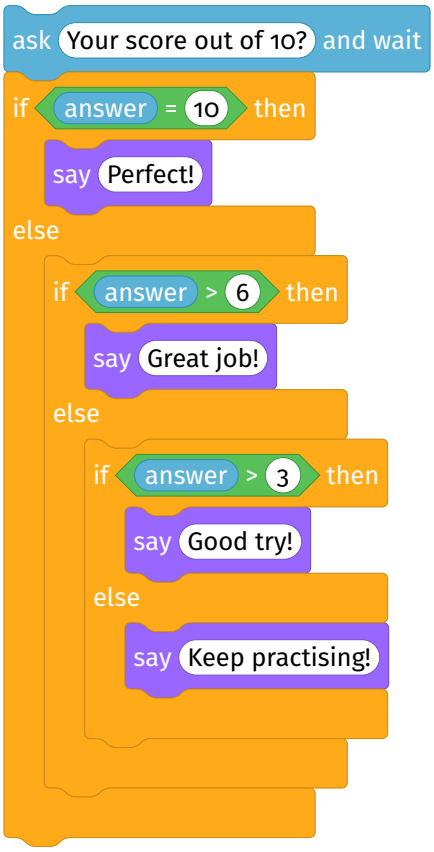
## If this, else that




```
age = int(input("How old are you? ")
    )
if age >= 10:
    print("Double figures!")
else:
    print("Not double figures yet")
```

**How it works.** `else:` is the second mouth of the **if then else** block. It means "otherwise". `else` lines up with `if` (no spaces in front) and needs its own colon. Exactly **one** of the two mouths runs, never both. Scratch has no "bigger or equal" block, so Scratch uses `> 9`; Python can say `>= 10` ("10 or more"). Both mean the same thing for whole numbers.

## A Python shortcut: elif



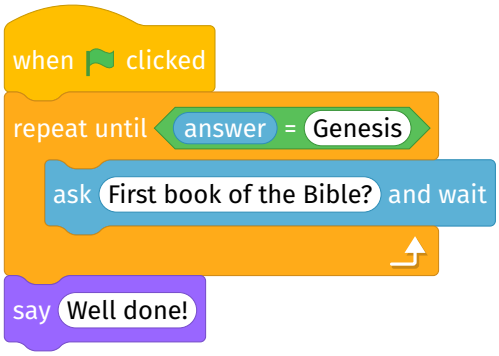

```
score = int(input("Your score out of 10? "))
if score == 10:
    print("Perfect!")
elif score >= 7:
    print("Great job!")
elif score >= 4:
    print("Good try!")
else:
    print("Keep practising!")
```

**How it works.** In Scratch, to choose between more than two things you put an **if then else** inside the **else** mouth of another one, and the script gets wider and wider. Python has a shortcut that Scratch doesn't: `elif`, short for "else if". Python checks the questions **from the top** and runs only the **first** one that is true, then jumps past all the rest. If none is true, the `else` part runs. All the `if`, `elif` and `else` lines line up, so the code stays neat and narrow.

**Why does 8 not say "Good try!" too?**

8 is bigger than 4, so `score >= 4` is true. But Python already found `score >= 7` was true, ran `Great job!`, and skipped everything below it. Only one part ever runs.

## Repeat until: Python flips the question




```
answer = ""
while answer != "Genesis":
    answer = input("First book of the Bible? ")
print("Well done!")
```

**How it works.** Python has no "repeat until". It has `while`, which means "keep going **as long as** this is true". So you flip the question. Scratch says repeat **until** the answer **is** "Genesis". Python says keep going **while** the answer is **not** "Genesis". `!=` means "is not equal to". It is like "stay outside until it rains" and "stay outside while it isn't raining": the same plan, said the other way round.

You can also flip it with the word `not`, which matches the green **not** block:

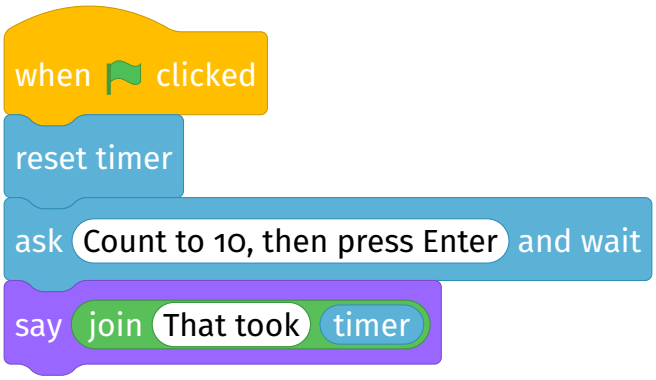
```
while not answer == "Genesis":
```

Both lines do the same thing. Most Python coders use `!=` because it is shorter.

**Why `answer = ""` on line 1?**

`while` checks its question **before** the first go, just like **repeat until**. Python needs a box called `answer` to look in. `""` is empty writing: a box with nothing in it yet. It is not "Genesis", so the loop starts.

## Your own stopwatch: the timer




```
import time
start = time.time()
input("Count to 10 in your head, then press Enter ")
taken = time.time() - start
print("That took " + str(round(taken, 1)) + " seconds")
```

**How it works.** Python has no timer that starts by itself. Instead, `time.time()` tells you what time it is right now, as a huge number of seconds. (It counts from 1 January 1970!) So you make your own stopwatch: write down the time at the start (`start = time.time()`, that is **reset timer**), and later take it away from the new time. `time.time() - start` is the **timer** block: the seconds since you started. **`round`**`(taken, 1)` keeps one number after the dot.

**Reset the timer**

To reset your stopwatch, just do `start = time.time()` again. You can have lots of stopwatches at once, too: `start1`, `start2`... Scratch has only one timer.

## Wait until



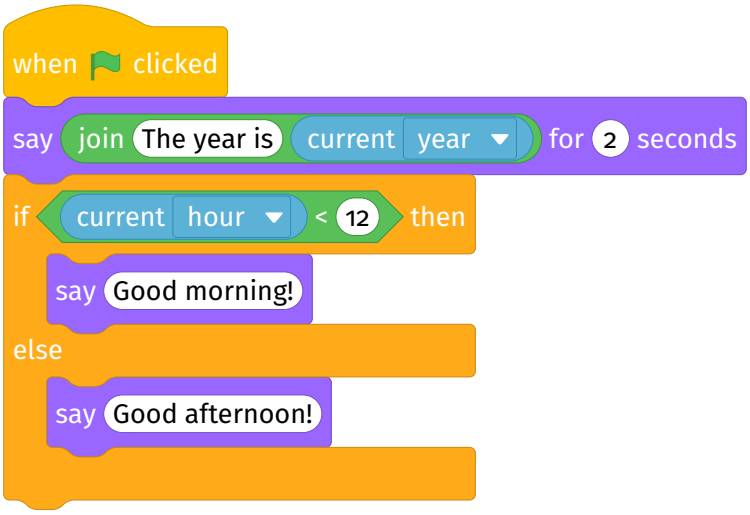

```
1 import time
2 start = time.time()
3 print("Wait for it...")
4 while not time.time() - start > 3:
5     time.sleep(0.1)
6 print("3 seconds are up!")
```

**How it works.** **wait until** is a `while not` loop that does nothing but wait. Each time round, `time.sleep(0.1)` has a tiny nap of a tenth of a second, then Python checks the question again. When the timer passes 3, the question `not ... > 3` is false, the loop ends, and line 6 runs. The nap stops the computer from asking "are we there yet?" millions of times a second.

**Something must change**

In Scratch, another sprite might make the question true. In Python only one script runs, so a `while` loop waits for ever if nothing inside it can change the answer. Waiting for the clock, or for an `input()`, is safe. If a cell runs for ever, press ■.

## What year is it? What time is it?




```
import datetime
now = datetime.datetime.now()
print("The year is " + str(now.year))
if now.hour < 12:
    print("Good morning!")
else:
    print("Good afternoon!")
```

**How it works.** `import datetime` brings in Python's calendar tools. `datetime.datetime.now()` reads the clock once and keeps the answer in `now`. Then `now.year`, `now.month`, `now.day`, `now.hour`, `now.minute` and `now.second` work like the menu in the **current year** block. (Yes, `datetime` is written twice: the first is the toolbox, the second is the tool inside it.)

**Colab's clock may be in another country**

Your code runs on a Google computer, which is often set to **UTC** (London's winter time), not your home time. If Colab says `Good afternoon!` at breakfast, your code is fine: the computer is just somewhere else in the world. The year and the date are almost always right.

## Touching? Use a measuring tape

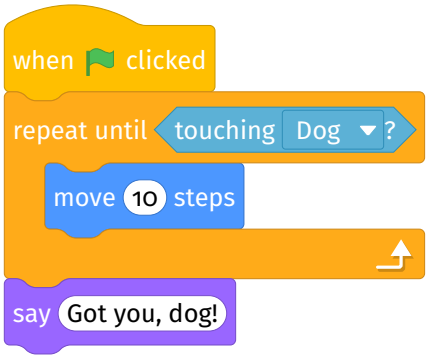
Now the turtle comes back. Scratch checks whether two pictures overlap. The turtle can't see pictures, but it can **measure**: `distance(x, y)` tells you how many steps away a spot is. If it is very close, say less than 20 steps, we say it is touching.




```
1  from ColabTurtlePlus.Turtle import *
2  clearscreen()
3  initializeTurtle((480, 360))
4  dot(20, "red")
5  penup()
6  goto(-200, 0)
7  pendown()
8  while distance(0, 0) >= 20:
9      forward(10)
10 print("Touched it!")
```

**How it works.** Line 4 draws a red spot in the middle, at $(0, 0)$. `distance(0, 0)` measures from the turtle to that spot. "Repeat **until** touching" flips into "keep going **while** the distance is 20 or more". It's like pretending each sprite has an invisible bubble around it.

To check whether two **sprites** touch, make a second turtle with `Turtle()` and measure to where it is. `dog.pos()` gives the dog's position, its $x$ and $y$ together.



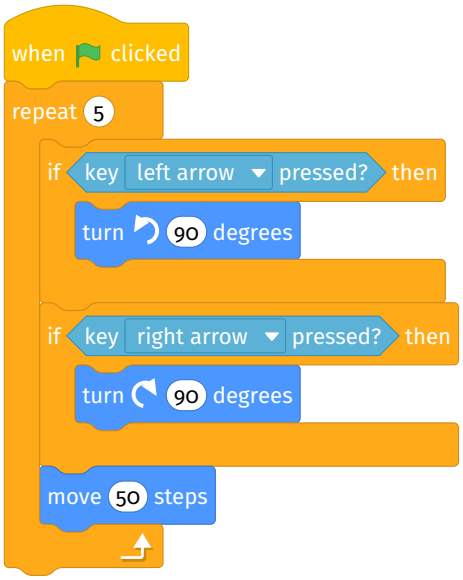

```
1  from ColabTurtlePlus.Turtle import *
2  clearscreen()
3  initializeTurtle((480, 360))
4  shape("turtle")
5  dog = Turtle()
6  dog.shape("circle")
7  dog.penup()
8  dog.goto(150, 0)
9  while distance(dog.pos()) >= 20:
10     forward(10)
11 print("Got you, dog!")
```

**How it works.** `dog = Turtle()` makes a new sprite and gives it the name dog. Commands with `dog.` in front move only the dog; plain commands move the first turtle, as always. Line 9 measures from the turtle to the dog's position. If you make two named sprites, the check looks like `cat.distance(dog.pos()) < 20`.

**Don't forget `.pos()`**

In ColabTurtlePlus you must measure to the dog's **position**, `dog.pos()`. Writing `distance(dog)` gives a `ValueError`, because the measuring tape wants numbers, not a whole sprite.

## Key pressed? Mouse down? Take turns instead




```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
shape("turtle")
for i in range(5):
    key = input("f, l or r? ")
    if key == "l":
        left(90)
    elif key == "r":
        right(90)
    forward(50)
```

**How it works.** Colab's turtle can't hear the keyboard or feel the mouse, so **key pressed?** and **mouse down?** don't work there. Instead we take **turns**: each time round the loop, **`input`**`()` waits for you to type a letter and press Enter, and **`if`** and **`elif`** decide what to do. Any other letter, like `f`, just goes forward. (Real key-press games are coming later, in Thonny.)

## Make your own “if on edge, bounce”




```
1  from ColabTurtlePlus.Turtle import *
2  clearscreen()
3  initializeTurtle((480, 360))
4  speed(13)
5  setheading(30)
6  for i in range(300):
7      forward(10)
8      if xcor() > 230 or xcor() < -230:
9          setheading(180 - heading())
10     if ycor() > 170 or ycor() < -170:
11         setheading(0 - heading())
```

**How it works.** The turtle has no bounce block and no walls: it would happily walk off the stage. So we build the walls with `if`. `xcor()` is the turtle’s $x$ position. If it is past 230 (the right wall) or past $-230$ (the left wall), we turn it round. `heading()` tells you which way the turtle points right now. Line 5 uses the 90 minus rule: Scratch direction 60 is Python `90 - 60`, which is 30.

### The bounce maths

A ball bouncing off a side wall keeps going up (or down) but swaps right for left. That is `180 - heading()`. Bouncing off the floor or ceiling keeps going right (or left) but swaps up for down. That is `0 - heading()`.

| Pointing | Hits | New heading |
|---|---|---|
| 30 (up and right) | right wall | $180 - 30 = 150$ (up and left) |
| 150 (up and left) | ceiling | $0 - 150 = -150$ (down and left) |
| $-150$ (down and left) | left wall | $180 - (-150) = 330$ (down and right) |

MISSION 16

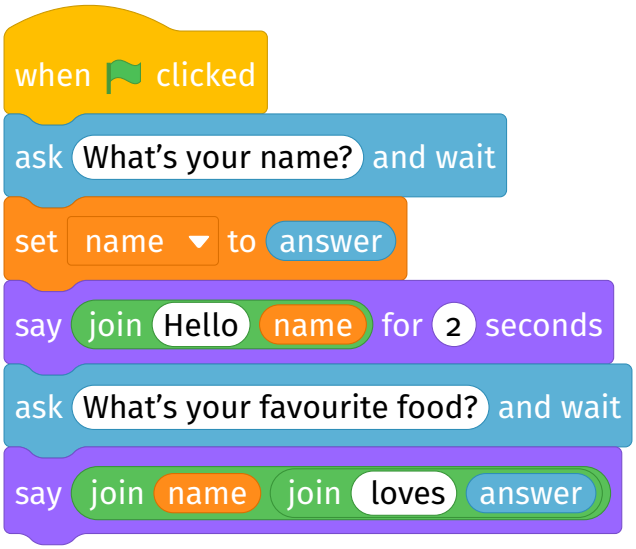
# Say hello

**You will learn:** `input`, joining words, and keeping more than one answer

The computer asks your name and your favourite food, then tells everyone what you love.

## Step 1: Build it in Scratch

Make a variable called **name**, then build:




Why the **name** variable? Scratch has only one **answer**. The second question wipes out the first answer, so you have to save the name somewhere safe first.

## Step 2: Type it in Colab

This one doesn't draw, so you don't need the starting lines.

**Type this into a code cell**

```
name = input("What's your name? ")
print("Hello " + name)
food = input("What's your favourite food? ")
print(name + " loves " + food + "!")
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| — | when green flag clicked | No code needed. The play button starts it. |
| 1 | ask and wait<br>set name to answer | Two blocks in one line! `input()` asks and waits, and the reply goes straight into the box called `name`. |
| 2 | say join Hello name | `+` joins writing together, like join. |
| 3 | ask and wait | A second question with its own box, `food`. Python can keep as many answers as you like. |
| 4 | say join name join … | Four pieces joined in a row. Scratch needs a join inside a join; Python just adds another `+`. |

## Step 4: Run it

Press . Type your name in the box and press Enter, then type a food and press Enter.

**You should see**

```
What's your name? Ada
Hello Ada
What's your favourite food? pizza
Ada loves pizza!
```

**Did you notice?**

Look at the spaces inside `" loves "`: one before and one after. Without them you would get `Adalovespizza!`

**Make it yours**

☐ Ask a third question, like your favourite animal, and add it to the last sentence.
☐ Ask `age = int(input("How old are you? "))` and print how old you will be in 10 years. Don't forget `str()` when you join a number!
☐ Use an `if` to give a special reply when the food is `"broccoli"`.
☐ **Super challenge:** leave out the `int()` in the age question and run it. Read the last line of the red writing. Which kind of error is it?

MISSION 17

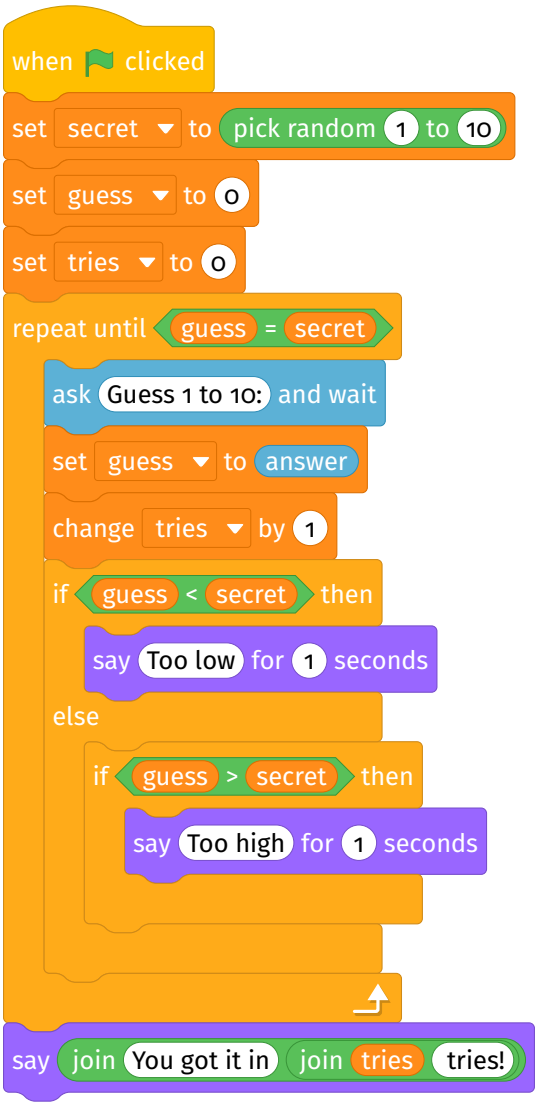
# Guess my number

**You will learn:** `random`, `while`, `if` and `elif`, `int`, and counting tries

The computer picks a secret number from 1 to 10. You keep guessing until you get it. It tells you “too low” or “too high”, and at the end it tells you how many tries you took.

## Step 1: Build it in Scratch

Make three variables called **secret**, **guess** and **tries**, then build:



## Step 2: Type it in Colab

**Type this into a code cell**

```
1  import random
2  secret = random.randint(1, 10)
3  guess = 0
4  tries = 0
5  while guess != secret:
6      guess = int(input("Guess 1 to 10: "))
7      tries += 1
8      if guess < secret:
9          print("Too low")
10     elif guess > secret:
11         print("Too high")
12 print("You got it in " + str(tries) + " tries!")
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1 | (Extensions) | Brings in Python's random number tools. |
| 2 | **set secret to pick random** | `random.randint(1, 10)` picks a whole number from 1 to 10. It can pick 1 or 10. |
| 3–4 | **set guess to 0**<br>**set tries to 0** | Make the two boxes. `guess` starts at 0, which can never be the secret, so the loop is sure to start. |
| 5 | **repeat until guess = secret** | Flipped! Keep going **while** the guess is **not** the secret. |
| 6 | **ask**<br>**set guess to answer** | Ask, turn the writing into a number with **`int( )`**, and keep it in `guess`. |
| 7 | **change tries by 1** | `tries += 1` is the short way to write `tries = tries + 1`. |
| 8–9 | **if guess < secret then** | Too small? Say so. |
| 10–11 | **else** **if guess > secret then** | **`elif`** means "otherwise, if". If the guess is neither too low nor too high, nothing is printed, because it must be right! |
| 12 | **say join You got it in …** | No spaces in front, so it runs once, after the loop. **`str`**`(tries)` turns the number into writing so it can be joined. |

### Step 4: Run it

Press ▶ and type a guess into the box. Press Enter after each one.

**You might see**

```
Guess 1 to 10: 5
Too low
Guess 1 to 10: 8
Too high
Guess 1 to 10: 7
You got it in 3 tries!
```

**Can you always win in 4 tries?**

Always guess the middle of the numbers that are left. 1 to 10: guess 5. Too low? Now it's 6 to 10: guess 8. This trick is called a **binary search**, and real programmers use it all the time.

**Make it yours**

☐ Make it harder: pick a number from 1 to **100**. Change the question too!
☐ Change the last line so it says `try` (not `tries`) when you get it first time. Use `if tries == 1:` and `else:`.
☐ Give the player only 5 tries. Change line 5 to `while guess != secret and tries < 5:`. After the loop, use `if`/`else` to say whether they won or ran out, and tell them the secret.
☐ Type `cat` as a guess. Read the last line of the red writing. What went wrong?

MISSION 18

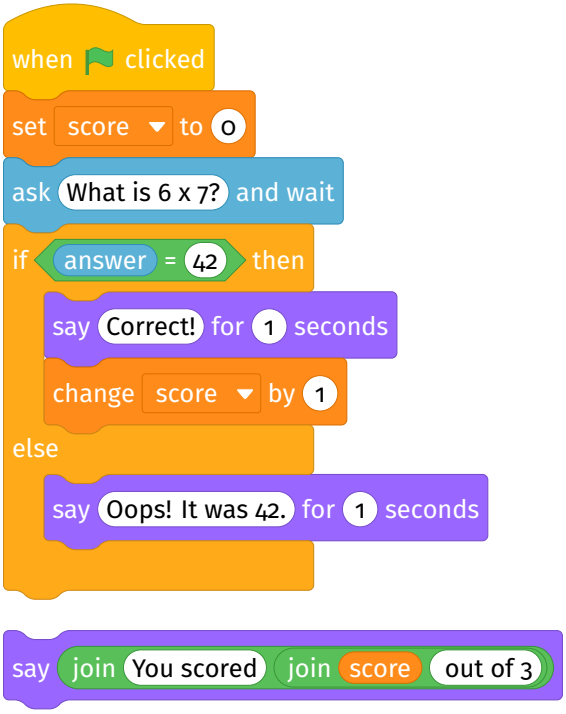
# Quiz with a score

**You will learn:** `if/else`, `.lower()` and keeping score with `+=`

Make a three-question quiz. Each right answer adds one to your score, and at the end the computer tells you how you did.

## Step 1: Build it in Scratch

Make a variable called **score**. Here is the first question. Questions 2 and 3 are built the same way, one under the other.



## Step 2: Type it in Colab

**Type this into a code cell**

```
score = 0

answer = int(input("What is 6 x 7? "))
if answer == 42:
    print("Correct!")
    score += 1
else:
    print("Oops! It was 42.")

answer = input("What colour is a banana? ")
if answer.lower() == "yellow":
    print("Correct!")
    score += 1
else:
    print("Oops! It was yellow.")

answer = input("Which animal says moo? ")
if answer.lower() == "cow":
    print("Correct!")
    score += 1
else:
    print("Oops! It was a cow.")

print("You scored " + str(score) + " out of 3")
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1 | **set score to 0** | The score starts at 0. Blank lines (2, 9, 16, 23) don't do anything; they just make the code easier to read. |
| 3 | **ask and wait** | The answer should be a number, so `int( )` turns it into one. |
| 4 | **if answer = 42 then** | Is the number 42? Two equals signs to check. |
| 5–6 | **say Correct!**<br>**change score by 1** | Both lines are inside the `if`. `score += 1` adds one to the score. |
| 7–8 | **else** | Otherwise, tell them the right answer. |

| | | |
|---|---|---|
| 10 | ask | A word answer this time, so no `int( )`. We reuse the box `answer`; the new reply replaces the old one, just like in Scratch. |
| 11 | if answer = yellow then | `answer.lower()` turns the reply into small letters first, so `Yellow`, `YELLOW` and `yellow` all count. |
| 12–15 | | The same pattern as lines 5 to 8. |
| 17–22 | | Question 3, built the same way. |
| 24 | say join You scored ... | At the left edge, so it always runs at the end. |

**Why `.lower()`?**

Scratch thinks `Yellow` and `yellow` are the same. Python doesn't: capital letters count! `.lower()` makes a small-letter copy of the writing, so it doesn't matter how the player types. Make sure the word you check against, like `"yellow"`, is in small letters too, or it can never match.

## Step 4: Run it

**You might see**

```
What is 6 x 7? 42
Correct!
What colour is a banana? Yellow
Correct!
Which animal says moo? sheep
Oops! It was a cow.
You scored 2 out of 3
```

**Make it yours**

☐ Change the questions to be about something you love: football, dinosaurs, space...
☐ Add a fourth question. Remember to change 3 to 4 on the last line.
☐ At the end, use **`if`**, **`elif`** and **`else`** to give a message: `Quiz champion!` for 3, `Well done!` for 1 or 2, and `Better luck next time!` for 0.
☐ **Super challenge:** time the whole quiz with a stopwatch, and print how many seconds it took.

MISSION 19

# Reaction timer

**You will learn:** the timer, `time.sleep`, `random` and `if/else`

How fast are you? The computer waits a surprise amount of time, then shouts GO! Press Enter as fast as you can, and it measures how long you took.

## Step 1: Build it in Scratch

Make a variable called **taken**, then build:



## Step 2: Type it in Colab

**Type this into a code cell**

```
import time
import random
input("Press Enter to start ")
print("Get ready...")
time.sleep(random.randint(2, 5))
start = time.time()
input("GO! Press Enter! ")
taken = time.time() - start
taken = round(taken, 2)
print("Your time: " + str(taken) + " seconds")
if taken < 0.5:
    print("Lightning fast!")
else:
    print("Good! Try again to beat it.")
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–2 | (Extensions) | Bring in the clock tools and the random tools. |
| 3 | **ask and wait** | We don't keep the answer, so there is no `answer =`. **`input`**`()` is used just for its waiting. |
| 4 | **say Get ready...** | |
| 5 | **wait pick random seconds** | `time.sleep( )` is **wait**. A random wait means you can't guess when GO is coming. |
| 6 | **reset timer** | Start the stopwatch. |
| 7 | **ask GO! and wait** | Waits until you press Enter. |
| 8 | **set taken to timer** | Now minus start: the seconds you took. |
| 9 | | **`round`**`(taken, 2)` keeps two numbers after the dot, so `0.4371825` becomes `0.44`. |
| 10 | **say join Your time: taken** | Turn the number into writing and join it. |
| 11–14 | **if then else** | Under half a second? Lightning! Otherwise, encourage another go. |

### Step 4: Run it

Press , then press Enter in the box. Get ready, and when you see GO!, press Enter as fast as you can.

**You might see**

```
Press Enter to start
Get ready...
GO! Press Enter!
Your time: 0.62 seconds
Good! Try again to beat it.
```

**Why is Colab a bit slow?**

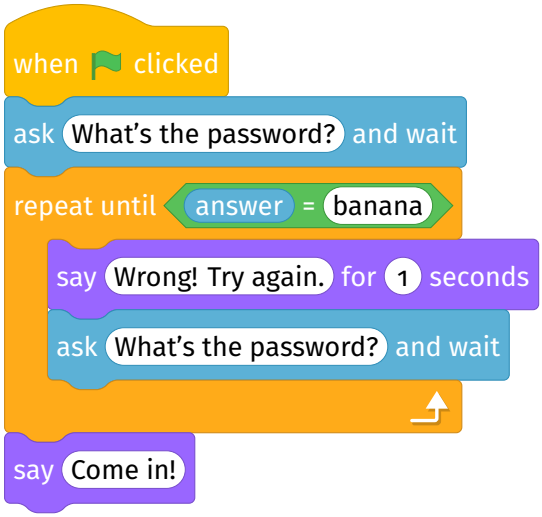
Colab runs on a computer far away. Your Enter press has to travel there over the internet, so your time includes the journey. Compare your times with each other, not with a friend's on a different computer. If `0.5` is too hard, change line 11 to `0.8`.

**Make it yours**

☐ Ask for the player's name first, and print it with their time.
☐ Add an `elif` so there are three messages: under 0.5, under 1, and the rest.
☐ **Best of three:** put lines 3 to 8 inside `for i in range(3):` (with four more spaces). Before the loop, set `best = 100`. Inside, after line 8, add `if taken < best:` and `best = taken`. Print the best time at the end.
☐ Make it sneakier: change the wait to `random.randint(1, 8)`.

## Translate it yourself

Here is a Scratch script you haven't seen before. Write the Python on the lines, then type it into Colab and run it. Does it do the same thing?




**My Python**

### Two clues

**repeat until** flips into **`while`** … `!=`. And for **say Wrong! Try again.** a `print` is enough.

### Two-line puzzle

Write **two lines** of Python that ask "How old are you?" and then print how old you will be next year, like `Next year you will be 10`.

## Part 5 checklist

☐ I can ask a question with **`input`**`()` and keep the answer in a variable.

☐ I can turn writing into a number with **`int`**`( )`, and I know a `ValueError` means I typed a word.

☐ I can make choices with **`if`**, **`else`** and **`elif`**, with a colon and four spaces.

☐ I can flip "repeat until" into a **`while`** loop with `!=` or **`not`**.

☐ I can make a stopwatch with `time.time()` and read the year with `datetime`.

☐ I can check "touching" with `distance()` and make a turtle bounce off the walls.

PART 6

# My Blocks

Invent your own Python blocks, get answers back from them, and fill the stage with sprites.

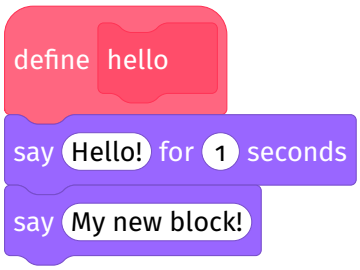
In Scratch, the pink **My Blocks** button lets you invent a brand-new block, like **square** or **jump**. You build it once, and then you can use it again and again. In Python, a home-made block is called a **function**. Think of it as a **recipe card**: you write the steps once, give the card a name, and then you just say the name whenever you want those steps.

In this part you will meet:

- **define square (size)** becomes `def square(size):`
- **square 100** becomes `square(100)`
- **create clone of myself** becomes `s = Turtle()` inside a loop

And you will learn a Python superpower that Scratch's pink blocks don't have: a block that **gives an answer back**.

## Make a block: `def`



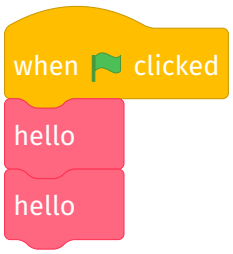

```
def hello():
    print("Hello!")
    print("My new block!")
```

**How it works.** `def` is short for **define**, so `def hello():` is the pink **define hello** hat. The name comes next, then a pair of round brackets (empty for now, because this block has no inputs). The **colon** opens the block's mouth, just like the colon after `for` or `if`. Every line with four spaces in front is **inside** the block. The first line back at the left edge is no longer part of it.

**Nothing happened?**

If you run just these three lines, **nothing appears**. That's correct! A **define** hat on its own doesn't run in Scratch either. You have written the recipe card, but nobody has cooked it yet. Read on to find out how.

## Use your block: brackets make it run




```
def hello():
    print("Hello!")
    print("My new block!")

hello()
hello()
```

**How it works.** Writing the name with round brackets, `hello()`, **calls** the block. That is the same as snapping the pink **hello** block into a script. Python jumps up to the **def**, does every line inside, and then comes back to carry on. Here it runs twice, so you see the two lines twice.

**Watch out: no brackets, no action**

If you type `hello` without the brackets, Python does **nothing**, and it doesn't show an error either! The brackets are the "go" button. Without them, you have only said the block's name, as if you pointed at the recipe card without cooking. When a block seems to be ignored, check for the ( ).

## A number input



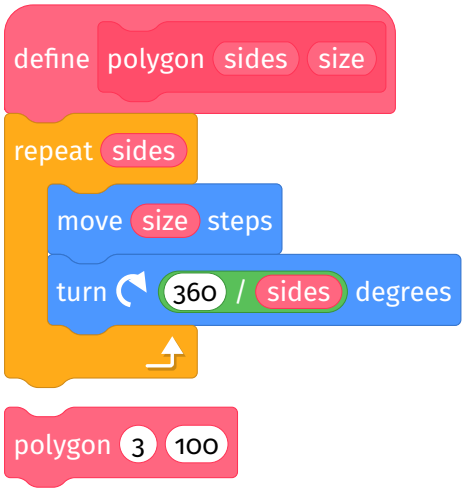

```
def square(size):
    for i in range(4):
        forward(size)
        right(90)

square(50)
square(100)
```

**How it works.** The word inside the brackets of the **def** line is an **input**. Python calls it a **parameter**. It is like the pink **size** oval in Scratch: a labelled box that gets filled each time the block is used. `square(50)` puts 50 into `size`, and `square(100)` puts 100 into

it. Inside the block, you use `size` wherever you would drag the pink oval. (Put the three starting lines above this code when you run it.)

## Two or more inputs




```
def polygon(sides, size):
    for i in range(sides):
        forward(size)
        right(360 / sides)

polygon(3, 100)
polygon(6, 50)
```

**How it works.** Put a comma between the inputs. When you call the block, the numbers go into the inputs **in order**: the first number goes into `sides` and the second into `size`. So `polygon(3, 100)` is a big triangle and `polygon(6, 50)` is a small hexagon.

**Watch out: order matters, and none can be missing**

`polygon(100, 3)` swaps them: 100 sides that are 3 steps long. You get a tiny circle! And if you forget an input, Scratch just uses a blank, but Python stops and tells you:

```
TypeError: polygon() missing 1 required positional argument:
'size'
```

Read it like a detective's clue: the block `polygon` is **missing** the input called `size`. Give it two numbers, like `polygon(6, 50)`.

**Inputs live inside the block**

In Scratch, if you drag the pink **size** oval out of the define hat and use it somewhere else, it just reports 0. In Python, an input only exists **inside** its block. Try `print(sides)` on a line at the left edge, after the block: Python says `NameError`, because `sides` belongs to `polygon` alone.

## Define it before you use it

In Scratch, the **define** hat can sit anywhere in the code area. Python is different: it reads from the **top down**, like you read a book. If a line uses a block that Python hasn't read yet, it stops:

```
flower()

def flower():
    print("A flower!")
```

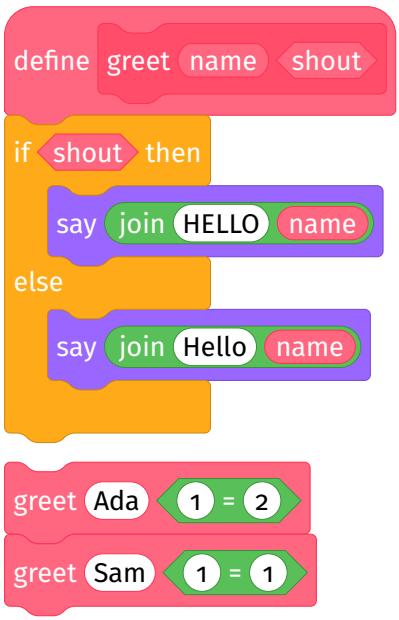
```
NameError: name 'flower' is not defined
```

The clue says Python has never heard of `flower`. The fix: move the `def` part **above** the line that uses it.

**Watch out in Colab: run the def cell first**

If your `def` is in one cell and you use the block in another, run the `def` cell **first**. And if you change a `def`, run its cell again, or Colab keeps using the old version.

## Text inputs and True/False inputs




```
def greet(name, shout):
    if shout:
        print("HELLO " + name)
    else:
        print("Hello " + name)

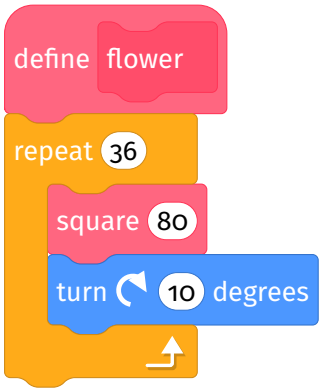
greet("Ada", False)
greet("Sam", True)
```

**You should see**

```
Hello Ada
HELLO Sam
```

**How it works.** An input can hold words too. Words need quote marks when you call the block: `greet("Ada", False)`. The pointy hexagon input in Scratch is a **True/False** input. Scratch has no True block, so Scratchers drop in a test like 1 = 1. Python just lets you type `True` or `False` (with a capital letter and no quote marks). Then `if shout:` asks "is `shout` True?"

## Run without screen refresh




☒ Run without screen refresh

```
def flower():
    animationOff()
    for i in range(36):
        square(80)
        right(10)
    animationOn()

flower()
```

**How it works.** When you make a block in Scratch, you can tick **Run without screen refresh** so it draws in a flash. Python has two ways to hurry the turtle up:

- `speed(13)` is the fastest speed where you still watch the lines being drawn.
- `animationOff()` makes the turtle **jump** instead of walk, so a big drawing appears very quickly. `animationOn()` turns walking back on.

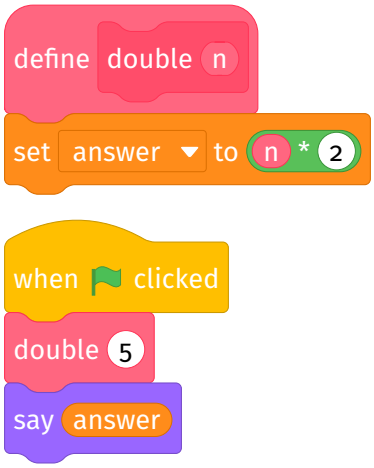
(This program also needs the three starting lines and the `def square` from the number-input page above it.)

**Watch out: don't use speed(0)**

In Colab, `speed(0)` shows **nothing at all** until the very end, and only if you add `done()`. It looks as if your program is broken. Stick to `speed(13)` or `animationOff()`.

## Python's superpower: `return`

Scratch's pink blocks can **do** things, but they can't **give an answer back**. There is no pink round reporter block. So Scratchers store the answer in a variable and read it afterwards. In Python, a block can hand the answer straight back with **`return`**.




```
def double(n):
    return n * 2

answer = double(5)
print(answer)
print(double(double(3)))
```

**You should see**

```
10
12
```

**How it works.** `return n * 2` means "work out `n * 2` and hand it back to whoever asked". So `double(5)` turns into `10`, and you can put it in a variable, print it, or even feed it into another `double`. `double(double(3))` works from the inside out: `double(3)` is 6, and `double(6)` is 12. Your block now works like a round green reporter, such as **pick random 1 to 10**!

**Watch out: print is not return**

`print` only **shows** a number on the screen. It doesn't hand it back. Look:

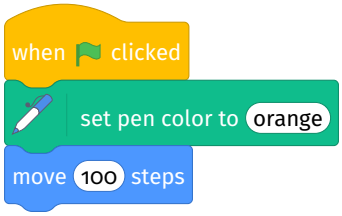
```
def show_double(n):
    print(n * 2)

x = show_double(5)
print(x)
```

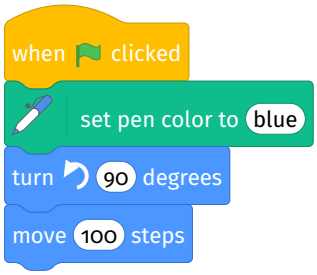
This shows `10` and then `None`. `None` is Python's word for "nothing". A block without **`return`** gives back nothing, so `x` is empty. If you want to **use** the answer, use **`return`**.

## More sprites: `Turtle()`

**Cat sprite**




**Dog sprite**

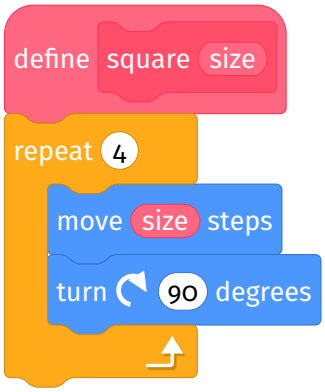



```
cat = Turtle()
dog = Turtle()
cat.pencolor("orange")
dog.pencolor("blue")
cat.forward(100)
dog.left(90)
dog.forward(100)
```

**How it works.** So far, your plain commands have moved one turtle. `cat = Turtle()` makes a **new sprite** and gives it the name `cat`. Every new sprite starts in the middle, facing right, with its pen down. To tell a sprite what to do, write its name, a dot, then the command: `cat.forward(100)`. The name before the dot is like clicking that sprite in Scratch's sprite list.

## Give a sprite to your block

In Scratch, a pink block belongs to **one** sprite. If the dog wants the cat's **square** block, you have to drag a copy over. In Python, one block can work for every sprite: you hand it the sprite as an input!




(one copy in the Cat, another copy in the Dog)

```
def square(t, size):
    for i in range(4):
        t.forward(size)
        t.right(90)

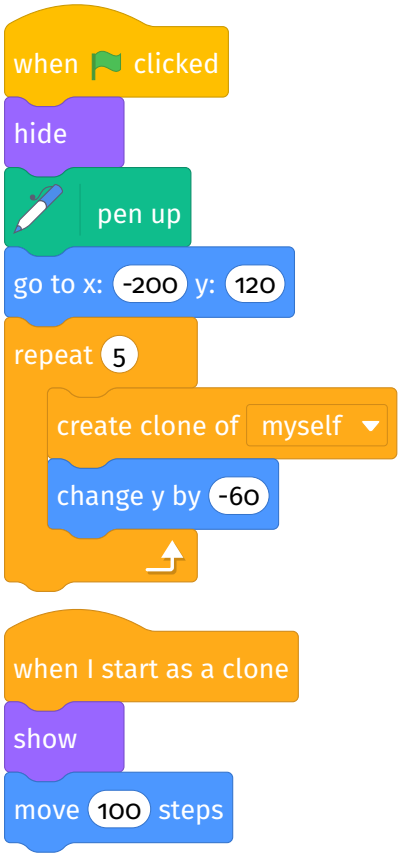
square(cat, 60)
square(dog, 100)
```

**How it works.** The input `t` holds a sprite. When you call `square(cat, 60)`, the cat goes into `t`, so `t.forward(size)` moves the cat. Call `square(dog, 100)` and the same lines move the dog. (Make `cat` and `dog` first, as above.)

**Speed is for one sprite**

`speed(13)` only speeds up the plain turtle. Each new sprite has its own speed, so write `cat.speed(13)` for the cat.

## Clones: a list of sprites




```
army = []
for i in range(5):
    s = Turtle()
    s.shape("turtle")
    s.penup()
    s.goto(-200, 120 - i * 60)
    army.append(s)

for s in army:
    s.forward(100)
```

**How it works.** Python has no "create clone" block. Instead, you make new sprites in a loop and keep them in a **list**, like a backpack full of sprites. Each time round, `Turtle()` makes a new sprite called `s`, we move it into place, and `army.append(s)` puts it in the backpack. `i` is 0, 1, 2, 3, 4, so `120 - i * 60` gives the heights 120, 60, 0, $-60$ and $-120$. Then `for s in army:` takes the sprites out one at a time and moves each one. That second loop is your **when I start as a clone** script.

**Taking turns**

Scratch clones seem to move all at the same time. Python moves them **one after another**, very quickly. Keep your army small (up to about 20 sprites), or Colab gets slow.

## Bonus: the cookie cutter

*This page is extra. Skip it if you like, and come back when you are ready.*

A **class** is a cookie cutter. You design the cutter once, then stamp out as many cookies as you like. Each cookie is its own sprite with its own position.

```
class Bug:
    def __init__(self, y):
        self.t = Turtle()
        self.t.shape("turtle")
        self.t.penup()
        self.t.goto(-200, y)

    def move(self):
        self.t.forward(30)

bugs = []
for i in range(3):
    bugs.append(Bug(i * 60))

for step in range(5):
    for b in bugs:
        b.move()
```

**How it works.** `class Bug:` starts the cookie cutter. `__init__` (two underscores on each side) runs once, the moment a new bug is made, just like **when I start as a clone**. `self` means "this bug, the one being made right now", so `self.t` is this bug's own turtle. `move` is a block that every bug knows. `Bug(60)` stamps out a new bug at height 60, and `b.move()` tells one bug to move.

## Bonus challenge: a block that uses itself

*Extra, for brave coders.* A block can call **itself**! This is called **recursion**. Each branch of this tree grows two smaller trees.

```
speed(13)

def tree(size):
    if size < 10:
        return
    forward(size)
    left(30)
    tree(size * 0.7)
    right(60)
    tree(size * 0.7)
    left(30)
    back(size)

penup()
goto(0, -170)
setheading(90)
pendown()
tree(80)
```

**How it works.** Lines 4 and 5 are the **stop rule**: when the branch is tiny, `return` ends the block straight away (like **stop this script**). Without it, the tree would grow for ever, and Python would stop with a `RecursionError`. Each new tree is 0.7 times the size of the one before. The last line, `back(size)`, walks back down the branch, so the turtle always ends where it started. (Add the three starting lines at the top.)

MISSION 20

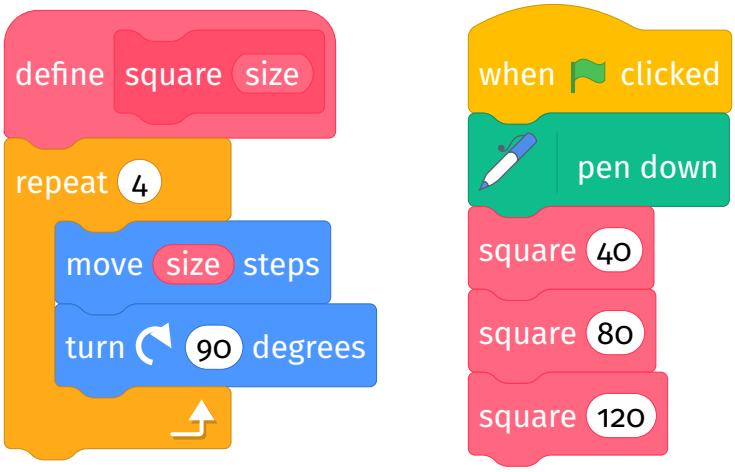
# Make your own block

**You will learn:** `def`, a number input, and using one block many times

In this mission you make a **square** block with a `size` input, then use it three times.

## Step 1: Build it in Scratch

Click **My Blocks**, then **Make a Block**. Call it `square`, and add a number input called `size`. Add the Pen extension too.




## Step 2: Type it in Colab

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))

def square(size):
    for i in range(4):
        forward(size)
        right(90)

square(40)
square(80)
square(120)
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | when green flag clicked | The three starting lines. The turtle's pen starts down, so you don't need pen down. |
| 5 | define square (size) | `def square(size):` makes a new block called `square` with an input called `size`. The colon opens it up. |
| 6 | repeat 4 | This line has four spaces, so it is inside the block. |
| 7–8 | move size steps<br>turn right 90 degrees | **Eight** spaces, because they are inside the loop, which is inside the block! `size` is used where the pink oval goes. |
| 10 | square 40 | This **uses** your block. The 40 goes into `size`. |
| 11–12 | square 80<br>square 120 | Use it again with bigger sizes. You wrote the square code once, but used it three times! |

## Step 4: Run it

Press ▶. You should see three squares of different sizes, all starting from the same corner in the middle of the stage, and growing down and to the right.

**Why is line 9 empty?**

Python doesn't need it. It helps **you** see where the block ends.

**Make it yours**

☐ Make a `triangle(size)` block. It works like `square`, but repeats 3 times and turns 120.

☐ **Draw a flower.** Add this at the bottom and run it. (Put `speed(13)` after line 3, so it doesn't take ages.)

```
for i in range(12):
    square(80)
    right(30)
```

☐ Give your block a second input for the colour: `def square(size, colour):`, then add `pencolor(colour)` as the first line inside. Use it like this: `square(80, "red")`.

☐ Type `square` on its own line, without brackets, and run it. What happens? Why?

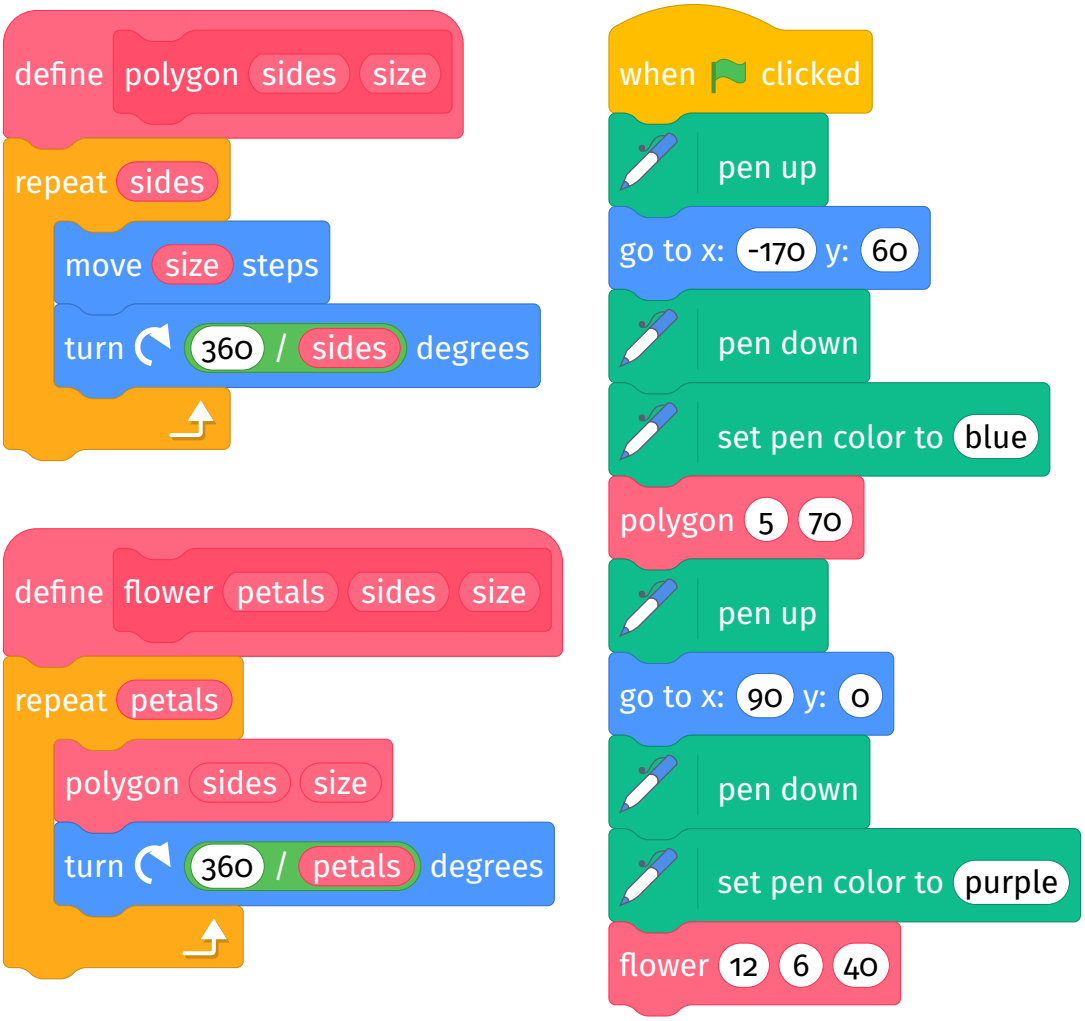
MISSION 21

# Shape machine

**You will learn:** blocks with two inputs, and a block that uses another block

One block can draw **any** shape. Then a second block uses the first one to draw a flower.

## Step 1: Build it in Scratch



## Step 2: Type it in Colab

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
speed(13)

def polygon(sides, size):
    for i in range(sides):
        forward(size)
        right(360 / sides)

def flower(petals, sides, size):
    for i in range(petals):
        polygon(sides, size)
        right(360 / petals)

penup()
goto(-170, 60)
pendown()
pencolor("blue")
polygon(5, 70)

penup()
goto(90, 0)
pendown()
pencolor("purple")
flower(12, 6, 40)
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | **when green flag clicked** | The three starting lines. |
| 4 | ☒ Run without screen refresh | `speed(13)` draws as fast as possible while you still watch. |
| 6 | **define polygon (sides) (size)** | A block with **two** inputs, with a comma between them. |
| 7 | **repeat sides** | **`range`**`(sides)` repeats as many times as the number in `sides`. |

| | | |
|---|---|---|
| 8–9 | **move size steps**<br>**turn right 360 / sides** | The turns of any shape add up to 360, so each turn is `360 / sides`. |
| 11 | **define flower (petals) (sides) (size)** | A second block with three inputs. |
| 12–14 | **polygon sides size** | The `flower` block **uses** the `polygon` block, then turns a little. Blocks can use other blocks! |
| 16–18 | **pen up**<br>**go to x: -170 y: 60**<br>**pen down** | Move to the left side without drawing a line. |
| 19–20 | **polygon 5 70** | A blue pentagon: 5 sides, each 70 steps long. |
| 22–24 | **go to x: 90 y: 0** | Move to the right side without drawing. |
| 25–26 | **flower 12 6 40** | A purple flower: 12 petals, each petal a hexagon with sides of 40. |

## Step 4: Run it

You should see a blue pentagon on the left and a purple flower made of 12 overlapping hexagons on the right.

**Order inside the brackets**

`polygon(5, 70)` and `polygon(70, 5)` are very different! The first number always goes into the first input. Try `polygon(70, 5)` and see what you get.

**Make it yours**

☐ Change line 26 to `flower(8, 4, 60)`. How many petals can you count? What shape is each petal?
☐ Add a fourth input to `flower` called `colour`, and use `pencolor(colour)` inside it.
☐ Draw a triangle, a square and a hexagon in a row, using only `polygon`, `penup`, `goto` and `pendown`.
☐ Swap line 4 for `animationOff()`. Is it faster?

MISSION 22

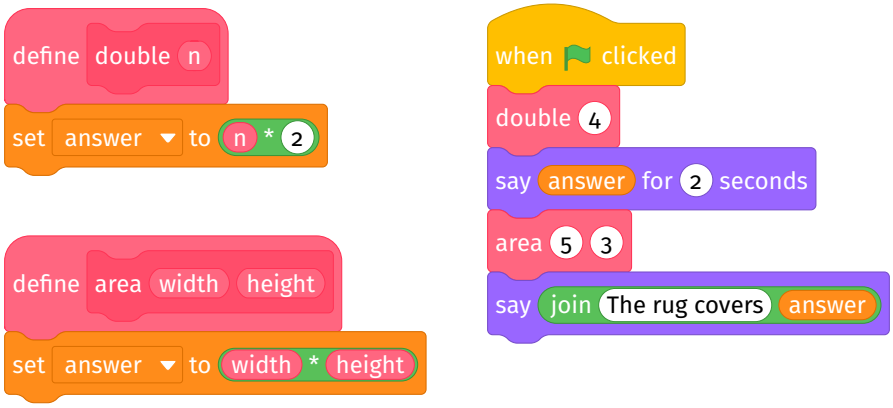
# Answer machine

**You will learn:** `return`: blocks that give back an answer

Now you build blocks that work out answers and hand them back, like your own reporter blocks. This program doesn't draw, so you don't need the three starting lines.

## Step 1: Build it in Scratch

Scratch can't make a reporter block, so these pink blocks put their answer in a variable called **answer**.




## Step 2: Type it in Colab

**Type this into a code cell**

```
def double(n):
    return n * 2

def area(width, height):
    return width * height

print(double(4))
print(double(21))
rug = area(5, 3)
print("The rug covers " + str(rug) + " squares")
print(area(double(2), 10))

w = int(input("How wide? "))
h = int(input("How tall? "))
print("Area: " + str(area(w, h)))
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–2 | define double (n) | `return n * 2` hands back double the number. No variable needed! |
| 4–5 | define area (width) (height) | The area of a rectangle is width times height. `return` hands it back. |
| 7 | double 4 · say answer | `double(4)` turns into 8, and `print` shows it. One line instead of two blocks. |
| 8 | | The same block with a different number gives 42. |
| 9 | area 5 3 | The answer, 15, is stored in a variable called `rug`. |
| 10 | say join The rug... | `str(rug)` turns the number into writing, so it can be joined with `+`. |
| 11 | | Blocks inside blocks: `double(2)` is 4, so this is `area(4, 10)`, which is 40. |
| 13–14 | ask How wide? and wait | Ask for two numbers. `int( )` turns each answer into a number. |
| 15 | say | Work out the area of **your** rectangle and show it. |

## Step 4: Run it

Press ▶. When it asks, type a width and a height, pressing Enter after each one.

**You might see**

```
8
42
The rug covers 15 squares
40
How wide? 7
How tall? 4
Area: 28
```

**Watch out: return ends the block**

As soon as Python reaches `return`, it leaves the block straight away. Any lines after `return` inside the block never run.

**Make it yours**

☐ Write `def half(n):` that gives back half of a number. Try `print(half(9))`.
☐ Write `def perimeter(width, height):` that gives back the distance all the way round a rectangle (that's `2 * width + 2 * height`).
☐ Change line 2 to `print(n * 2)`. Run it. Why do you see `None`, and why does line 11 now go wrong?
☐ **Super challenge:** write `def is_even(n):` that gives back `n % 2 == 0`. Then try `if is_even(7):`. Scratch can't make a pink True/False block at all!

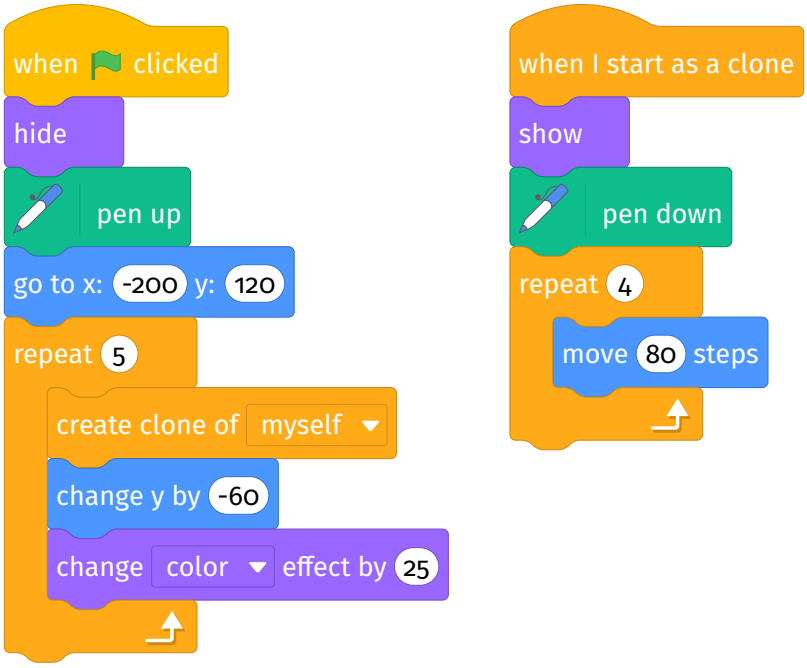
MISSION 23

# Sprite army

**You will learn:** making many sprites in a loop, keeping them in a list, and moving them together

You will make five turtle sprites, each a different colour, and march them across the stage together. These are Python's version of clones.

## Step 1: Build it in Scratch



## Step 2: Type it in Colab

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))

colours = ["red", "orange", "gold", "green", "blue"]
army = []
for i in range(5):
    s = Turtle()
    s.shape("turtle")
    s.color(colours[i])
    s.speed(13)
    s.penup()
    s.goto(-200, 120 - i * 60)
    s.pendown()
    army.append(s)

def march(steps):
    for s in army:
        s.forward(steps)

for step in range(4):
    march(80)
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | when green flag clicked | The three starting lines. |
| 5 | change color effect by 25 | A list of five colours, one for each sprite. |
| 6 | | An empty list: the backpack that will hold the army. |
| 7 | repeat 5 | Go round 5 times. `i` counts 0, 1, 2, 3, 4. |
| 8 | create clone of myself | `Turtle()` makes a brand-new sprite, called `s` for now. |
| 9–11 | show | Give it the turtle costume, its own colour (`colours[i]`), and the fastest speed. `color` sets the pen and the costume colour together. |

| | | |
|---|---|---|
| 12–14 | **pen up**<br>**go to x: -200 y: 120**<br>**pen down** | Jump to the start line without drawing. Each sprite is 60 steps lower than the one before. |
| 15 | | Put the new sprite in the backpack. |
| 17–19 | **define march (steps)** | A block that moves **every** sprite in the army forward. |
| 21–22 | **repeat 4**<br>**move 80 steps** | March 4 times. Each time, every sprite takes 80 steps. |

## Step 4: Run it

Five coloured turtles appear on the left, one above the other. Then they march to the right together, step by step, each leaving a line of its own colour. They stop at $x = 120$.

**Why not five names?**

You could write `red = Turtle()`, `orange = Turtle()` and so on. But with a list, 5 sprites take the same number of lines as 50. Just change the 5 in line 7 (and add more colours)!

**Make it yours**

☐ Give the army 8 sprites. Add three more colours to the list, and change `i * 60` to `i * 35` so they fit on the stage.
☐ Make a `turn_all(angle)` block that turns every sprite, and use it to march the army up the stage.
☐ Import `random` and make each sprite take `random.randint(10, 80)` steps instead of the same number. Now it's a race!
☐ Use `army[0]` to move only the first sprite. Which one is it?

YOUR TURN

# Translate it yourself

Here is a Scratch project with a home-made block. Write the Python on the lines (with the three starting lines at the top). Then type it into Colab and check that the turtle draws the same thing. Clue: don't forget to **call** your block, with brackets, below the `def`.




**My Python**

## Two-line puzzle

Write a block called `triple` that **gives back** three times the number you put in. It takes exactly **two lines**. When it works, this line should show 18:

```
print(triple(triple(2)))
```

## Part 6 checklist

I can...

☐ make my own block with **`def`** and a colon, and call it with brackets.

☐ give a block one or more inputs, in the right order.

☐ explain why the **`def`** must come before the call (and fix a `NameError`).

☐ use **`return`** to give an answer back, and tell `print` and **`return`** apart.

☐ make a new sprite with `Turtle()` and hand it to a block.

☐ make an army of sprites in a loop and move them all with another loop.

PART 7

# Looks, Sound and Events

Make your sprites dress up, sing a tune and pass messages, then race them all in one loop.

This is the last Part, and it is a fun one. You will dress your turtle up, make Colab play music, and learn how Python does the job of Scratch's yellow hat blocks. Here are some of the blocks you will meet:

**say Hello! for 2 seconds** **next costume** **set size to 200 %** **switch backdrop to blue sky**

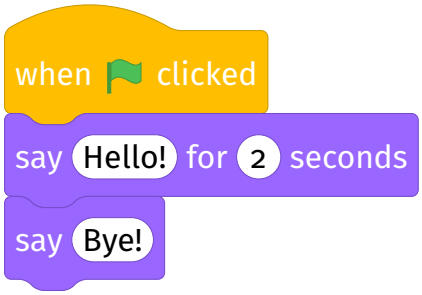
**start sound Meow** **play sound Meow until done**

**when green flag clicked** **broadcast message1** **when I receive message1**

Some of these work almost the same in Python. A few need a clever trick. And one big idea is different: Scratch looks as if it runs lots of scripts at once, but Python does **one thing at a time**. By the end of this Part you will know how to make one loop move every sprite.

## Saying things: `print`

You met `print` right at the start of the book. It is Python's **say** block. The words appear under the code cell instead of in a speech bubble.




```
import time
print("Hello!")
time.sleep(2)
print("Bye!")
```

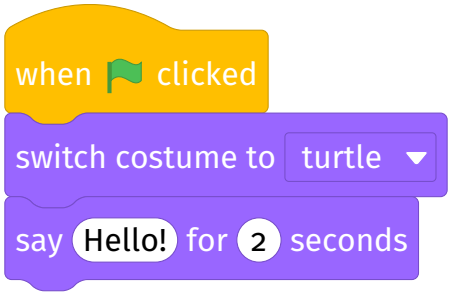
**How it works:** `print("Hello!")` is the **say** part. The "for 2 seconds" part is a wait, so it becomes `time.sleep(2)`, which you met with the **wait** block. `import time` at the top brings in the clock tools. In Scratch, the bubble disappears after 2 seconds. Printed words stay on the page, so you will see `Hello!` and then, 2 seconds later, `Bye!` underneath it.

**What about think?**

Python has no cloud bubble, so **think Hmm...** is just `print("Hmm...")` too. If you want people to know it is a thought, you could write `print("(thinks) Hmm...")`.

## Words on the stage: a speech turtle

Do you want the words to appear **on the stage**, next to your sprite? The turtle has a `write()` command that writes words where it is standing. The trick is to make a second, **invisible** turtle whose only job is to write. Let's call it `speech`.




**Type this into a code cell**

```
1  from ColabTurtlePlus.Turtle import *
2  clearscreen()
3  initializeTurtle((480, 360))
4  import time
5
6  shape("turtle")
7  speech = Turtle()
8  speech.hideturtle()
9  speech.penup()
10 speech.goto(0, 30)
11 speech.write("Hello!", align="center",
12              font=("Arial", 20, "normal"))
13 time.sleep(2)
14 speech.clear()
```

**How it works:** lines 7 to 10 make a new sprite called `speech`, hide it, lift its pen and move it just above the turtle. Lines 11 and 12 are the **say** part: `write` writes the words where `speech` is standing. `align="center"` puts the middle of the words there, and `font=` gives the letter style, the size (20) and `"normal"`. It is one command split over two lines: Python knows it isn't finished until the bracket is closed. Line 13 waits, and line 14 makes the "bubble" go away: `clear()` rubs out everything `speech` has written or drawn.

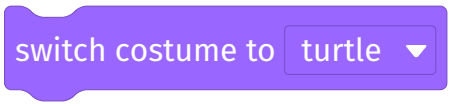
**Why a separate speech turtle?**

`clear()` wipes **all** of that turtle's drawing, not just the words. If your main turtle wrote the words and then cleared them, its lines would vanish too! A speech turtle is like a friend holding a whiteboard: to change the sign, the friend wipes the whiteboard and

writes again, and your drawing is safe. For a thought, change `"normal"` to `"italic"` and the words come out slanted, like thinking.

## Costumes: shape

In Scratch, a sprite can change its costume. The turtle can too, with `shape()`. You give it the name of a costume in quote marks.




```
shape("turtle")
```

The turtle has nine costumes. Try them all!

| | | |
|---|---|---|
| `"classic"` (the one you start with) | `"arrow"` | `"triangle"` |
| `"square"` | `"circle"` | `"turtle"` |
| `"turtle2"` (a pointy turtle) | `"ring"` | `"blank"` (invisible) |

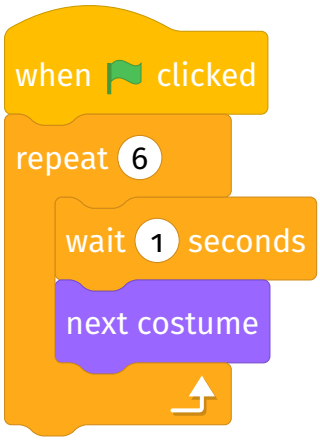
**How it works:** the costume name is writing, so it needs quote marks, and it must be spelled exactly. If you type a name that isn't on the list, Python stops with a `ValueError` and shows you the list of names it knows. That is a helpful clue note!

**No picture costumes in Colab**

In Scratch you can paint a costume or pick a cat. The Colab turtle only has the shapes in the table: it can't load pictures. Instead, make your costumes different by changing their **shape**, **colour** and **size**. That's what the rest of this page is about.

## Next costume: a list and a hanger

Scratch keeps a wardrobe of costumes for you. In Python you build the wardrobe yourself, as a **list**. A number called n is the hanger you are pointing at.



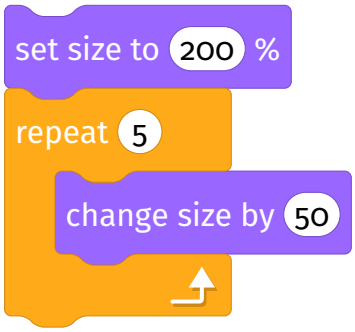

```
1 costumes = ["turtle", "circle",
2               "square"]
3 n = 0
4 shape(costumes[n])
5 for i in range(6):
6     time.sleep(1)
7     n = (n + 1) % len(costumes)
8     shape(costumes[n])
```

**How it works:** `costumes[n]` picks the costume on hanger n. Remember that lists start counting at 0, so hanger 0 is `"turtle"`. Line 7 is the clever part. `n + 1` moves to the next hanger. `len(costumes)` is 3, the number of costumes. The % sign is Scratch's **mod** block: it gives the remainder after dividing. So when `n + 1` reaches 3, `3 % 3` is 0 and you are back at the first hanger, just like Scratch going from the last costume back to the first.

| n before | `n + 1` | `(n + 1) % 3` | costume |
|---|---|---|---|
| 0 | 1 | 1 | `"circle"` |
| 1 | 2 | 2 | `"square"` |
| 2 | 3 | 0 | `"turtle"` |

Scratch's **costume number** counts from 1, so in Python it is `n + 1`.

## Size: `shapesize`




```
shapesize(2)
size = 200
for i in range(5):
    size = size + 50
    shapesize(size / 100)
```

**How it works:** Scratch measures size in percent, and 100 % is normal. Python uses "how many times as big", so `shapesize(1)` is normal (100 %), `shapesize(2)` is double (200 %) and `shapesize(0.5)` is half (50 %). To change the size bit by bit, keep your own `size` variable in percent, just like Scratch, and divide it by 100 when you give it to `shapesize`.

**Never size 0**

`shapesize(0)` makes Python stop with a `ValueError`. Scratch quietly stops your sprite getting too small, but Python doesn't. If your size goes down in a loop, make sure it never reaches 0. Use 0.1 for a tiny turtle.

## Show and hide




```
for i in range(3):
    hideturtle()
    time.sleep(0.5)
    showturtle()
    time.sleep(0.5)
```

**How it works:** `hideturtle()` makes the turtle invisible and `showturtle()` brings it back. The brackets are empty because there is nothing to tell it, but you still need them: they are what makes Python **do** the command. A hidden turtle can still move and draw. That's how the speech turtle works!

## Backdrops and colours




```
bgcolor("skyblue")
rainbow = ["red", "orange",
           "gold", "green"]
for c in rainbow:
    color(c)
    time.sleep(0.5)
```

**How it works:** `bgcolor()` ("background colour") paints the whole stage. Colab backdrops are plain colours only, not pictures, so choose a colour that feels right: `"skyblue"` for the sky, `"black"` for space, `"lightgreen"` for a field. `color(c)` colours the turtle itself. The loop goes through the `rainbow` list and gives the turtle each colour in turn, which is a simple way to do Scratch's colour effect. Python spells it `color`, the American way, so we do too in code. Remember: colour names are all small letters.

## Sound in Colab: the sound cell

Colab runs on Google's computers, far away, so Python can't reach your speakers. But it **can** send a sound to the web page, and your browser plays it. To do that, run this **sound cell** once, after the "In the beginning" cell. It makes a new block called `beep`.

**Sound cell: run it once each time you open Colab**

```
import numpy as np
from IPython.display import Audio, display

def beep(freq=440, secs=0.5):
    times = np.linspace(0, secs, int(22050 * secs))
    sound = np.sin(2 * np.pi * freq * times)
    display(Audio(sound, rate=22050, autoplay=True))
```

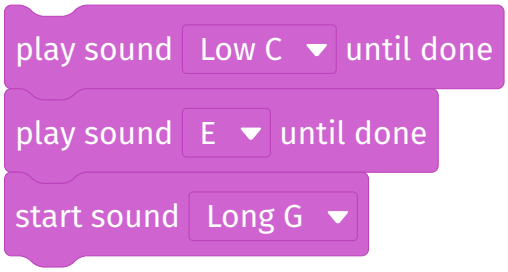
You don't need to understand every line of this. It is a recipe card that someone else wrote, and you just use it, like a block from an extension. Here is the idea. Lines 1 and 2 bring in two extensions: `numpy` (super-fast maths) and the tools that put a sound player on the page. Line 4 makes your own block, `beep`, with two inputs: how high the note is (`freq`, short for *frequency*) and how long it lasts (`secs`). `freq=440` means "if nobody says, use 440". Lines 5 and 6 work out the shape of the sound wave, and line 7 puts a little sound

player on the page and plays it straight away (`autoplay=True`).

**No sound?**

- Check that your computer's volume is up.
- Some browsers block sounds that play by themselves. If you see a small player with a ▶ button under the cell but hear nothing, press its play button.
- `NameError: name 'beep' is not defined`? Run the sound cell first.

## Start sound, and play until done




```
import time
beep(262)
time.sleep(0.5)
beep(330)
time.sleep(0.5)
beep(392, 1)
```

**How it works:** `beep(262)` is **start sound**: it starts the note and Python carries straight on to the next line. **play sound until done** waits for the sound to finish. Colab can't tell when a sound has finished, so you wait yourself: the note lasts 0.5 seconds, so `time.sleep(0.5)` waits 0.5 seconds. `beep(392, 1)` gives the second input too, so that note lasts 1 second.

**The doorbell and the radio**

**start sound** is like switching on the radio and walking off to do something else. **play sound until done** is like waiting by the radio until the song ends. Take the two `time.sleep` lines out and run it again. The notes all start at once and play on top of each other!

## Notes are numbers

The number you give beep decides how high the note is. Bigger numbers are higher. Here are the notes of a scale, starting at middle C:

| **Note** | C | D | E | F | G | A | B | high C |
|---|---|---|---|---|---|---|---|---|
| **Number** | 262 | 294 | 330 | 349 | 392 | 440 | 494 | 523 |
| **Scratch Music note** | 60 | 62 | 64 | 65 | 67 | 69 | 71 | 72 |

A tune is a list of notes, so you can play it with a **for** loop:

```
notes = [262, 294, 330, 349, 392, 440, 494, 523]
for note in notes:
    beep(note, 0.3)
    time.sleep(0.4)
```

This plays the whole scale, from C up to high C. Each note lasts 0.3 seconds, and the 0.4-second wait leaves a tiny gap before the next one.

**For grown-ups: if all the notes play at the end**

On most computers each note plays as soon as beep runs. If yours saves them all up and plays them together when the cell finishes, join the notes into one sound and play that instead. Paste this under the sound cell and use `play_tune([262, 294, 330])`:

```
def play_tune(notes, secs=0.4):
    parts = []
    for note in notes:
        times = np.linspace(0, secs, int(22050 * secs))
        parts.append(np.sin(2 * np.pi * note * times))
    song = np.concatenate(parts)
    display(Audio(song, rate=22050, autoplay=True))
```

## The green flag: running the cell

Every Scratch script starts with a hat block. The one you use most is **when green flag clicked**. In Colab the green flag is the play button : running a cell starts everything in it, from the top. That is why your drawing programs start with the three starting lines. They do the jobs a green flag script does at the start: clear the stage and get the sprite ready.

Scratch's other hat blocks need different tricks in Python. Here they are:

| Scratch | Python in Colab |
|---|---|
| **when green flag clicked** | Press  to run the cell. |
| **when I receive message1** | Make a function: `def message1():` |
| **broadcast message1** | Call the function: `message1()` |
| **when space key pressed** | Not in Colab. Ask with `input()` instead, or use Thonny (page at the end of this Part). |
| **when this sprite clicked** | Not in Colab. Ask with `input()` instead. |

## Broadcasts are function calls

In Scratch, one script can **broadcast** a message, and another script that starts with **when I receive** does its job. In Python, the receiving script is a **function**, and broadcasting is **calling** it.




```
1 def go():
2     print("Go, go, go!")
3
4 print("Ready...")
5 go()
6 print("Done!")
```

**You should see**

```
Ready...
Go, go, go!
Done!
```

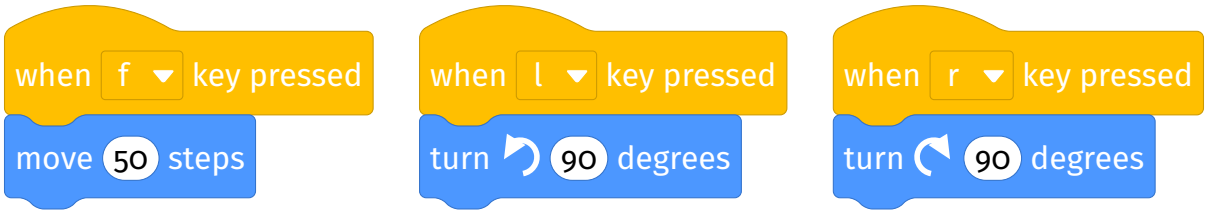
**How it works:** lines 1–2 are the **when I receive go** script. Python learns the recipe but doesn't do it yet. Line 4 runs first. Line 5 is the broadcast: Python jumps into go, does everything inside it, then comes back and carries on with line 6.

**A call always waits**

Scratch has two broadcast blocks. **broadcast** sends the message and carries on at once. **broadcast and wait** waits until the other script has finished. A Python function call **always waits**: Python finishes `go()` before it moves to the next line. So a plain call is like **broadcast and wait**. Good news: that makes it easy to know what happens in which order.

## Key presses become questions

The Colab turtle can't hear the keyboard or the mouse while a program runs. So instead of **when f key pressed**, you **ask** which key, with **`input()`**. The player types a letter and presses Enter. It's like a board game: one move per turn.




**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))

shape("turtle")
for turn in range(5):
    key = input("f, l or r? ")
    if key == "f":
        forward(50)
    elif key == "l":
        left(90)
    elif key == "r":
        right(90)
```

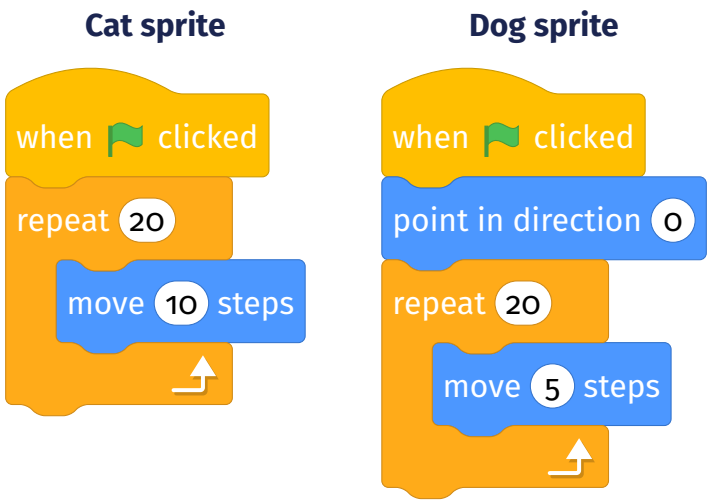
**How it works:** three Scratch scripts become one loop with five turns. Each turn, **`input`** waits for a letter. Then the **`if`** checks which letter it was and does the job of the matching hat block. **`elif`** means "else, if": it is only checked when the lines above it were not true. Try typing `f`, `l`, `f`, `r`, `f`. The turtle draws a zigzag step!

**Watch out: always give your loop a way out**

In Scratch, **forever** is safe because you can click the red stop sign. In Colab, **`while True:`** with no way out makes the cell run for ever, and nothing else can happen. Use a **`for`** loop like **`for turn in range(5):`**, or a **`while`** whose condition will become false, like the one in the Turtle race later in this Part. If a cell does get stuck, press the stop button ■.

## Many sprites, one loop

In Scratch, every sprite can have its own **when green flag clicked** script, and they all seem to run at the same time. Here a cat walks right while a dog walks up:




Python does **one thing at a time**. If you wrote the cat's loop and then the dog's loop, the cat would finish its whole walk before the dog even started. Instead, write **one loop that moves every sprite a little**:

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))

hideturtle()
cat = Turtle()
cat.shape("turtle")
dog = Turtle()
dog.shape("circle")
dog.left(90)
for step in range(20):
    cat.forward(10)
    dog.forward(5)
```

**How it works:** each time round the loop, the cat takes one small step, then the dog takes one small step. It happens so fast that they seem to move together. This is called a **game loop**, and every real computer game has one. Scratch has one too, hidden inside it, like a juggler giving each ball a little push.

MISSION 24

# Costume parade

**You will learn:** lists of costumes and colours, `shapesize`, `%` and `time.sleep`

Your turtle is going to put on a fashion show! It changes costume, colour and size every half a second, then takes a bow and disappears.

## Step 1: Build it in Scratch

Give a sprite four costumes, then build:



### Step 2: Type it in Colab

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
import time

costumes = ["turtle", "circle", "square", "triangle"]
colours = ["green", "red", "blue", "orange"]
n = 0
size = 100
bgcolor("lavender")
for i in range(8):
    shape(costumes[n])
    color(colours[n])
    shapesize(size / 100)
    print("Costume", n + 1, costumes[n])
    time.sleep(0.5)
    n = (n + 1) % len(costumes)
    size = size + 50
hideturtle()
print("Ta-da!")
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–4 | **when green flag clicked** | The three starting lines, plus the clock tools. |
| 6 | (the costumes tab) | The wardrobe: a list of four costumes. |
| 7 | **change color effect** | A list of four colours, one for each costume. |
| 8 | **switch costume to turtle** | The hanger starts at 0, the first costume. |
| 9 | **set size to 100 %** | Your own size variable, in percent like Scratch. |
| 10 | **switch backdrop to lavender** | Paint the stage a pale purple. |
| 11 | **repeat 8** | Do the inside lines 8 times. |
| 12–14 | | Put on costume number `n`, in colour number `n`, at the right size. |
| 15 | (the costume number) | Say which costume it is. `print` can show several things at once if you put commas between them, and it adds spaces for you. `n + 1` because Scratch counts costumes from 1. |
| 16 | **wait 0.5 seconds** | Pause so you can see the costume. |
| 17 | **next costume** | Move to the next hanger, and back to 0 after the last one. |
| 18 | **change size by 50** | Grow by 50 % for next time. |
| 19 | **hide** | Disappear… |
| 20 | **say Ta-da!** | …and take a bow. |

## Step 4: Run it

The stage turns pale purple. The turtle changes costume and colour every half a second, and gets bigger each time. Then it vanishes. Under the cell you will see:

**You should see**

```
Costume 1 turtle
Costume 2 circle
Costume 3 square
Costume 4 triangle
Costume 1 turtle
Costume 2 circle
Costume 3 square
Costume 4 triangle
Ta-da!
```

**Did you notice?**

The costume and the colour always go together: costume 2 is always red. That is because both lists use the same hanger number n. The two lists are like two clothes rails standing side by side.

**Make it yours**

☐ Add a fifth costume, `"arrow"`, to the end of the costume list. What else do you need to add, so that line 13 doesn't crash? (Hint: count the colours.)

☐ Make the turtle **shrink** instead: start with `size = 400` and take 40 away each time. Check that it never reaches 0!

☐ Add a list of backdrop colours, like `["lavender", "pink", "skyblue", "lightyellow"]`, and change the backdrop every time too.

☐ Use a speech turtle to write the costume name on the stage above the turtle, instead of printing it. Remember to `clear()` the old name first.

MISSION 25

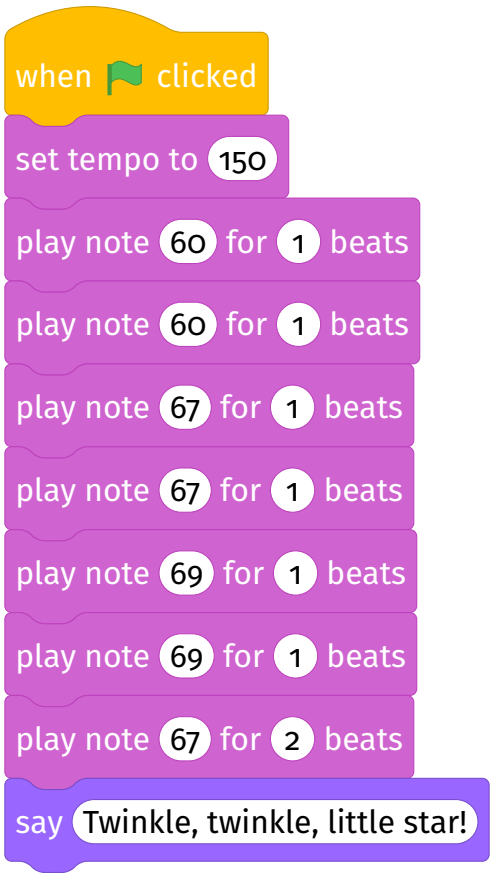
# Make music

**You will learn:** playing a tune from a list of notes, and `range(len(...))`

Now let's make Colab sing. You will play the first line of "Twinkle, Twinkle, Little Star". Before you start, run the "In the beginning" cell and the **sound cell**.

## Step 1: Build it in Scratch

Click **Extensions** and add **Music**. Its blocks are green in Scratch. This book draws them in pink, with the sound blocks.



### Step 2: Type it in Colab

**Type this into a code cell**

```
import time
C = 262
D = 294
E = 330
F = 349
G = 392
A = 440

tune = [C, C, G, G, A, A, G]
beats = [1, 1, 1, 1, 1, 1, 2]
for i in range(len(tune)):
    secs = beats[i] * 0.4
    beep(tune[i], secs)
    time.sleep(secs + 0.1)
print("Twinkle, twinkle, little star!")
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| — | when green flag clicked | No turtle needed, so no starting lines. The sound cell must have been run already. |
| 1 | | The clock tools, for waiting. |
| 2–7 | (the note numbers) | Six variables, one for each note. Now you can write G instead of remembering 392. Capital letters are fine for variable names, and they look like real music. |
| 9 | the seven play note blocks | The tune is a list of notes, in order. |
| 10 | the beats | How many beats each note lasts. The last note is twice as long. |
| 11 | repeat 7 | `len(tune)` is 7, so `range(7)` gives `i` = 0, 1, 2 … 6: one number for each note. |
| 12 | set tempo to 150 | One beat lasts 0.4 seconds. Multiply to get the length of this note. |
| 13 | play note for beats | Play note number `i` for the right time. |
| 14 | (“until done”) | Wait for the note to finish, plus a tiny gap so you can hear two Cs, not one long one. |
| 15 | say | Printed when the tune is over. |

**Why `range(len(tune))`?**

Line 11 counts through the positions, 0 to 6, instead of the notes themselves. That’s because you need the **same position** from two lists: `tune[i]` is the note and `beats[i]` is how long it lasts. It’s the same trick as the two clothes rails in the Costume parade.

## Step 4: Run it

You should hear “Twinkle, twinkle, little star” and see a small sound player appear for each note. Then the words appear under the cell:

**You should see**

```
Twinkle, twinkle, little star!
```

If you hear nothing, look back at the **No sound?** box on the sound cell page.

**Make it yours**

☐ Make it faster: change `0.4` in line 12 to `0.25`. That's like **set tempo to 240**.
☐ Add the next line of the song, "how I wonder what you are": the notes are F F E E D D C, and the last one is long. Add them to the end of both lists.
☐ **Happy Birthday!** Add `B = 494` and `C2 = 523`, then play `[G, G, A, G, C2, B]` with beats `[0.75, 0.25, 1, 1, 1, 2]`.
☐ Make a dancing turtle: add the three starting lines at the top and put `right(30)` inside the loop, so the turtle turns on every note.

MISSION 26

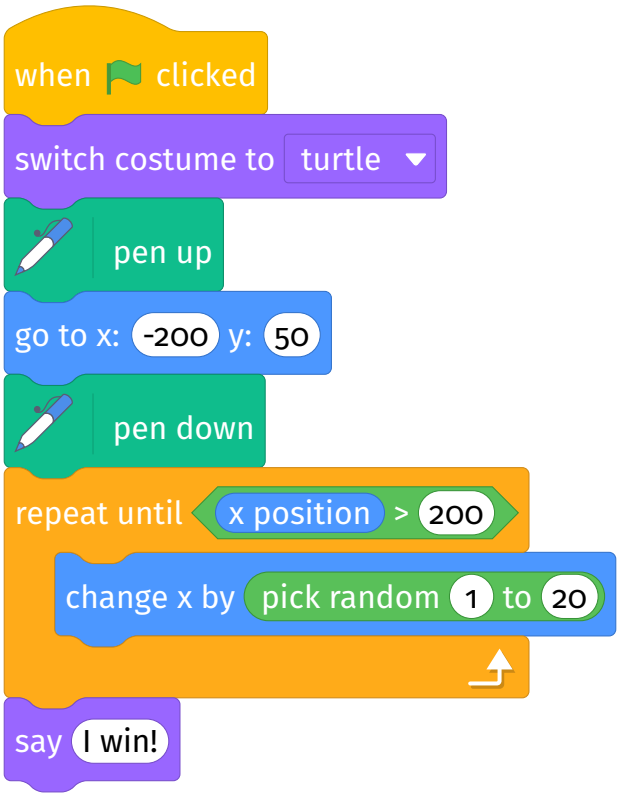
# Turtle race

**You will learn:** two sprites in one loop, a function that makes sprites, and `if/else`

Two turtles race across the stage. Each step of the loop, both move forward by a random amount. Who will win? Even the computer doesn't know!

## Step 1: Build it in Scratch

Make two sprites called **Red** and **Blue**. The red one gets this script, and the blue one gets the same script with $y = -50$:




In Scratch, each sprite runs its own script **at the same time**. In Python, **one loop moves both turtles**, one step each, again and again. And because both sprites are set up in the same way, a function can do the setting up for you.

### Step 2: Type it in Colab

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
import random

def make_racer(colour, y):
    racer = Turtle()
    racer.shape("turtle")
    racer.color(colour)
    racer.penup()
    racer.goto(-200, y)
    racer.pendown()
    return racer

hideturtle()
red = make_racer("red", 50)
blue = make_racer("blue", -50)

while red.xcor() < 200 and blue.xcor() < 200:
    red.forward(random.randint(1, 20))
    blue.forward(random.randint(1, 20))

if red.xcor() > blue.xcor():
    print("Red wins!")
    red.write("I win!", font=("Arial", 16, "bold"))
else:
    print("Blue wins!")
    blue.write("I win!", font=("Arial", 16, "bold"))
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1–4 | **when green flag clicked** | The three starting lines, plus the random number tools. |
| 6 | (a sprite factory) | A function with two inputs: the racer's colour and how high up it starts. |
| 7 | (a new sprite) | Make a brand-new sprite and call it `racer` for now. |
| 8–9 | **switch costume to turtle** | The turtle costume, in the colour you asked for. |
| 10–12 | **pen up**<br>**go to x: -200 y: 50**<br>**pen down** | Go to the start line without drawing, then put the pen down so it leaves a trail. |
| 13 | | `return` hands the finished sprite back to whoever asked for it. |
| 15 | **hide** | Hide the plain turtle in the middle: it isn't racing. |
| 16–17 | (two sprites) | Use the factory twice. Each call makes a new sprite, and you give it a name. |
| 19 | **repeat until x position > 200** | Keep going **while** red has not reached 200 **and** blue has not reached 200. This is the game loop, and it has a way out: as soon as one turtle crosses the line, it stops. |
| 20–21 | **change x by pick random** | One small step for each turtle, every time round. |
| 23 | **if then else** | Is red further right than blue? |
| 24–25 | **say I win!** | Red wins: print it, and write it on the stage next to red, in bold. |
| 26–28 | (the else part) | Otherwise blue wins. |

**Why a function?**

Without `make_racer`, you would type the six setting-up lines twice, once for red and once for blue. With it, adding a green racer is just one more line: `green = make_racer("green", 0)`. A function is a recipe card you can use again and again.

### Step 4: Run it

Watch the turtles race! The winner writes **I win!** at the finish, and under the drawing you will see one of these:

**You should see**

```
Red wins!
```

Run it again. Do you get the same winner every time?

**Make it yours**

☐ Add a third racer: `green = make_racer("green", 0)`. Then change line 19 so the race stops when **any** of the three crosses the line, and add a `forward` line for green. (Hint: three turtles means `and` twice.)
☐ Make one turtle a cheater: let it move `random.randint(1, 25)`. Run the race 5 times. Who wins most?
☐ Give `make_racer` a third input, `costume`, so one racer can be a `"circle"` and the other a `"triangle"`.
☐ Play a sound when the race is over: run the sound cell first, then add `beep(523, 1)` at the very end.

MISSION 27

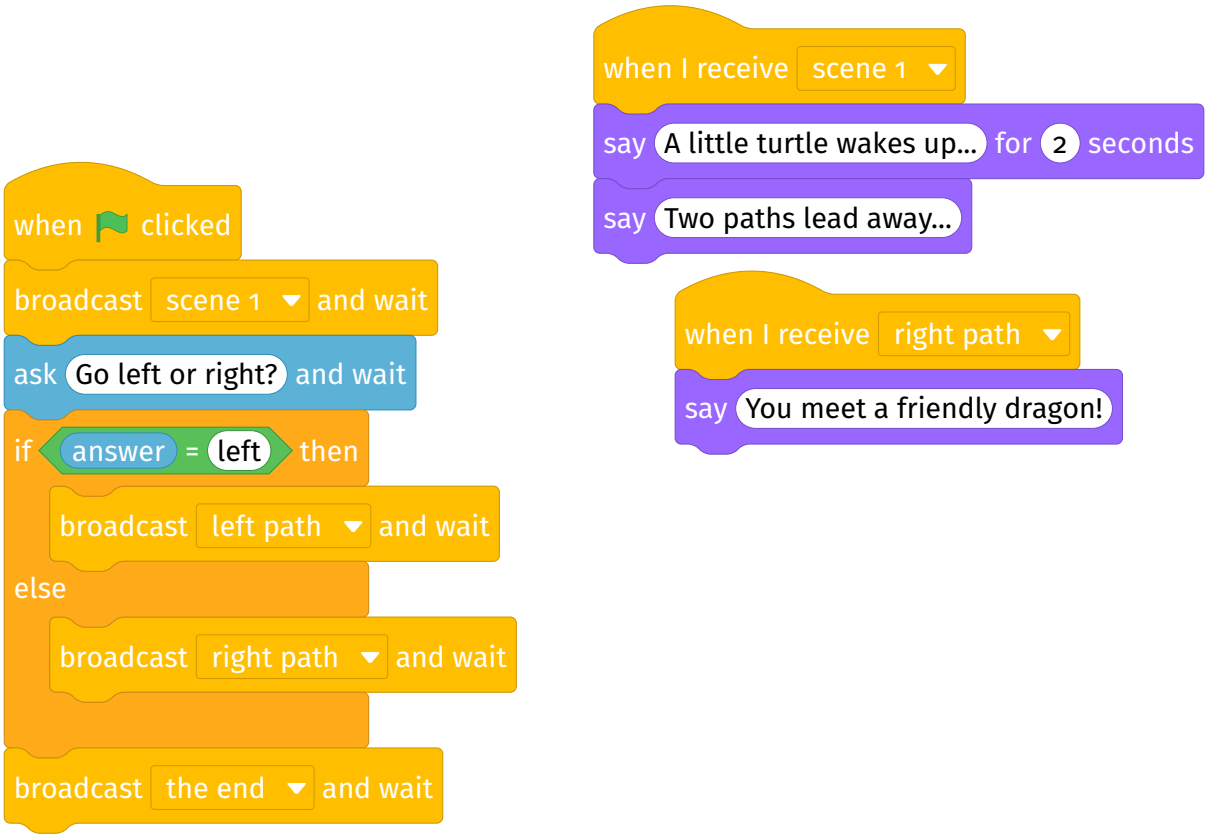
# Story with messages

**You will learn:** functions as broadcasts, and choices with `input()`

In Scratch, big stories are often built with messages. The stage broadcasts `scene 1`, then asks a question, then broadcasts a different message depending on the answer. In Python, each message is a function, and each broadcast is a function call.

## Step 1: Build it in Scratch

These scripts can go on the stage or on a sprite. The left one runs the story. The others wait for their messages. (The `left path` and `the end` scripts look like the `right path` one.)



## Step 2: Type it in Colab

**Type this into a code cell**

```
import time

def scene1():
    print("A little turtle wakes up in a forest.")
    time.sleep(2)
    print("Two paths lead away from here.")

def left_path():
    print("You find a river full of fish. Yum!")

def right_path():
    print("You meet a friendly dragon!")

def the_end():
    time.sleep(1)
    print("THE END")

scene1()
choice = input("Go left or right? ")
if choice == "left":
    left_path()
else:
    right_path()
the_end()
```

## Step 3: Read it line by line

| Line | Scratch block | What it means |
|---|---|---|
| 1 | | The clock tools. No turtle needed: this story is all words. |
| 3–6 | when I receive scene 1 | The `scene 1` script. Python learns it now, and does it later when it is called. |
| 8–9 | when I receive left path | The `left path` script. Function names can't have spaces, so we use an underscore: `left_path`. |
| 11–12 | when I receive right path | The `right path` script. |
| 14–16 | when I receive the end | The `the end` script. |

| | | |
|---|---|---|
| 18 | **broadcast scene 1 and wait** | The story starts here! Call `scene1`, and wait until it has finished. |
| 19 | **ask and wait** | Ask the question and keep the answer in `choice`. |
| 20–23 | **if then else** with two broadcasts | Call the function that matches the answer. |
| 24 | **broadcast the end and wait** | This line is back at the edge, so it happens whichever path you chose. |

**The story starts at line 18, not line 1**

Lines 3 to 16 only **teach** Python the scenes. Nothing is printed until line 18 calls the first one. If you forget the calls at the bottom, you will run the cell and see... nothing at all! That's like building **when I receive** scripts and never broadcasting.

## Step 4: Run it

Type `left` or `right` when the question appears, then press Enter.

**You might see**

```
A little turtle wakes up in a forest.
Two paths lead away from here.
Go left or right? left
You find a river full of fish. Yum!
THE END
```

**Did you notice?**

What happens if you type `Left` with a capital L? Python thinks `"Left"` and `"left"` are different, so you go right! You can fix it by changing line 19 to `choice = input("Go left or right? ").lower()`, which turns the answer into small letters.

**Make it yours**

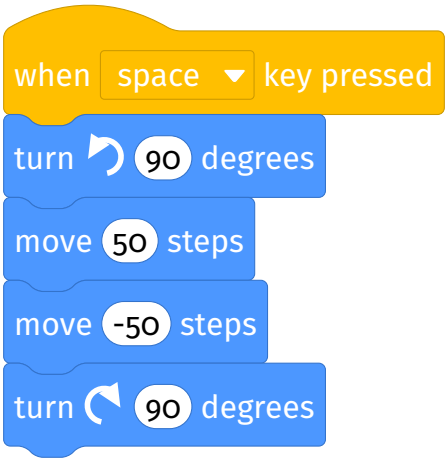
☐ Add a third choice, `straight on`, with its own function. You will need an `elif`.
☐ Inside `right_path`, ask another question ("Will you be friends? yes or no") and call a different function for each answer.
☐ Make the story musical: run the sound cell, then put a `beep` inside each scene. A low note (`beep(262)`) for scary bits, a high one (`beep(523)`) for happy bits.
☐ Add a backdrop: put the three starting lines at the top, and call `bgcolor("darkgreen")` inside `scene1` for the forest and `bgcolor("skyblue")` inside `left_path` for the river.

BONUS

# Key presses on your computer: Thonny

The Colab turtle can't hear key presses, but Python on your own computer can. **Thonny** (`thonny.org`) is a free Python editor made for beginners. It works offline and needs no account. Ask a grown-up to install it. Thonny comes with Python's own `turtle`, which opens its own window and **can** listen to the keyboard.

This is **when space key pressed** in Thonny:




**Type this into Thonny, then press the green Run button**

```
import turtle

t = turtle.Turtle()
t.shape("turtle")
screen = turtle.Screen()

def jump():
    t.left(90)
    t.forward(50)
    t.back(50)
    t.right(90)

screen.onkeypress(jump, "space")
screen.listen()
turtle.mainloop()
```

| Line | What it means |
|---|---|
| 1–5 | Thonny's starting lines. There is no `pip install`: the turtle is already there. The turtle is called `t`, so each command starts with `t.`, and `screen` is the window. |
| 7–11 | A function: the script under the hat block. |
| 13 | "When the space key is pressed, run `jump`." This is the hat block! Key names are fussy: `"space"`, `"Right"`, `"Left"`, `"Up"`, `"Down"`, or a letter like `"a"`. |
| 14 | Tell the window to start listening to the keyboard. |
| 15 | Keep the window open and keep waiting for key presses. This must be the **last** line. |

Run it, then **click on the turtle window** so it knows you are talking to it, and press space. The turtle jumps! Can you add a `go_right` function for the `"Right"` arrow key?

**Three things that catch everyone**

- **No brackets after the function name.** Write `screen.onkeypress(jump, "space")`, not `jump()`. You are telling Python *which* block to run later, not running it now.
- **Use `onkeypress`, not `onkey`.** `onkey` waits until you let go of the key, which feels slow. `onkeypress` works the moment you press it, like Scratch.
- **Click the turtle window first**, or your key presses go to the code instead.

In Thonny, `t.setheading()` uses the same compass as Colab, so the **90 minus** rule still works.

## Translate it yourself

Here is a Scratch script you haven't seen before. Write the Python on the lines. Start with the three starting lines and `import time`. Then type it into Colab and check that the turtle does the same as the Scratch sprite.

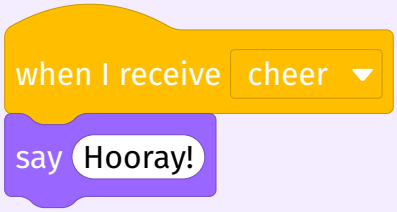



**My Python**

## Two-line puzzle

**Two-line puzzle**

Write the Python for this Scratch script. It needs only two lines!




Then add a third line that **broadcasts** `cheer`, and run it.

## Part 7 checklist

☐ I can make the turtle say things with `print`, and on the stage with a speech turtle.

☐ I can change the turtle's costume, size, colour and backdrop, and make it hide and show.

☐ I can use a list and `%` to do **next costume**.

☐ I can play notes and a tune in Colab with `beep` and `time.sleep`.

☐ I can turn **when I receive** into a function and **broadcast** into a call.

☐ I can move two sprites with one loop, and I always give my loops a way out.

BIG PROJECTS

# Big Projects

Eight real projects that use everything you know, and then one of your own.

You have finished all seven Parts. You can draw, loop, keep things in variables and lists, do maths, pick random numbers, ask questions, make choices with `if`, repeat with `while`, build your own functions, and make extra sprites. That is a whole toolbox! Now it is time to build **big things** with it.

Every project here is a classic Scratch project, rebuilt in Python. They get a little harder as you go, so do them in order:

| | Project | Big idea |
|---|---|---|
| 1 | Animate your name | loops over letters, `write()` and colours |
| 2 | Pen flower | a function inside a loop, maths in turns |
| 3 | Maths quiz | a function that gives back a score, a timer |
| 4 | Guess the number deluxe | levels, tries, and a "play again" loop |
| 5 | Choose-your-own story | functions as scenes |
| 6 | Virtual pet | variables that change every turn |
| 7 | Chase game | three sprites, `towards()` and `distance()` |
| 8 | Catch game | a falling sprite and a real game loop |

## How to do a big project

Each project has four parts. Work through them like this:

1. **The Scratch version.** Look at the Scratch script. You probably know a project like it already. If you like, build it in Scratch first.
2. **The Python version. Predict** what it will do. Then type it in and **run** it. It is longer than the missions, so take your time and check every colon and bracket.
3. **How it works.** Read the table. Point at each line in your code as you read about it.
4. **Level up. Change** it! Pick at least one challenge. Then add an idea of your own. This is the most important part: the project becomes *yours*.

**Games in Colab work in turns**

Colab cannot feel your keyboard while a program runs, so there is no **when space key pressed**. Instead, the games here ask you what to do with `input()`, then move everything, then ask again. It is like a board game: you take a turn, then the computer takes a turn. The game ideas are exactly the same as in Scratch: move, check for touching, change the score.

**A long program is typed a bit at a time**

Don't type 40 lines and then press ▶ for the first time. Type the first part (up to a blank line), run it, and check it works. Then add the next part. Programmers do this all the time. If an error appears, it is in the part you just added, so it is easy to find.

## My big projects

Tick each project when it works, and again when you have levelled it up.

| Project | It works! | I levelled it up |
|---|---|---|
| 1 Animate your name | ☐ | ☐ |
| 2 Pen flower | ☐ | ☐ |
| 3 Maths quiz | ☐ | ☐ |
| 4 Guess the number deluxe | ☐ | ☐ |
| 5 Choose-your-own story | ☐ | ☐ |
| 6 Virtual pet | ☐ | ☐ |
| 7 Chase game | ☐ | ☐ |
| 8 Catch game | ☐ | ☐ |
| My own project | ☐ | ☐ |

MISSION 28

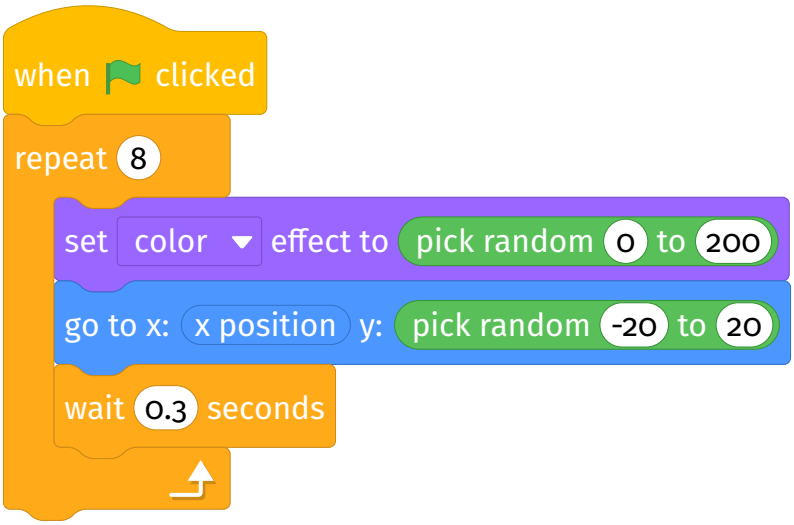
# Animate your name

**You will learn:** looping over the letters of a word, `write()`, random colours and `clear()`

Your name dances across the stage in flashing colours. Every letter jumps up and down on its own!

## The Scratch version

In Scratch you would make one sprite for each letter. Every letter sprite gets a script like this one:




In Python you don't need a sprite for each letter. A **`for`** loop visits the letters one by one, and the turtle **writes** each letter in the right place.

**Predict first**

Before you type, look at the Python on the next page. Find the line that asks for your name. Find the two **`for`** loops: one is the 8 frames and one is the letters. Which is which? What do you think happens if your name is very long?

## The Python version

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
import random
import time

bgcolor("black")
hideturtle()
penup()
speed(13)
colours = ["red", "orange", "gold", "lime", "skyblue",
           "violet", "hotpink"]

name = input("What is your name? ")
gap = 40
if len(name) > 10:
    gap = 400 / len(name)
start = 0 - (len(name) - 1) * gap / 2

for frame in range(8):
    clear()
    x = start
    for letter in name:
        color(random.choice(colours))
        goto(x, random.randint(-20, 20))
        write(letter, align="center",
              font=("Arial", 40, "bold"))
        x = x + gap
    time.sleep(0.3)

goto(0, -120)
color("white")
write("Hello, " + name + "!", align="center",
      font=("Arial", 24, "italic"))
```

## How it works

| Line | Scratch block | What it means |
|---|---|---|
| 1–5 | when green flag clicked | The three starting lines, then `random` and `time`. |
| 7–10 | hide | A black stage, and the turtle hides and lifts its pen. We only want the letters, not the turtle or lines. |
| 11–12 | (a list) | The colours to choose from. A long list can carry on to the next line, because Python knows the `[` is not closed yet. |
| 15–18 | length of | `len(name)` counts the letters. Long names get a smaller gap so they fit on the stage. `start` is where the first letter goes, so the name ends up in the middle. |
| 20–21 | repeat 8<br>erase all | Eight frames of animation. `clear()` wipes the turtle's old letters, ready to write them again. |
| 23 | | `for letter in name:` visits the letters one at a time, like reading a word with your finger. |
| 24–25 | set color effect<br>go to x: y: | A random colour, then jump to a random height. |
| 26–27 | say | `write()` prints the letter on the stage. `align="center"` puts the middle of the letter at the turtle; `font` gives the font name, size and style. |
| 28 | change x by 40 | Move along for the next letter. |
| 29 | wait 0.3 seconds | A short pause, so you can see each frame. |
| 31–34 | say Hello! | A white greeting at the bottom. |

**Code spelling, again**

The list is called `colours` (British spelling, our own name), but the command is `color()`. That is Python's own spelling, and Python only understands it that way.

**Level up**

☐ Make it 20 frames instead of 8, and try `time.sleep(0.1)` for a faster dance.
☐ Add your own favourite colours to the list (small letters only: `"deeppink"`, `"aqua"`).
☐ Make the letters grow: write each frame with size `20 + frame * 5` instead of 40.
☐ Make a rainbow: instead of a random colour, use `colours[frame % 7]` so every letter in one frame is the same colour, and the colour changes each frame.

MISSION 29

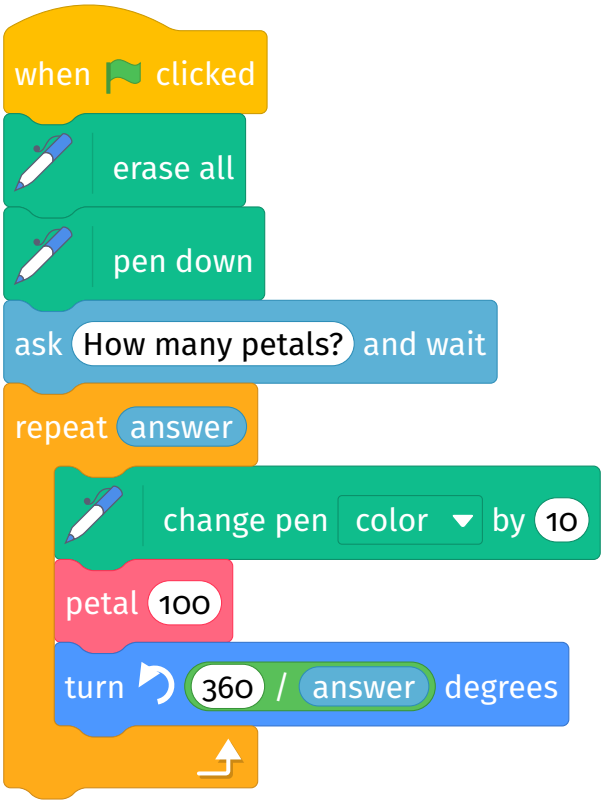
# Pen flower

**You will learn:** a function inside a loop, maths in a turn, % to repeat a list, and `circle()`

A flower with as many petals as you like, in rainbow colours, with a spinning spirograph pattern in the middle.

## The Scratch version

This uses the Pen extension and a My Block called **petal**:




**Predict first**

If you type 6 petals, how far will the turtle turn after each petal? (Work out $360 \div 6$.) And with 8 petals? Look for the line in the Python that does this sum for you.

## The Python version

**Type this into a code cell**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))

bgcolor("black")
speed(13)
pensize(2)
colours = ["red", "orange", "gold", "lime",
           "skyblue", "violet"]

def petal(size):
    circle(size, 60)
    left(120)
    circle(size, 60)
    left(120)

petals = int(input("How many petals? (3 to 20) "))
for p in range(petals):
    color(colours[p % 6])
    petal(100)
    left(360 / petals)

color("white")
for ring in range(36):
    circle(40)
    left(10)

penup()
goto(0, 0)
dot(25, "gold")
hideturtle()
```

## How it works

| Line | Scratch block | What it means |
|---|---|---|
| 1–3 | when green flag clicked | The three starting lines. |
| 5–7 | set pen size to 2 | Black stage, fastest speed, thicker pen. |
| 11–15 | define petal (size) | Our own block. `circle(size, 60)` draws one sixth of a circle (60 degrees of it). Two of those curves with turns make a pointy petal shape. |
| 17 | ask and wait | Ask for a number. `int()` turns the typed text into a number we can do maths with. |
| 18 | repeat (answer) | One trip round the loop for each petal. `p` counts 0, 1, 2, … |
| 19 | change pen color by 10 | `p % 6` is the remainder after dividing by 6. It goes 0, 1, 2, 3, 4, 5, 0, 1, …so the colours go round and round the list and never run out. |
| 20 | petal 100 | Use our block. |
| 21 | turn (360 / answer) | Share a full turn (360 degrees) equally between the petals. |
| 23–26 | repeat 36 | 36 circles, turning 10 degrees each time: $36 \times 10 = 360$, all the way round. That makes the spirograph. |
| 28–31 | pen up | Go back to the middle, put a gold dot there, and hide the turtle. |

### Level up

☐ Try 3, 7 and 20 petals. Which looks best?
☐ Change `petal(100)` to `petal(150)`. What happens? Can you make a small flower inside the big one?
☐ Give the spirograph its own colours: inside the `ring` loop, add `color(colours[ring % 6])`.
☐ Make a whole garden: put lines 18–21 inside a function `flower(x, y)` that goes to `(x, y)` first, and draw three flowers with smaller petals.

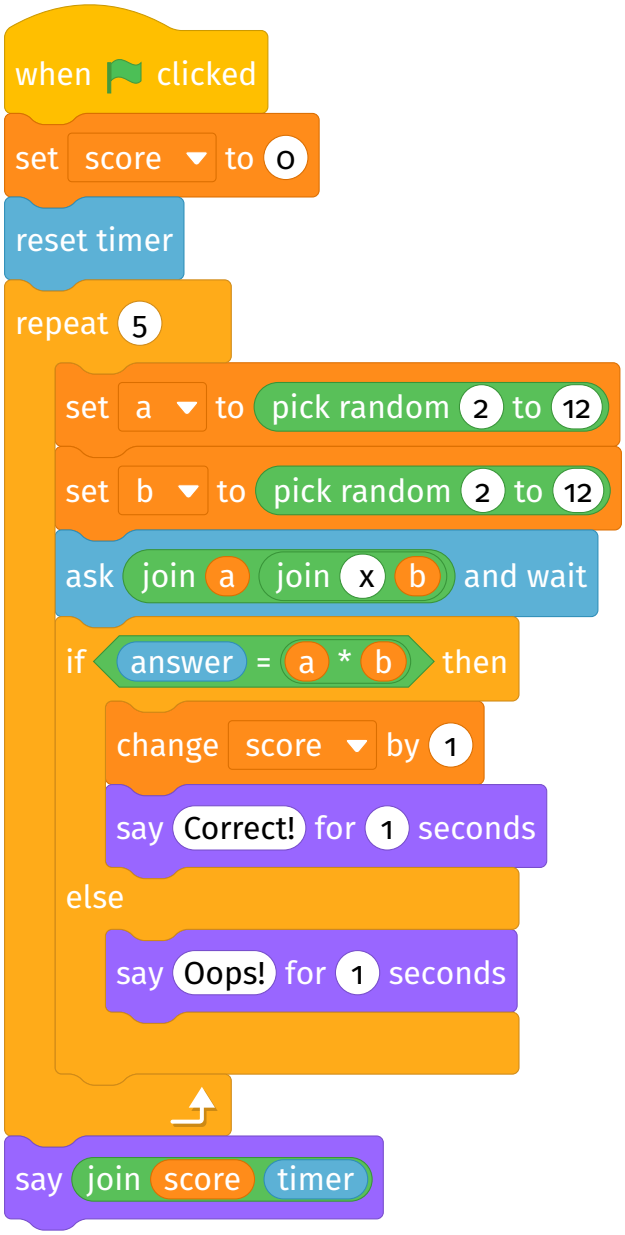
MISSION 30

# Maths quiz

**You will learn:** a function that gives back an answer with `return`, keeping score, and a stopwatch

Five times-tables questions. The computer keeps score and times you. Can you get 5 out of 5 in under 30 seconds?

## The Scratch version



**Predict first**

What do you think happens if you type `cat` instead of a number? Will the program crash? Read line 6 of the Python and the table to find out.

## The Python version

**Type this into a code cell**

```
import random
import time

def question(a, b):
    answer = input(str(a) + " x " + str(b) + " = ")
    if answer == str(a * b):
        print("Correct!")
        return 1
    else:
        print("Oops! It was", a * b)
        return 0

print("Times tables quiz: 5 questions. Go!")
score = 0
start = time.time()
for i in range(5):
    a = random.randint(2, 12)
    b = random.randint(2, 12)
    score = score + question(a, b)

seconds = round(time.time() - start)
print("You scored", score, "out of 5")
print("You took", seconds, "seconds")
if score == 5 and seconds < 30:
    print("Lightning maths champion!")
elif score >= 3:
    print("Great work!")
else:
    print("Keep practising. You'll get there!")
```

## How it works

| Line | Scratch block | What it means |
|---|---|---|
| 4 | **define question (a) (b)** | Our own block, with two inputs. |
| 5 | **ask (join a " x " b) and wait** | `str( )` turns a number into text so it can be joined with `+`. |
| 6 | **answer = a * b** | `input( )` always gives text, so we turn the right answer into text too and compare text with text. Now typing `cat` can't crash the quiz: it is just wrong! |
| 8, 11 | (no Scratch block!) | `return 1` hands the number 1 back to whoever used the block. Scratch My Blocks can't do this. It is a Python superpower. |
| 14–15 | **set score to 0**<br>**reset timer** | `time.time( )` reads the clock (in seconds). We save it in `start`, like pressing a stopwatch. |
| 19 | **change score by 1** | Ask a question and add what comes back: 1 if right, 0 if wrong. |
| 21 | **timer** | Clock now minus clock at the start = how long you took. `round( )` makes it a whole number. |
| 24–29 | **if then else** | Three different messages. `elif` means "else, if". |

**You might see (your questions will be different)**

```
Times tables quiz: 5 questions. Go!
12 x 4 = 48
Correct!
9 x 11 = 98
Oops! It was 99
…
```

**Level up**

☐ Make it 10 questions. Which two lines must change?
☐ Only practise your hardest table: make `a = 7` every time.
☐ Make the timer stricter: lightning champion only if `seconds < 20`.
☐ If you made `beep( )` in Part 7, run that cell first, then play a happy high note in the `if` part of `question` and a low note in the `else` part.

MISSION 31

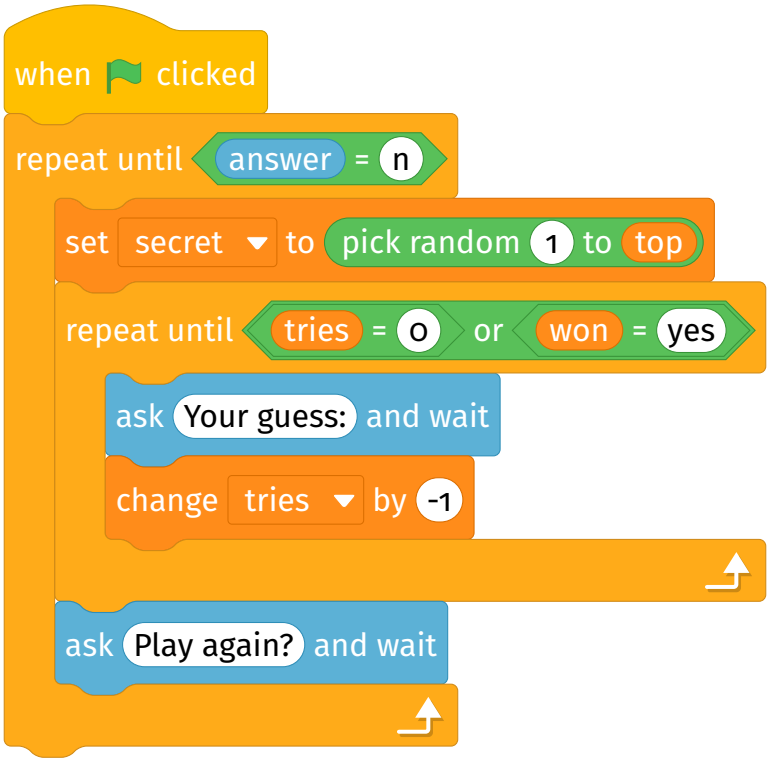
# Guess the number deluxe

**You will learn:** nested `while` loops, `and`/`not`, difficulty levels and "play again"

The guessing game, with three levels, a limited number of tries, and a "play again?" question at the end.

## The Scratch version

This is the important part: one "repeat until" inside another.




**Predict first**

The Python has two **`while`** loops, one inside the other, just like the Scratch script. Find them. Which one is one whole game, and which one is one guess? What do you think `tries = tries - 1` does?

## The Python version

**Type this into a code cell**

```
import random

again = "y"
while again == "y":
    print("Levels: 1 easy, 2 medium, 3 hard")
    level = input("Pick a level: ")
    if level == "1":
        top = 10
        tries = 4
    elif level == "2":
        top = 50
        tries = 6
    else:
        top = 100
        tries = 7
    secret = random.randint(1, top)
    print("I'm thinking of a number from 1 to", top)

    won = False
    while tries > 0 and not won:
        print("Tries left:", tries)
        guess = int(input("Your guess: "))
        tries = tries - 1
        if guess == secret:
            won = True
        elif guess < secret:
            print("Too low!")
        else:
            print("Too high!")

    if won:
        print("You got it! It was", secret)
    else:
        print("Out of tries! It was", secret)
    again = input("Play again? (y/n) ").lower()

print("Thanks for playing!")
```

## How it works

| Line | Scratch block | What it means |
|---|---|---|
| 3–4 | **repeat until answer = n** | The big loop: one trip round = one whole game. It keeps going while `again` is `"y"`. |
| 7–15 | **if then else** | The level sets two variables: the biggest number and how many tries you get. Anything other than 1 or 2 gives the hard level. |
| 19 | **set won to no** | won is a true-or-false variable. It starts `False`. |
| 20 | **repeat until tries = 0 or won** | The small loop: one trip round = one guess. Keep going while you still have tries **and** you have **not** won yet. |
| 22–23 | **ask and wait** | Ask for a guess, turn it into a number, and use up one try. |
| | **change tries by -1** | |
| 24–25 | **set won to yes** | Right! `won = True` makes the small loop stop. |
| 31–34 | **if then else** | After the small loop: did you win or run out? `if won:` is short for `if won == True:`. |
| 35 | **ask Play again?** | `.lower()` turns `Y` into `y`, so capitals work too. |
| 37 | **say** | This line is back at the margin, so it runs once, after the big loop has finished. |

### Is it fair?

On hard, you get 7 tries for 100 numbers. Is that enough? Yes, **if** you are clever! Always guess the middle of the numbers that are left (50, then 25 or 75, and so on). Each guess cuts the numbers in half, and 7 halvings take you from 100 down to 1. That is why easy has 4 tries and medium has 6. Try it!

### Level up

☐ Add a level 4, “super hard”: numbers up to 1000. How many tries is fair? (Keep halving 1000 until you get to 1.)
☐ Count how many games you won, and print it at the very end.
☐ Warn the player when they are close: if the guess is within 5 of the secret, print `Hot!`
☐ Make the game ignore guesses that are too big: if `guess > top`, print a message and don’t use up a try.

MISSION 32

# Choose-your-own story

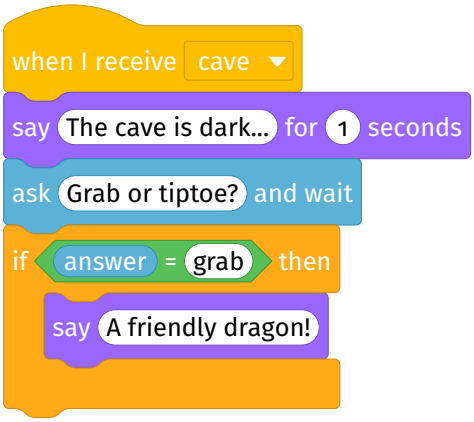
**You will learn:** functions as scenes, a function that checks the answer, and `return`

An adventure where **you** decide what happens. Every choice leads somewhere different. There are four endings. Can you find them all?

## The Scratch version

In Scratch, each scene could be a **broadcast**. The main script sends the message, and another script **receives** it:




In Python, each scene is a **function**. Calling the function, like `cave()`, is the same as **broadcast cave and wait**.

**Predict first**

Count the scenes. How many different endings can the story have? Draw a little map: a box for each scene and an arrow for each choice.

## The Python version

**Type this into a code cell**

```
import time
def choose(question, a, b):
    prompt = question + " (" + a + " or " + b + ") "
    answer = ""
    while answer != a and answer != b:
        answer = input(prompt).lower()
    return answer

def say(words):
    print(words)
    time.sleep(1)

def cave():
    say("The cave is dark. Something glitters...")
    say("...and something growls!")
    pick = choose("Grab or tiptoe?", "grab", "tiptoe")
    if pick == "grab":
        say("A friendly dragon shares its gold!")
    else:
        say("You sneak out safely.")

def river():
    say("A wide river. There is a wobbly boat.")
    pick = choose("Row or swim?", "row", "swim")
    if pick == "row":
        say("You reach an island of talking parrots!")
    else:
        say("Brrr! Too cold. Hot chocolate time!")

hero = input("What is your hero's name? ")
say(hero + " stands at the edge of a dark forest.")
way = choose("Go left or right?", "left", "right")
if way == "left":
    cave()
else:
    river()
say("THE END")
```

## How it works

| Line | Scratch block | What it means |
|---|---|---|
| 2–7 | **define choose** | A helper block. It asks the question and **keeps asking** until you type one of the two choices. Then `return` hands your choice back. |
| 5 | **repeat until** | Keep going while the answer is not a **and** not b. `!=` means "is not equal to". |
| 9–11 | **say for 1 seconds** | Our own "say for 1 second" block: print, then wait. |
| 13–20 | **when I receive cave** | The cave scene. Defining it does not run it yet. It waits until it is called, just like a "when I receive" script waits for its message. |
| 16 | **ask and wait** | `choose()` gives back `"grab"` or `"tiptoe"`, and we keep it in `pick`. |
| 22–28 | **when I receive river** | The river scene. |
| 30–37 | **when green flag clicked** | The story starts here, at the margin. |
| 34, 36 | **broadcast cave and wait** | Calling a scene. The story jumps into the function, runs it, then comes back and carries on at line 37. |

**Level up**

☐ Change the words so the story is about something you love: space, football, dinosaurs.
☐ Add a new scene `def bridge():` and give the river a third choice that leads to it. (Hint: you can call `choose()` from inside a scene, and you can call one scene from inside another.)
☐ Count the treasure: give each ending a different number of gold coins and print it.
☐ Make a story map on paper first: boxes for scenes, arrows for choices. Then write one function for each box.

MISSION 33

# Virtual pet

**You will learn:** several variables that change every turn, a forever loop with `break`, and keeping numbers in range

Look after your own pet. Feed it and play with it. If it gets too hungry or too sad, it runs away!

## The Scratch version




**Predict first**

What do you think happens if you only ever type `play` and never `feed`? How many days will your pet last? Run it and see if you were right.

## The Python version

**Type this into a code cell**

```
import random

name = input("Name your pet: ")
hunger = 3
happy = 7
day = 0

while True:
    day = day + 1
    print("Day", day, "|", name, "| hunger", hunger,
          "| happy", happy)
    choice = input("feed, play or quit? ").lower()
    if choice == "feed":
        hunger = hunger - 3
        print(name, "munches happily. Yum!")
    elif choice == "play":
        happy = happy + 2
        print(name, "chases the ball!")
    elif choice == "quit":
        print("Bye!", name, "will miss you.")
        break
    else:
        print("Try feed, play or quit.")

    hunger = hunger + 1
    happy = happy - random.randint(0, 1)
    if hunger < 0:
        hunger = 0
    if happy > 10:
        happy = 10
    if hunger >= 10 or happy <= 0:
        print("Oh no!", name, "ran away!")
        print("Look after your next pet better.")
        break
```

## How it works

| Line | Scratch block | What it means |
|---|---|---|
| 3–6 | **set hunger to 3** | The pet's "stats". Hunger goes from 0 (full) to 10 (starving); happy goes from 0 (sad) to 10 (very happy). |
| 8 | **forever** | `while True:` repeats forever. That is safe here, because `input()` waits for you every turn, and `break` gets us out. |
| 10–11 | **say** | Show the stats. The \| signs are just a fence between the numbers. |
| 13–23 | **if then else** | One answer, four possible actions. `elif` checks the next choice only if the earlier ones were wrong. |
| 21 | **stop this script** | `break` jumps out of the loop straight away. |
| 25–26 | **change hunger by 1** | Every day, the pet gets hungrier and maybe a little sadder (it loses 0 or 1 happy point at random). |
| 27–30 | **if then** | Keep the numbers in range: hunger can't go below 0, and happy can't go above 10. |
| 31–34 | **stop all** | `or` means either one is enough: too hungry **or** too sad, and the pet runs away. |

**You might see**

```
Name your pet: Biscuit
Day 1 | Biscuit | hunger 3 | happy 7
feed, play or quit? feed
Biscuit munches happily. Yum!
Day 2 | Biscuit | hunger 1 | happy 7
```

**Level up**

☐ Add a third stat, `energy`, and a new choice `sleep` that fills it up. Playing should use some energy.
☐ Give the pet a mood message: if `happy` is 8 or more, print `(purring)`.
☐ Make the pet win: if you look after it for 15 days, print a certificate and `break`.
☐ Draw your pet! After the three starting lines, draw a face whose colour depends on `happy` (for example `"gold"` when happy, `"gray"` when sad), and redraw it every turn.

MISSION 34

# Chase game

**You will learn:** three sprites, `towards()` to point at a sprite, and `distance()` for "touching"

You are a turtle. Collect the golden stars, but watch out: an orange cat is chasing you! Each turn you type `w`, `a`, `s` or `d` and press Enter, then the cat takes a step towards you.

## The Scratch version

In Scratch the player, the cat and the star each have their own scripts:

**Player**




**Cat**




**Star**




In Python there is **one** game loop. Each trip round the loop is one turn: the player moves, we check the star, the cat moves, we check the cat.

**Predict first**

Find the three sprites in the Python. Two of them have names (`cat` and `star`). Which sprite is moved by the plain commands like `forward(40)`, with no name in front?

### The Python version

The first part makes the three sprites. The plain commands are **you**; `cat` and `star` are new sprites.

**Type this into a code cell (part 1)**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
import random

speed(13)
shape("turtle")
color("green")
penup()

cat = Turtle()
cat.speed(13)
cat.shape("circle")
cat.color("orange")
cat.penup()
cat.goto(200, 140)

star = Turtle()
star.speed(13)
star.shape("triangle")
star.color("gold")
star.penup()
star.goto(120, -80)
```

Run it now: you should see a green turtle in the middle, an orange circle (the cat) at the top right and a gold triangle (the star). Then type part 2 **in the same cell**, underneath, and run the whole cell again.

**Type this into a code cell (part 2)**

```
score = 0
for turn in range(20):
    move = input("Move with w, a, s or d: ").lower()
    if move == "w":
        setheading(90)
    elif move == "s":
        setheading(270)
    elif move == "a":
        setheading(180)
    elif move == "d":
        setheading(0)
    forward(40)
    if xcor() > 220 or xcor() < -220:
        back(40)
    if ycor() > 160 or ycor() < -160:
        back(40)

    if star.distance(pos()) < 20:
        score = score + 1
        print("You got the star! Score:", score)
        x = random.randint(-5, 5) * 40
        y = random.randint(-4, 4) * 40
        star.goto(x, y)

    cat.setheading(cat.towards(pos()))
    cat.forward(25)
    if cat.distance(pos()) < 30:
        print("The cat caught you!")
        break

print("Game over! Stars:", score)
```

## How it works

| Line | Scratch block | What it means |
|---|---|---|
| 6–9 | **switch costume** | You are a green turtle with the pen up. |
| 11–16 | (a new sprite) | `cat = Turtle()` makes a new sprite called `cat`. Every command for it starts with `cat.` |
| 18–23 | (a new sprite) | The star, the same way. |
| 26 | **forever** | 20 turns. Each trip round the loop is one turn. |
| 27 | **when key pressed** | Ask for a key. Colab can't feel key presses, so you type the key and press Enter. |
| 28–35 | **point in direction** | Face the way you chose. Python directions: 90 is up, 0 is right (remember: Python = 90 − Scratch). Any other key: keep going the same way. |
| 36–40 | **move 40 steps** **if on edge, bounce** | Move. If that took you past the edge of the stage, step back again. |
| 42 | **touching Star?** | `pos()` gives your position as a pair `(x, y)`. `star.distance(pos())` is how far the star is from you. Less than 20 counts as touching. |
| 45–47 | **go to random position** | Pick a new random place. The `* 40` puts the star on spots you can land on, because you move 40 steps at a time. |
| 49 | **point towards Player** | `cat.towards(pos())` works out which direction points from the cat to you, and the cat turns that way. |
| 50–53 | **touching Player?** **stop all** | The cat steps 25 towards you. If it is closer than 30, you are caught and **break** ends the game. |

**Why is the cat slower than you?**

You move 40 steps a turn and the cat moves 25, so you can always run away, but only until you reach the edge! Corners are dangerous. Change the cat's 25 to 35 and see how much harder it gets.

**Level up**

☐ Make the game longer (more turns) or harder (a faster cat).
☐ Give the cat a speed-up: every time you get a star, the cat's step gets 5 bigger. (Hint: keep the step in a variable `cat_step`.)
☐ Add a second cat that starts in the bottom-left corner.
☐ Make your turtle leave a trail: use `pendown()` instead of `penup()` and a nice `pencolor`.

MISSION 35

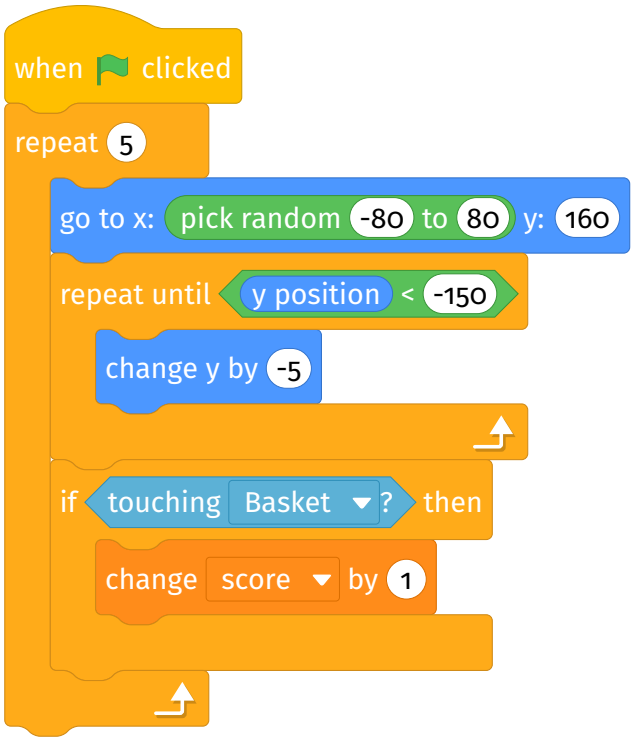
# Catch game

**You will learn:** a falling sprite, a game loop inside a game loop, `jumpto()` and `distance()`

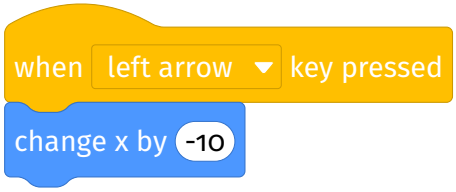
Apples fall from the sky, and they drift in the wind. Watch the first half of the fall, then work out where the apple will land and move your basket there. Can you catch all five?

## The Scratch version

### Apple




### Basket




**Predict first**

The basket waits for you to type a number. What do you think happens if you type a word, like `left`, instead? (Try it! Then read the last line of the error message like a detective.)

### The Python version

**Type this into a code cell (part 1)**

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
import random

bgcolor("skyblue")
hideturtle()

basket = Turtle()
basket.shape("square")
basket.color("brown")
basket.penup()
basket.goto(0, -150)

apple = Turtle()
apple.shape("circle")
apple.color("red")
apple.penup()
apple.speed(13)
```

Run it: a brown square (the basket) at the bottom and a red circle (the apple). Now add part 2 underneath, in the same cell.

**Type this into a code cell (part 2)**

```
score = 0
for apples in range(5):
    x = random.choice([-80, 0, 80])
    drift = random.choice([-20, 0, 20])
    apple.jumpto(x, 160)
    for fall in range(4):
        apple.goto(apple.xcor() + drift,
                   apple.ycor() - 40)

    spot = int(input("Basket x (-240 to 240): "))
    basket.goto(spot, -150)

    for fall in range(4):
        apple.goto(apple.xcor() + drift,
                   apple.ycor() - 40)
    if basket.distance(apple.pos()) < 50:
        score = score + 1
        print("Caught it!")
    else:
        print("Splat! It landed at x =",
              round(apple.xcor()))

print("You caught", score, "out of 5 apples")
```

## How it works

| Line | Scratch block | What it means |
|---|---|---|
| 6–7 | switch backdrop | A sky-blue stage. We hide the plain turtle because this game only needs the basket and the apple. |
| 9–19 | (two new sprites) | The basket and the apple, each with its own name. |
| 22 | repeat 5 | The big loop: one trip round = one apple. |
| 23–24 | pick random | `random.choice()` picks one item from a list. The apple starts at $-80$, 0 or 80, and the wind blows it $-20$, 0 or 20 steps sideways on every step down. |
| 25 | go to x: y: | `jumpto()` moves instantly, with no gliding, so the apple pops up at the top. |
| 26–28 | repeat until<br>change y by -40 | The small loop: four steps down (and sideways). This is the first half of the fall. |
| 30–31 | when key pressed<br>set x to | You type an x position and the basket glides there. |
| 33–35 | change y by -40 | Four more steps: the second half of the fall. |
| 36 | touching Basket? | `apple.pos()` is the apple's position. If the basket is closer than 50 steps to it, that counts as a catch. |

### The secret of catching

The apple falls 8 steps and drifts the same amount on every step. When it stops halfway, it has done 4 steps. So in the second half it will drift **exactly the same amount again**. If it started at 0 and is now at 80, where will it land?

### Level up

☐ Let the apple start anywhere from $-160$ to 160 (but be careful: a strong wind could blow it off the stage).
☐ Make it harder: catch only if the basket is closer than 30.
☐ Make a computer player! Delete the `input` line and let the basket work out the landing spot by itself.
☐ Leave a mark: after each apple lands, add `apple.stamp()` so you can see where all five apples fell. If you have `beep()` from Part 7, beep when you catch one!

YOUR TURN

# Your own project

You have built eight big projects. Now for the best one: **your own**. The best way to start is to **remix one of your own Scratch projects**. Open it in Scratch, look at its scripts, and plan how to turn it into Python.

## Is it a good one for Colab?

Most Scratch projects can be turned into Python. Here is how to swap the tricky blocks:

| In Scratch | In Python (Colab) |
|---|---|
| when key pressed | a turn: `move = input("w, a, s or d? ")` |
| touching Sprite2? | `cat.distance(pos()) < 20` |
| point towards Sprite2 | `cat.setheading(cat.towards(pos()))` |
| forever | `for turn in range(20):` or `while True:` with `input()` and `break` |
| broadcast and wait | calling your own function: `cave()` |
| say Hello! | `print("Hello!")` or `write("Hello!")` |
| play sound | `beep(262)` (from Part 7) |
| a second sprite | `cat = Turtle()` |
| costumes | shapes (`"turtle"`, `"circle"`, ...) and colours |

## My plan

**My project**

**Name of my project:**

_____

**It is a remix of my Scratch project:**

_____

**What happens (in three sentences):**

_____

_____

_____

**My sprites**

| Sprite in Scratch | Python name | What it does |
|---|---|---|
| | | |
| | | |
| | | |

## Build it, step by step

- ☐ Start a new Colab notebook. Put the name of your project in a text cell at the top.
- ☐ Type the three starting lines (if it draws) and make your sprites. Run it.
- ☐ Add **one** thing at a time: movement, then touching, then the score. Run after each one.
- ☐ When something goes wrong, read the error message like a detective: the last line first, then the line number.
- ☐ Show your project to someone at home. Ask them to play it, and watch what they do.
- ☐ Add one more idea that they gave you. That is a remix too!

## Blocks I will need, and their Python

| Scratch block | Python line |
| --- | --- |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

## Things to fix or add next time

**You are a real Python programmer!**

You can turn your own ideas into programs,
in blocks and in Python. Keep making things!

WHEN THINGS GO WRONG

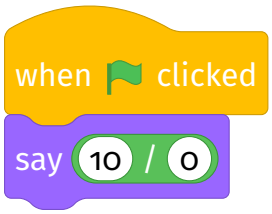
# The Bug Zoo

Meet the twelve bugs every coder catches, and learn to tame them like a detective.

A mistake in code is called a **bug**, and finding and fixing it is called **debugging**. Every programmer in the world makes bugs every single day, including the people who invented Python. In this zoo you will meet each kind of bug and learn how to catch it.

## Why does Python complain when Scratch never did?

Scratch was **designed** to have no error messages. Its blocks are like LEGO: they only snap together in ways that make sense, so you can never put a block in the wrong place. And when something odd happens, Scratch quietly does something harmless and carries on. Try this in Scratch:




The cat just says **Infinity** and keeps going.

```
print(10 / 0)
```

Python does not sweep problems under the rug. When it reaches something it cannot do, it **stops** and leaves you a note:

```
ZeroDivisionError: division by zero
```

Think of Python as a detective's helpful assistant. When something goes wrong, it freezes the scene and writes you a **clue note**. The note tells you **where** it happened (the line number, like a room in a house), **what kind** of trouble it was (the error name, like a suspect's name) and gives you a **hint**. Red writing does not mean "you failed". It means "a clue has arrived!"

**Two families of bugs**

**Grammar bugs** (`SyntaxError` and `IndentationError`): Python reads the whole cell first, finds a line it cannot understand, and runs **nothing**, not even line 1.
**Running bugs** (all the others): Python runs the lines one by one and stops at the bad one. Anything printed **before** that line still appears, which is a clue too: "it got this far".

## How to read an error in Colab

Here is a cell with a bug in it. Can you spot it before you read on?

```
score = 10
print(scroe)
```

When you run it, Colab shows something like this under the cell (the top lines may look a little different on your screen, but the last line will be the same):

**What Colab shows under the cell**

```
---------------------------------------------------------------------
NameError                           Traceback (most recent call last)
/tmp/ipykernel_1604/434711600.py in <cell line: 0>()
      1 score = 10
----> 2 print(scroe)

NameError: name 'scroe' is not defined
```

Grammar bugs look a little different. Here Python shows the line and points at the exact spot with little arrows, ^:

**What Colab shows under the cell**

```
  File "/tmp/ipython-input-2961531340.py", line 1
    if 5 > 3
            ^
SyntaxError: expected ':'
```

**The Detective's Routine**

1. **Read the LAST line first.** The word before the colon is the **kind** of bug (which animal in the zoo). The words after it are the clue.
2. **Find the line number.** `line 2`, or the arrow `---->` next to a line, tells you which room the crime happened in. Switch on line numbers in Colab: **Tools**, then **Settings**, then **Editor**, then **Show line numbers**.
3. **Look at the markers.** For grammar bugs, Python puts ^ markers under the exact part of the line it did not like. For running bugs, Colab points at the line with an arrow `---->` instead.
4. **Read the hint.** Some clues carry a hint, like `expected ':'`. If the clue says a name is **not defined**, check the spelling and the capital letters very carefully: Colab won't guess the right word for you.

**Only one clue at a time**

Python stops at the **first** problem it finds. If your code has three bugs, you will meet them one at a time: fix one, run again, read the next clue. That is completely normal.

Under the error, Colab also shows an **Explain error** button. Be the detective first: read the last line, make your guess, and only then press it to check.

## The clue decoder

| The last line says | It means | Try this |
|---|---|---|
| `SyntaxError` | Python can't read the line. | Look for a missing `:`, bracket `)` or quote mark `"`. |
| `IndentationError` | A line isn't lined up properly. | Lines inside a loop, `if` or `def` need four spaces in front. |
| `NameError` | Python doesn't know that word. | Check spelling and capitals. Did you make the variable first? Did you run the cells above? |
| `TypeError` | You mixed up text and numbers. | Use `int( )` to make a number, or `str( )` to make text. |
| `ValueError` | That value doesn't make sense here. | Did you type a word where a number was needed? |
| `IndexError` | That item number doesn't exist. | Python counts from 0. Check with `len( )`. |
| `ZeroDivisionError` | You divided by zero. | Check the number is not 0 before you divide. |
| `AttributeError` | That sprite doesn't have that command. | Check the spelling after the dot, as in `cat.forward`. |
| `ModuleNotFoundError` | The turtle tools aren't loaded. | Run the "In the beginning" cell again. |
| `KeyboardInterrupt` | You pressed stop. | Nothing is broken! Python is just saying "you stopped me". |

## Welcome to the Bug Zoo!

Here are the twelve bugs you will meet most often. Each card shows the Scratch habit that lets the bug sneak in, a tiny broken program, the **real** clue Python gives (copied from Python 3.13, the version Colab uses), and how to fix it. Try typing each broken program into Colab yourself to see the bug in the wild!

### Bug 1: Missing colon — *the Colon Cobra*

**Scratch habit:** the C-shaped if block opens its own mouth. In Python, the colon `:` is the mouth opening, and you must type it.

```
if 5 > 3
    print("yes")
```

```
    if 5 > 3
            ^
SyntaxError: expected ':'
```

**Fix it:** add `:` at the end of every `if`, `for`, `while` and `def` line: `if 5 > 3:`

### Bug 2: Missing spaces — *the Wobbly Stack*

**Scratch habit:** blocks snap **inside** the C-block by themselves. In Python, four spaces at the start of a line put it inside.

```
for i in range(3):
print(i)
```

```
    print(i)
    ^^^^^
IndentationError: expected an indented block after 'for' statement on line 1
```

**Fix it:** put four spaces (or press Tab) before `print(i)`. Python even tells you which line opened the mouth.

### Bug 3: Spaces where none belong — *the Wandering Stray*

**Scratch habit:** you can drag a block anywhere on the Scratch screen. In Python, a line with spaces in front must be inside a C-block.

```
print("Hello")
    print("there")
```

```
    print("there")
IndentationError: unexpected indent
```

**Fix it:** line 2 is not inside anything, so delete its spaces. No arrows this time: the clue is the word **unexpected**.

### Bug 4: Spelling slip — *the Misspelled Monkey*

**Scratch habit:** you pick variables from the palette, so you can never misspell one. In Python you type every name.

```
score = 10
print(scroe)
```

```
    print(scroe)
          ^^^^^
NameError: name 'scroe' is not defined
```

**Fix it:** spell it exactly. Turtle commands are names too: `forwad(100)` gives `NameError: name 'forwad' is not defined`

### Bug 5: Capital letters — *the Capital Copycat*

**Scratch habit:** block words are already written for you. To Python, `Print` and `print` are as different as `cat` and `dog`.

```
Print("hi")
```

```
    Print("hi")
    ^^^^^
NameError: name 'Print' is not defined
```

**Fix it:** Python words are lowercase: `print`, `if`, `for`. A capital `If` gives just `SyntaxError: invalid syntax`. And `Name` and `name` are two different boxes!

### Bug 6: Missing quotes — *the Quote Thief*

**Scratch habit:** you type text into a white bubble, with no quote marks. In Python, quotes say "this is just text".

```
print(hello)
```

```
    print(hello)
          ^^^^^
NameError: name 'hello' is not defined
```

**Fix it:** `print("hello")`. Without quotes, Python thinks `hello` is a variable's name. Forget only the **closing** quote, `print("hello)`, and you get `SyntaxError: unterminated string literal`: "you started some text and never finished it".

### Bug 7: Missing bracket — *the Bracketless Bat*

**Scratch habit:** the holes in a block are drawn for you. In Python, the holes are brackets ( ), and every ( needs its ).

```
print("hi"
```

```
    print("hi"
          ^
SyntaxError: '(' was never closed
```

**Fix it:** add the closing bracket: `print("hi")`. Its cousin, `print "hello"`, gives `SyntaxError: Missing parentheses in call to 'print'. Did you mean print(...)?`

### Bug 8: One equals instead of two — *the Equals Eel*

**Scratch habit:** the green ( ) = ( ) block asks a question, and set is a different block. Python uses `=` to set and `==` to ask.

```
x = 5
if x = 5:
    print("five")
```

```
    if x = 5:
       ^^^^^
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

**Fix it:** `if x == 5:`. Ignore the `:=` part of the hint; that is a tool for much later.

### Bug 9: Text plus a number — *the Glue Goblin*

**Scratch habit:** join glues anything to anything. Python will not glue text to a number with `+`.

```
score = 10
print("Score: " + score)
```

```
    print("Score: " + score)
          ~~~~~~~~~~^~~~~~~
TypeError: can only concatenate str (not "int") to str
```

**Fix it:** "concatenate" means glue, `str` means text, `int` means whole number. Use a comma, `print("Score:", score)`, or turn the number into text: `"Score: " + str(score)`. The same goblin bites `input() + 1`, because `input()` always gives text.

### Bug 10: A word that is not a number — *the Number Faker*

**Scratch habit:** Scratch treats a word as 0 and carries on. Python's `int()` can read `"5"` but not `"five"`.

```
age = int("five")
```

```
    age = int("five")
ValueError: invalid literal for int() with base 10: 'five'
```

**Fix it:** ignore "literal" and "base 10"; the clue is `'five'`. You get the same clue if someone types `five` into `int(input("How old are you? "))`. Type digits: 5.

### Bug 11: An item that isn't there — *the Off-by-One Owl*

**Scratch habit:** lists start at item 1, and **item 99 of list** just gives a blank. Python counts from **0**.

```
fruits = ["apple", "banana", "cherry"]
print(fruits[3])
```

```
    print(fruits[3])
          ~~~~~~^^^
IndexError: list index out of range
```

**Fix it:** "index" means item number. The three fruits are `fruits[0]`, `fruits[1]` and `fruits[2]`. The last one is also `fruits[-1]`. Check how many with `len(fruits)`.

### Bug 12: Dividing by zero — *the Zero Zombie*

**Scratch habit:** Scratch says **Infinity** and keeps going. Sharing 12 sweets among 0 friends makes no sense, so Python stops.

```
sweets = 12
friends = 0
print(sweets / friends)
```

```
    print(sweets / friends)
          ~~~~~~~^~~~~~~~~
ZeroDivisionError: division by zero
```

**Fix it:** check first with `if friends != 0:` (`!=` means "is not equal to"), and only divide inside the `if`.

## Sneaky bugs: no red writing!

The sneakiest animals in the zoo make no noise at all. The program runs, there is no error, but it does the **wrong thing**. These are just like Scratch bugs, so you already know how to hunt them: run the program and check what it **really** did.

**The Lost Hooray** *wrong indentation level*

```
for i in range(3):
    print("Hip hip")
    print("Hooray!")
```

This prints "Hooray!" three times. If you only wanted one at the end, take the spaces off line 3. The spaces decide what is inside the loop, and Python never warns you if you pick the wrong level.

**The Picky Parrot** *"Yes" is not "yes"*

```
answer = input("Like pizza? ")
if answer == "yes":
    print("Me too!")
```

Type `Yes` with a capital Y and nothing is printed. Scratch thinks `Yes` and `yes` are the same; Python does not. Fix: `if answer.lower() == "yes":` (`.lower()` turns the answer into small letters first).

**The Zero Hero** *counting from 0*

```
fruits = ["apple", "banana"]
print(fruits[1])
```

You wanted the first fruit, but it prints `banana`. Scratch's **item 1** is Python's `[0]`.

**The Forever Phantom** *a loop that never ends*

```
count = 0
while count < 3:
    print("hi")
```

It prints `hi` again and again and never stops, because `count` stays 0. Press the ■ stop button on the cell (it replaces the play button while the cell runs). Then add `count = count + 1` inside the loop: Scratch's **change count by 1**.

**The Silent Recipe** *forgetting the brackets*

```
def cheer():
    print("Hooray!")

cheer
```

No "Hooray!". Without `()`, you only **named** the recipe; you didn't cook it. Colab may show a strange line starting `<function`. Fix: `cheer()`.

**The Rounding Riddle** *round(2.5) is 2*

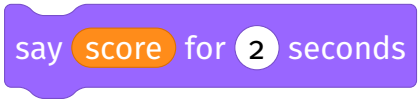
```
print(round(2.5))
print(round(3.5))
```

This prints 2 and then 4! Scratch rounds 2.5 up to 3. Python rounds an exact half to the nearest **even** number. This is not really a bug in Python, but it can make a bug in your program, so now you know.

## Detective tricks

Scratch coders who are good at fixing bugs use the same tricks again and again. Researchers Karen Brennan and Mitchel Resnick asked young Scratchers how they hunt bugs, and these are the tricks they named. They work just as well in Python.

**1. Make it say what it knows.** In Scratch you can add a say block to see what is inside a variable. In Python, add a `print`:




```
print("score is", score)
```

Put one before and one after the line you suspect. When the number is not what you expected, you have found the scene of the crime. Delete the extra `print` lines when you are done.

**2. Find where the problem is.** Read the line number, or run the program in small pieces. Split a big cell into smaller cells and run them one at a time, like clicking one Scratch block to test it.

**3. Read it out loud.** Explain your code, line by line, to a grown-up, a friend or a rubber duck. Programmers really do keep a toy duck on their desk for this! Very often you hear the bug yourself halfway through.

**4. Change one thing at a time.** Make one change, run, look. If you change five things at once, you won't know which one helped (or which one made a new bug).

**5. Compare with a working example.** Find a program in this book that works and does something similar. Put them side by side and spot the difference, like a spot-the-difference puzzle.

**6. Take a break.** Get a drink, stretch, look out of the window. Brains keep working on puzzles in the background, and the bug is often obvious when you come back.

**7. Tell someone.** Asking for help is a real programmer's skill. Say what you wanted, what happened instead, and what the last line of the error says.

## Bug hunt! Catch them all

Each program below has a bug: some are loud (red writing), some are sneaky (no error, wrong result). Type each one into Colab, run it and read the clue. Then fix it, and write what was wrong on the lines. Before you run it, can you **predict** which zoo animal will appear? (Answers are at the back of the book, but try first!)

**Bug hunt 1** It should say Meow 5 times.

```
count = 1
while count <= 5:
print("Meow number", count)
    count = count + 1
```

**Bug hunt 2** It should show your score.

```
score = 0
score = score + 10
Print("Your score is", score)
```

**Bug hunt 3** Type 9 when it asks.

```
age = input("How old are you? ")
print("Next year you will be", age + 1)
```

**Bug hunt 4** Type `cat123` when it asks.

```
password = input("Password? ")
if password = "cat123":
    print("Come in!")
```

---

---

**Bug hunt 5** It should print the first and the last fruit. (Two bugs!)

```
fruits = ["apple", "banana", "cherry"]
print("First:", fruits[1])
print("Last:", fruits[3])
```

---

---

**Bug hunt 6** It should share the sweets fairly, or say there is nobody to share with.

```
sweets = 20
friends = 0
print("Each friend gets", sweets / friends)
```

---

---

**Bug hunt 7** It should cheer.

```
def cheer():
    print("Hip hip hooray!")

cheer
```

---

---

**Bug hunt 8** It should count down 5, 4, 3, 2, 1 and blast off. (Have the stop button ready!)

```
n = 5
while n > 0:
    print(n)
print("Blast off!")
```

---

---

**Bug hunt 9** It should draw a square. (Put the three starting lines above it.)

```
for i in range(4):
    forward(50)
right(90)
```

---

---

**Bug hunt 10** Type your name when it asks.

```
name = input("What is your name? ")
print("Hello, " + name)
print("Nice to meet you!)
```

---

---

## Bug Zoo checklist

Tick each animal when you have caught it in your own code. Every bug you catch makes you a better detective!

- ☐ Colon Cobra
- ☐ Wobbly Stack
- ☐ Wandering Stray
- ☐ Misspelled Monkey
- ☐ Capital Copycat
- ☐ Quote Thief
- ☐ Bracketless Bat
- ☐ Equals Eel
- ☐ Glue Goblin
- ☐ Number Faker
- ☐ Off-by-One Owl
- ☐ Zero Zombie
- ☐ Lost Hooray
- ☐ Picky Parrot
- ☐ Zero Hero
- ☐ Forever Phantom
- ☐ Silent Recipe
- ☐ Rounding Riddle

☐ I read the **last line** of an error first.

☐ I can find the line number and the ^^^ markers.

☐ I can stop a forever loop with the stop button.

☐ I use `print` to make my program say what it knows.

☐ I change one thing at a time, and I take a break when I'm stuck.

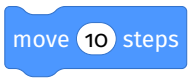
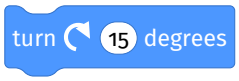
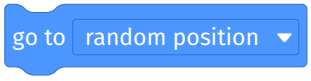
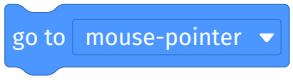
LOOK IT UP

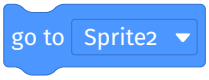
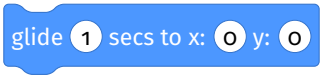
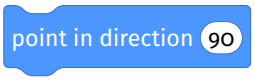
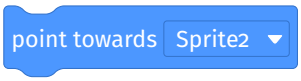
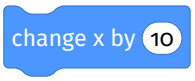
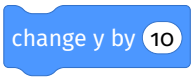
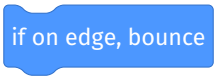
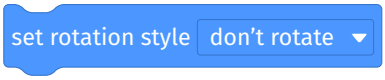
# The Block Dictionary

Know the Scratch block but not the Python? Find it here.

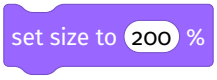
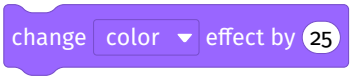
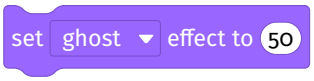
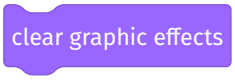
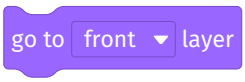
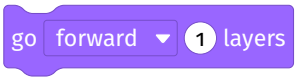
Every Scratch 3.0 block is in this dictionary, in the same order as the Scratch palette: **Motion**, **Looks**, **Sound**, **Events**, **Control**, **Sensing**, **Operators**, **Variables**, **Lists**, **My Blocks**, **Pen** and a little bit of **Music**.

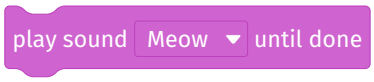
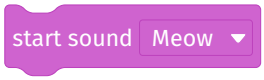
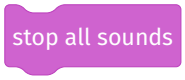
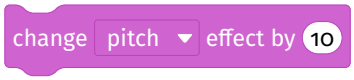
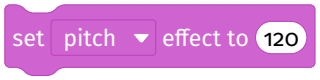
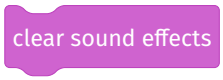
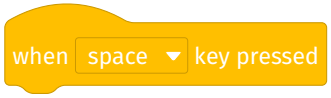
- The Python column uses plain commands, like `forward(10)`. They move **the turtle**, which plays the part of your sprite. Put the three starting lines at the top of your cell first.
- `cat` is a second sprite that you made with `cat = Turtle()`. Then `cat.forward(10)` moves the cat.
- Some blocks need a line at the very top of your cell: `import random`, `import time` or `import math`.
- Some blocks can't work in Colab. The Tip column always says what to do instead.

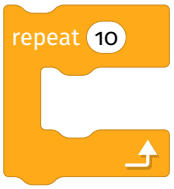
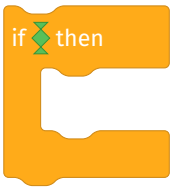
| Scratch | Python | Tip |
|---|---|---|
| **Motion** | | |
|  | `forward(10)` | Backwards: `back(10)`. The turtle draws as it moves, unless you use `penup()`. |
|  | `right(15)` | Turning works just like Scratch. |
|  | `left(15)` | |
|  | `x = random.randint(`<br>`  -240, 240)`<br>`y = random.randint(`<br>`  -180, 180)`<br>`goto(x, y)` | Put `import random` at the top. A line can carry on inside a bracket. |
|  | `x = int(input("x? "))`<br>`y = int(input("y? "))`<br>`goto(x, y)` | Not possible in Colab: it has no mouse. Type the place instead. |

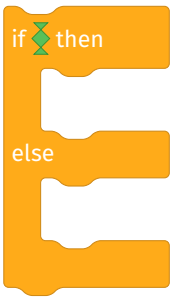
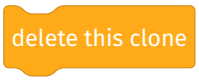
| Scratch | Python | Tip |
|---|---|---|
|  | `goto(cat.pos())` | `pos()` gives a sprite's x and y together. |
|  | `goto(0, 0)` | Same numbers as Scratch: the stage is the same size. `home()` goes to (0, 0) and points right. |
|  | `speed(2)`<br>then the **go to random** lines | Any **go to** becomes a glide when the speed is slow. |
|  | `speed(1)`<br>`goto(0, 0)` | Speed goes from 1 (slow) to 13 (fastest). It is not in seconds. |
|  | `setheading(0)` | **The 90 minus rule!** Use `setheading(90 - d)`: up 0 is 90, right 90 is 0, down 180 is $-90$, left $-90$ is 180. |
|  | `p = cat.pos()`<br>`d = towards(p)`<br>`setheading(d)` | No rule needed here: `towards` already uses Python's compass. Mouse-pointer: not in Colab. |
|  | `setx(xcor() + 10)` | Take x, add 10, then set it. |
|  | `setx(0)` | |
|  | `sety(ycor() + 10)` | |
|  | `sety(0)` | |
|  | `h = heading()`<br>`if abs(xcor()) > 240:`<br>`  setheading(180-h)`<br>`if abs(ycor()) > 180:`<br>`  setheading(-h)` | Python has no walls, so you check the edges yourself. Put these lines inside your loop, after `forward`. |
|  | `shape("circle")` | A turtle always points the way it goes. A round shape looks the same every way, so it seems not to turn. |

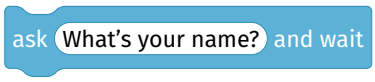
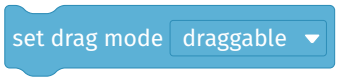
| Scratch | Python | Tip |
|---|---|---|
| x position y position | `xcor()`<br>`ycor()` | To print a tidy number: **`round`**`(xcor())` |
| direction | `90 - heading()` | The 90 minus rule works backwards too. Below $-180$? Add 360. |
| **Looks** | | |
| say Hello! for 2 seconds | `print("Hello!")`<br>`time.sleep(2)` | Put **`import`** `time` at the top. |
| say Hello! | `print("Hello!")` | The words appear under the cell. To write on the stage: `write("Hello!")` |
| think Hmm... for 2 seconds | `print("(Hmm...)")`<br>`time.sleep(2)` | There are no bubbles in Python. Brackets can show it is a thought. |
| think Hmm... | `print("(Hmm...)")` | |
| switch costume to turtle | `shape("turtle")` | Costumes are shapes: `"turtle"`, `"arrow"`, `"circle"`, `"square"`, `"triangle"`, `"classic"`. No pictures in Colab. |
| next costume | `n = (n + 1) % 4`<br>`shape(costumes[n])` | First make a list of 4 shapes called `costumes`, and `n = 0`. The `% 4` goes back to the start after the last one. |
| switch backdrop to blue sky | `bgcolor("skyblue")` | A plain colour, not a picture. Colour names use small letters. |
| next backdrop | `b = (b + 1) % 3`<br>`bgcolor(skies[b])` | Just like next costume, with a list of 3 colours called `skies`. |
| change size by 10 | `size = size + 10`<br>`shapesize(size / 100)` | Keep the size in your own box, starting with `size = 100`. |

| Scratch | Python | Tip |
|---|---|---|
|  | `shapesize(2)` | 1 means 100%, so 2 means 200%. Never use 0: that is an error. |
|  | `fillcolor("gold")` | No effects in Python, so pick a new colour. `color("gold")` colours the turtle and the pen. |
|  | `fillcolor("pink")` | Not possible in Colab: no see-through turtles. Try a paler colour, or `hideturtle()` for 100. Fisheye, whirl, pixelate and mosaic can't work either. |
|  | `color("black")` | A new turtle is black. |
|  | `showturtle()` | |
|  | `hideturtle()` | A hidden turtle can still draw. |
|  | Make this sprite **last** | No layer command. The sprite you make last is drawn on top. |
|  | Change the order you make sprites | Lines and writing are always under the turtles. |
|  | `n + 1`<br>`shape()` | Python counts from 0, so add 1. `shape()` with empty brackets gives the costume's name. |
|  | `b + 1`<br>`bgcolor()` | `bgcolor()` with empty brackets gives the colour. |
|  | `size` | Your own box from **change size**. |

| Scratch | Python | Tip |
|---|---|---|
| **Sound** | | |
|  | `beep(262)`<br>`time.sleep(0.5)` | `beep` is the helper from Part 7. Colab doesn't wait for a sound, so you wait yourself. No Meow: it plays notes. |
|  | `beep(262)` | Python carries on at once. |
|  | The stop button | Beeps are short, so they stop by themselves. |
|  | `freq = freq * 1.06`<br>`beep(freq)` | Keep the note in a box called `freq`. Times 1.06 is one Scratch "10". |
|  | `freq = 523` | 262 is middle C; 523 is the C above it (Scratch pitch 120). |
|  | `freq = 262` | Back to middle C. |
|  | Use the volume on your computer | `beep` has no volume knob, so turn your speakers up or down. |
| **Events** | | |
|  | The play button | Or press Shift and Enter. Python starts at line 1 of the cell. |
|  | `k = input("Key? ")`<br>`if k == "j":`<br>`    jump()` | Not possible in Colab: try `input()`, then press Enter. Real key presses work in Thonny with `onkeypress` (not `onkey`). |
|  | `c = input("Click? ")` | No mouse in Colab, so ask with `input()`. Thonny can do clicks with `onclick`. |

| Scratch | Python | Tip |
|---|---|---|
|  | `def night():` | Make your own block, and call `night()` straight after you change the `bgcolor`. |
|  | `input("Clap! ")` | Not possible: Colab can't hear you. Pressing Enter can be your pretend clap. For **timer**, see the Sensing timer row. |
|  | `def start():` | A message is like your own block. The lines inside get four spaces. |
|  | `start()` | Python runs the whole block, then carries on. |
|  | `start()` | The same! Python always waits for a block to finish. |
|  | `time.sleep(1)` | Put `import time` at the top. |
|  | `for i in range(10):`<br>`  forward(10)` | The four spaces are the mouth of the C. Use whole numbers: `range(2.5)` is an error. |
|  | `while True:`<br>`  forward(10)` | It never stops by itself, so press ■. In Colab a big `for` loop like `range(200)` is safer. |
|  | `if score > 5:`<br>`  print("Wow")` | The colon opens the mouth. |

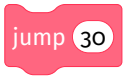
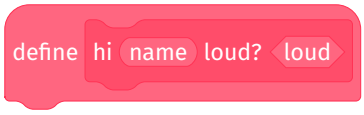
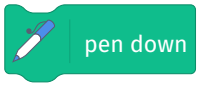
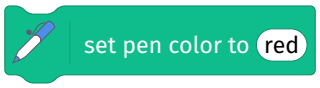
| Scratch | Python | Tip |
|---|---|---|
|  | `if score > 5:`<br>`    print("big")`<br>`else:`<br>`    print("small")` | `else:` lines up with the `if`. Python also has `elif` for "else if", which Scratch doesn't. |
|  | `a = ""`<br>`while a != "yes":`<br>`    a = input("Go? ")` | Keeps asking until you type `yes`. Python does one thing at a time, so something in the loop must change. |
|  | `while score != 5:` | Flipped! Repeat **until** it is 5 means keep going **while** it is not 5. |
|  | `sys.exit()` | Put `import sys` at the top. Colab shows a little warning: that's fine. |
|  | `break`<br>`return` | `break` jumps out of a loop. `return` leaves your own block. |
|  | Not needed | Python only runs one script at a time. |
|  | `s = Turtle()`<br>`s.shape("circle")`<br>`s.goto(0, 100)` | The lines straight after `Turtle()` set up each new sprite. |
|  | `star = Turtle()`<br>or, for lots:<br>`stars = []`<br>`s = Turtle()`<br>`stars.append(s)` | This makes a new sprite and gives it a name. For many clones, keep them in a list. |
|  | `s.hideturtle()`<br>`stars.remove(s)` | Hide it, then take it out of the list. |

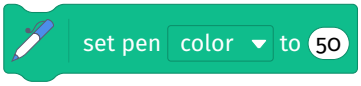
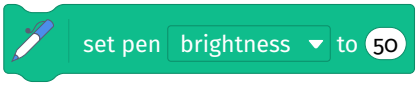
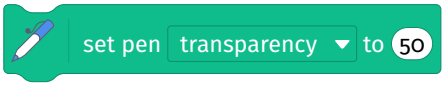
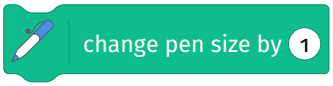
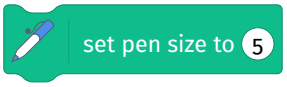
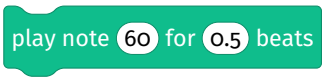
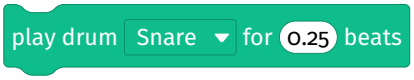
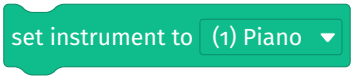
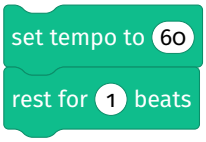
| Scratch | Python | Tip |
|---|---|---|
| **Sensing** | | |
|  | `p = cat.pos()`<br>`distance(p) < 20` | Asks: is the cat closer than 20 steps? Edge: `abs(xcor()) > 240` |
|  | `ycor() < -100` | Not possible: turtles can't see colours. Remember where things are instead: "the lava is below $y = -100$". |
|  | Use places, not colours | The same trick as **touching color**. |
|  | `distance(cat.pos())` | Mouse-pointer: not in Colab. |
|  | `name = input("Who? ")` | You choose the name of the box. |
|  | `name` | `input` always gives **writing**. For a number: `int(name)` |
|  | `k = input("Key? ")`<br>`k == " "` | Colab can't feel keys. Type, then press Enter. Thonny can: `onkeypress`. |
|  | `input("Click? ")` | Not possible in Colab: there is no mouse. Ask instead. |
|  | `x = int(input("x? "))` | Type the numbers instead. |
|  | Not in Colab | Try it in Thonny: `ondrag(goto)` |
|  | `input("How loud? ")` | Not possible: Colab can't use your microphone. Ask for a number instead. |
|  | `time.time() - start` | First write `start = time.time()`. It's your own stopwatch. |
|  | `start = time.time()` | Write down a new start time. |
|  | `cat.xcor()` | Also `cat.ycor()` and `cat.heading()`. |

| Scratch | Python | Tip |
|---|---|---|
| current year | `d = time.localtime()`<br>`d.tm_year` | Also `d.tm_mon`, `d.tm_mday`, `d.tm_hour`, `d.tm_min`. Colab's clock may be on London time. |
| days since 2000 | `s = time.time()`<br>`s / 86400 - 10957` | Seconds since 1970 become days; then take away the days from 1970 to 2000. |
| username | `name = input("Who? ")` | Colab doesn't know your name, so ask! |
| **Operators** | | |
| 7 + 2 7 - 2<br>7 * 2 7 / 2 | `7 + 2  7 - 2`<br>`7 * 2  7 / 2` | `/` always gives a decimal: `6 / 3` is `2.0`. Dividing by 0 is an error. |
| pick random 1 to 10 | `random.randint(1, 10)` | Put `import random` at the top. 1 and 10 can both come up. |
| a > 50 | `a > 50` | "Bigger or equal" is `a >= 50`. |
| a < 50 | `a < 50` | |
| a = 50 | `a == 50` | Two equals signs to check! Capitals count: `"Cat" == "cat"` is `False`. |
| and | `a > 1 and a < 9` | Small letters: `and` |
| or | `a < 1 or a > 9` | |
| not | `not a == 5` | Same as `a != 5` |
| join apple banana | `"apple " + "banana"` | Joining a number? Use `str(5)`. |
| letter 1 of apple | `"apple"[0]` | Python counts from 0! Letter n is `[n - 1]`. |
| length of apple | `len("apple")` | |
| apple contains a? | `"a" in "apple"` | The order flips round. Capitals count. |
| 7 mod 3 | `7 % 3` | The remainder. Even numbers: `n % 2 == 0` |

| Scratch | Python | Tip |
|---|---|---|
| round 2.5 | `round(2.5)`<br>`math.floor(x + 0.5)` | Careful! `round(2.5)` gives 2, but Scratch gives 3. The second line rounds like Scratch. |
| abs of -5 | `abs(-5)` | No import needed. |
| floor of 2.7<br>ceiling of 2.1 | `math.floor(2.7)`<br>`math.ceil(2.1)` | Put `import math` at the top. Ceiling is called `ceil`. |
| sqrt of 16 | `math.sqrt(16)` | |
| sin of 30<br>cos tan | `r = math.radians(30)`<br>`math.sin(r)` | Python measures angles in radians, so change the degrees first. Same for `cos` and `tan`. |
| asin of 0.5<br>acos atan | `a = math.asin(0.5)`<br>`math.degrees(a)` | Change the answer back into degrees. |
| ln of x log of x | `math.log(x)`<br>`math.log10(x)` | Python's `log` is Scratch's `ln`. |
| e ^ of x 10 ^ of x | `math.exp(x)`<br>`10 ** x` | Power is `**`. The ^ sign means something else in Python. |
| **Variables** | | |
| **Make a Variable** | `score = 0` | No button: the first = makes the box. |
| score | `score` | Use the name anywhere a number goes. |
| set score to 0 | `score = 0`<br>`name = "Ada"` | = puts a value into the box. Writing needs quote marks. |
| change score by 1 | `score = score + 1`<br>`score += 1` | Both lines do the same. Inside your own block, write `global score` first. |
| show variable score | `print(score)` | It shows the value once. Print again to see a new value. |
| hide variable score | Just don't print it | |

| Scratch | Python | Tip |
|---|---|---|
| **Lists** | | |
| **Make a List** | `food = []` | Square brackets make an empty list. |
| food | `" ".join(food)` | All the items, with spaces between. |
| add apple to food | `food.append("apple")` | Adds to the end. |
| delete 1 of food | `del food[0]` | Counting from 0! Item n is `[n - 1]`. Last: `food.pop()` |
| delete all of food | `food.clear()` | |
| insert fig at 1 of food | `food.insert(0, "fig")` | |
| replace item 1 of food with kiwi | `food[0] = "kiwi"` | |
| item 1 of food | `food[0]` | The first item is number 0! Last item: `food[-1]`. Random: `random.choice(food)` |
| item # of fig in food | `food.index("fig") + 1` | Add 1, because Python counts from 0. Check with `in` first: a missing item is an error. |
| length of food | `len(food)` | |
| food contains apple? | `"apple" in food` | Capitals count: `"Apple"` is different. |
| show list food | `print(food)` | |
| hide list food | Just don't print it | |
| **My Blocks** | | |
| define jump | `def jump():`<br>`  forward(20)` | The lines inside get four spaces. Run this cell before you use the block. |
| jump | `jump()` | The brackets make it run. |

| Scratch | Python | Tip |
|---|---|---|
|  | `def jump(height):`<br>`  forward(height)` | The input is a box that only works inside the block. |
|  | `jump(30)` | |
|  | `def hi(name, loud):`<br>`  if loud:`<br>`  print("HI!")` | Call it with `hi("Ada", True)`. `True` and `False` start with capitals. |
|  | `speed(13)` | The fastest speed that still shows the drawing. |
| A block that gives an answer back (Scratch can't do this!) | `def double(n):`<br>`  return n * 2` | `print(double(4))` shows 8. Like a round reporter block you made yourself. |
| <br> | `clear()` | Wipes this turtle's drawing. `clearscreen()` wipes everything, sprites too. |
|  | `stamp()` | Stamps the turtle's shape. `clearstamps()` rubs out all stamps: Scratch can't do that! |
|  | `pendown()` | The turtle starts with its pen already down. |
|  | `penup()` | Use it to move without drawing. |
|  | `pencolor("red")` | Colour names in small letters. `color("red")` colours the turtle and the pen. |
|  | `c = (c + 1) % 3`<br>`pencolor(rainbow[c])` | First make a list of 3 colours called `rainbow`, and `c = 0`. |

| Scratch | Python | Tip |
|---|---|---|
|  | `pencolor("cyan")` | Scratch numbers: 0 red, 8 orange, 16 yellow, 33 green, 50 cyan, 66 blue, 75 purple. |
|  | `pencolor("darkred")` | Saturation and brightness: pick a paler or darker name, like `"pink"` or `"darkred"`. |
|  | `pencolor("pink")` | Not possible for the pen. A paler colour looks a bit see-through. |
|  | `w = pensize() + 1`<br>`pensize(w)` | `pensize()` with empty brackets gives the size now. |
|  | `pensize(5)` | |
| No Scratch block: a bonus! | `fillcolor("gold")`<br>`begin_fill()`<br>`circle(50)`<br>`end_fill()` | Colours in a shape. |
| **Music**<br> | `beep(262)`<br>`time.sleep(0.5)` | Note 60 is middle C (262); E is 330. Note n:<br>`f = 2 ** ((n-69)/12)`<br>`beep(440 * f)` |
|  | `beep(100, 0.1)`<br>`time.sleep(0.25)` | A very low, short note sounds a bit like a drum. |
|  | Not possible with `beep` | `beep` has one sound. Try short and long notes instead. |
|  | `beat = 60 / 60`<br>`time.sleep(beat)` | Tempo 60 means one beat each second; a rest is a wait. A smaller `beat` is faster. |
| A tune | `tune = [262, 330]`<br>`for f in tune:`<br>`  beep(f)`<br>`  time.sleep(0.5)` | A list of notes and a loop make a tune! Add more notes to the list. |

KEEP THIS PAGE NEAR YOU

# My Python cheat sheet

## Start a drawing: the three starting lines

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
```

Once per session, first: `!pip install ColabTurtlePlus`

## Move

```
forward(100)  back(50)
right(90)  left(90)
goto(0, 0)
setheading(90 - d) (the 90 minus rule)
speed(13) (1 slow to 13 fast)
```

## Pen

```
penup()  pendown()
pencolor("red")
pensize(5)
clear()  stamp()
bgcolor("skyblue")
```

## Say and ask

```
print("Hi")
write("Hi") (on the stage)
name = input("Who? ")
age = int(input("Age? "))
```

`input` always gives writing!

## Loops

```
for i in range(4):
    forward(100)
while score != 5:
    score = score + 1
```

## If

```
if guess < secret:
    print("Too low")
elif guess > secret:
    print("Too high")
else:
    print("Yes!")
```

## Variables

```
score = 0
score = score + 1
score += 1 (the same)
print("Score: " + str(score))
```

## Lists

```
food = ["apple", "fig"]
food.append("kiwi")
food[0] (the first item!)
len(food)  "fig" in food
for f in food:
    print(f)
```

## Operators

```
+  -  *  /  7 % 3 (remainder)
==  !=  <  >  <=  >=
and  or  not
import random
random.randint(1, 10)
```

### My own block

```
def square(size):
    for i in range(4):
        forward(size)
        right(90)
square(50)
def double(n):
    return n * 2
print(double(4)) (shows 8)
```

### Extra sprites

```
cat = Turtle()
cat.shape("turtle")
cat.goto(-100, 0)
cat.forward(50)
cat.distance(pos()) < 20 (touching?)
```

### Wait and sound

```
import time
time.sleep(1) (wait 1 second)
beep(262) (middle C, helper from Part 7)
time.sleep(0.5)
```

### Remember

**:** A colon opens a loop, `if` or `def`.

**Four spaces** put a line inside, like the mouth of a C block.

**=** puts into a box. **==** checks. **!=** means "not equal".

**"Words"** need quote marks. Numbers don't.

Python counts from **0**: item 1 is `[0]`.

Capitals matter: `print`, not `Print`, and `"Cat" == "cat"` is **`False`**.

**`input()`** gives writing. Use **`int( )`** for a number.

**`round(2.5)`** is 2 in Python, but 3 in Scratch.

Directions: `setheading(90 - d)`.

Colour names in small letters: `"red"`.

Run the "In the beginning" cell each session.

Read the **last line** of an error first. It's a clue!

FOR GROWN-UPS

# Helping your coder

You don't need to know Python to help. These pages tell you how.

This book is made for two people: your child at the keyboard and you beside them. You do not need to know Python. Your job is to be the coach and the audience: ask questions, read error messages out loud together, and be impressed when something works.

Some of the advice below comes from research on children moving from block coding to text coding. Where a piece of advice is only our own experience and not a research finding, it is marked *(judgement)*. The papers are listed at the end of this section.

## How the book works

The book has seven Parts. They do **not** follow the order of the block colours in Scratch. They follow the order that research suggests for moving from blocks to text:

| | Part | Why it comes here |
|---|---|---|
| 1 | **Motion and Pen** | Drawing gives a picture straight away, and it matches blocks your child already knows. Running a cell plays the part of the green flag. |
| 2 | **Loops** | Shapes need **repeat** at once. Loops are harder in text than in blocks, so they get early practice. |
| 3 | **Variables and Lists** | Scores, names and sizes, as soon as a project needs them. |
| 4 | **Operators** | Maths, random numbers and comparisons, which the next Part needs. |
| 5 | **Sensing and Decisions** | `input()` and `if`: quizzes and guessing games. |
| 6 | **My Blocks** | Your child's own super-blocks, written with `def`. |
| 7 | **Looks, Sound and Events** | Last, on purpose (see below). |

**Why events come last.** In Scratch, **when green flag clicked**, **when space key pressed** and many scripts running at once cost nothing. In text code they are the hardest part. When Dann and colleagues (2012) taught students to move from the block language Alice to Java, events and scripts running at the same time were the hardest things to carry across. In

Python, a key press needs a *function* that runs when the key is pressed, so functions (Part 6) must come first.

After the seven Parts come three sections your child can use at any time:

- **Big Projects**: longer programs that use everything, for when the Parts are done.
- **Bug Zoo**: the common Python error messages, what they mean and how to fix them. Keep it open from the first day.
- **Block Dictionary**: look up a Scratch block and find its Python.

Every Part has the same shape: short pages that put each Scratch block next to its Python, then two to four **missions**, then **Translate it yourself**, a **Two-line puzzle** and an "I can…" checklist. Answers are at the back of the book.

## Running a session

- **Keep it short: 20 to 30 minutes.** End while something is working and can be shown to someone. Stop before frustration builds. A mission usually takes two sessions: one to get it working, one for the "Make it yours" challenges. *(judgement: the research gives no session lengths)*
- **Scratch first, every time.** Step 1 of each mission is building it in Scratch. Then the only new thing is the typing, and the jump feels small. *(judgement)*
- **Say the links out loud.** This is the step that matters most. Weintrop and Wilensky (2019) followed students who had started with blocks: their head start had faded once they moved to a text language, in a class where nobody linked the blocks to the code. The method with the best evidence is to put the blocks and the code side by side, point from each block to the line that does the same job, and then translate a script you have not seen before (Dann et al., 2012). The side-by-side pages, the "Read it line by line" tables and the **Translate it yourself** page in every Part are built for exactly this. Say things like "`for i in range(4):` is the repeat 4 block, and the spaces are the mouth of the C".
- **Predict, then run, then change.** Starting from code that works is easier than starting from an empty cell. Two well-known teaching methods use this order. PRIMM (Sentance, Waite and Kallia, 2019) has five steps: **P**redict what the code will do, **R**un it, **I**nvestigate ("what does this line do?"), **M**odify (change one number, colour or word), **M**ake (add something new). Use-Modify-Create (Lee et al., 2011) is the same idea in three steps. Before your child runs anything, ask "What do you think will happen?"
- **Let them type.** Copying code letter by letter is how children learn colons, brackets and spaces. Try not to take the keyboard or fix typos for them. *(judgement)*

- **Errors are normal.** Scratch never shows an error message, so your child has probably never seen one. Red text is a clue, not a failure. Read the **last line** out loud together, find the line number, and look the error up in the Bug Zoo. "Take a break" is a real debugging strategy that children reported to Brennan and Resnick (2012), so you can teach it as one.
- **Celebrate "real programmer" moments.** Students who moved to text code said they felt like real programmers, and that feeling motivates (Weintrop and Wilensky, 2017).

## The four-question check at the end of a Part

There are no tests in this book. Instead, when your child finishes a Part, pick one of their programs and ask four questions. They come from the interviews Brennan and Resnick (2012) used to see what children really understood:

1. **Explain:** "What does your program do? Show me the line that does ..."
2. **Extend:** "How could you make it bigger or better?"
3. **Fix a bug:** change one thing secretly (take away a colon, a space or a bracket) and ask them to find it.
4. **Remix:** "Add one new thing of your own."

If one of the four is hard, spend another session on that Part before moving on. Knowing what a loop *is* matters less than being able to use it, change it and fix it.

## A suggested pace

This is only a guide *(judgement)*. It assumes two or three sessions of 20 to 30 minutes a week. Go slower if a Part is hard; that is time well spent.

| | Part | Sessions | Weeks |
|---|---|---|---|
| | Getting ready (Colab, the starting lines) | 1–2 | 1 |
| 1 | Motion and Pen | 6–8 | 2–3 |
| 2 | Loops | 5–7 | 2–3 |
| 3 | Variables and Lists | 6–8 | 2–3 |
| 4 | Operators | 5–7 | 2–3 |
| 5 | Sensing and Decisions | 6–8 | 2–3 |
| 6 | My Blocks (functions) | 6–8 | 2–3 |
| 7 | Looks, Sound and Events | 6–8 | 2–3 |
| | Big Projects | 6–10 | 3–4 |
| | **Total** | **about 50–65** | **about 5 months** |

Parts 2 and 6 often need extra time. In studies of older students, questions about loops and about calling functions were answered better in blocks than in text (Weintrop and Wilensky, 2017).

## Colab tips

- **Google account.** Colab needs one. Accounts for children (Family Link or some school accounts) may be blocked from Colab. If so, sign in with your own account.
- **Saving.** Notebooks save automatically to your Google Drive, in a folder called **Colab Notebooks**. **File**, **Save a copy in Drive** makes a copy, which is handy before a big change.
- **When Colab resets.** After a while without use, Colab forgets everything, including the installed library. If you see `No module named 'ColabTurtlePlus'` or a `NameError` for something that worked before, open the **Runtime** menu and choose **Run all**. This is not your child's mistake.
- **Line numbers.** Turn them on so they match the book: **Tools**, **Settings**, **Editor**, **Show line numbers**.
- **The library.** The turtle in this book comes from **ColabTurtlePlus**. We tested everything in Colab on 7 November 2026 with Python 3.13 and ColabTurtlePlus 2.2.0. The "In the

beginning" cell installs it once per session, and every drawing program starts with the same three lines:

```
!pip install ColabTurtlePlus
```

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
```

`initializeTurtle((480, 360))` makes a stage exactly like Scratch's: (0, 0) in the middle, x from $-240$ to 240, y from $-180$ to 180.

**Four things that go wrong**

- **Install the right library.** There is an older library with a very similar name, **ColabTurtle** (without "Plus"). It is a different library: it has no `Turtle()`, it draws in a black 800 by 500 window, and its (0, 0) is at the top-left corner. If your drawing looks like that, check the first cell says `ColabTurtlePlus`.
- **Do not use** `mode("logo")`. Some tutorials online use it to make directions match Scratch. In ColabTurtlePlus it has a bug: we confirmed in Colab that `setheading(90)` then points *down*. The book stays in the normal mode and uses the rule "Python direction = 90 minus Scratch direction".
- **Use** `speed(13)` **for fast drawing, not** `speed(0)`. With 0, nothing appears until the program calls `done()`.
- **Colour names are lowercase:** `"red"`, `"skyblue"`, not `"Red"`.

**What Colab cannot do.** ColabTurtlePlus draws a picture inside the notebook. It cannot listen for key presses, mouse clicks or timers, and it has no sound or picture costumes. So in Colab the book uses `input()` (typing an answer and pressing Enter) for interaction. Real key-press games are for Thonny, below. Please tell your child this plainly, so they don't think they did something wrong.

## Sound in Colab

Colab runs on Google's computers, so Python cannot reach your speakers directly. It can send a sound to the web page, though. Part 7 uses this helper. Run it once in a cell (it does not need ColabTurtlePlus):

```
import numpy as np
from IPython.display import Audio, display

def beep(freq=440, secs=0.5):
    times = np.linspace(0, secs, int(22050 * secs))
    sound = np.sin(2 * np.pi * freq * times)
    display(Audio(sound, rate=22050, autoplay=True))
```

Then `beep(262)` plays middle C, and `beep(523, 1)` plays the C above it for one second. Some notes to try: C 262, D 294, E 330, F 349, G 392.

- **Autoplay.** The sound starts by itself because running the cell counts as a click. Some browsers (often Safari), or a muted tab, block this. Then a small player appears: press its play button.
- **Notes on top of each other.** Python finishes the cell before the browser starts playing, so several `beep`s in one cell can overlap. Put `time.sleep(0.6)` after each `beep` (with `import time` at the top), or put each note in its own cell.

## Beyond Colab

When your child wants real key-press games, or you have no Google account, these are the next places to go.

**Thonny** (`thonny.org`) is a free Python editor for Windows, Mac and Linux, made for beginners. It needs no account, works offline, and nothing leaves your computer. It includes Python's standard `turtle`, which opens its own window and *can* react to the keyboard. The turtle commands are the same as in Colab, and the "90 minus" direction rule still works. Here is a small coin game: the arrow keys steer the turtle, and touching the gold coin scores a point.

In Thonny, click on the turtle window first, then press the arrow keys. The score shows in the window's title bar. Three things to notice in the program below:

- `screen.onkeypress(go, "Up")` is **when up arrow key pressed**. Use `onkeypress`, not `onkey`: `onkey` waits until the key is let go.
- Write `go` without brackets. You are telling Python which function to run *later*, not running it now.
- `screen.listen()` switches the keyboard on, and `turtle.mainloop()` keeps the window waiting for keys. It must be the last line.

```
import turtle
import random

screen = turtle.Screen()
screen.setup(480, 360)
player = turtle.Turtle()
player.shape("turtle")
player.penup()
coin = turtle.Turtle()
coin.shape("circle")
coin.color("gold")
coin.penup()
coin.goto(100, 80)
score = 0

def check_coin():
    global score
    if player.distance(coin) < 20:
        score = score + 1
        screen.title("Score: " + str(score))
        x = random.randint(-200, 200)
        y = random.randint(-150, 150)
        coin.goto(x, y)

def go():
    player.forward(15)
    check_coin()

def turn_left():
    player.left(30)

def turn_right():
    player.right(30)

screen.onkeypress(go, "Up")
screen.onkeypress(turn_left, "Left")
screen.onkeypress(turn_right, "Right")
screen.listen()
turtle.mainloop()
```

This example uses only things from Parts 1 to 6 plus `global` (Part 6), which is why events are left until last.

**Pygame Zero, later.** Pygame Zero is a game library whose ideas match Scratch closely: an `Actor` is a sprite, `update()` is a forever loop, and `on_key_down` is a key-press hat. Thonny has a **Run**, **Pygame Zero mode** menu and installs it through **Tools**, **Manage packages**. One caution: a problem has been reported (Thonny issue #3158) where Pygame Zero mode fails on newer Python versions. Test it on your own computer before you promise your child a game.

**No computer to install on (iPad, Chromebook):** the **micro:bit Python editor** (`python.microbit.org`) runs in the browser, needs no account, and has a simulator, so you do not need a micro:bit board. It has buttons, sound and forever loops. It draws on a 5 by 5 grid of lights, not with a turtle. **MakeCode Arcade** (`arcade.makecode.com`) makes retro games and lets your child switch between blocks and a version of Python, which is a gentle bridge.

**Tools you may read about but should skip:** Trinket closed in August 2026, and the Mu editor is archived and no longer updated, according to their own notices.

## Showing your child their own Scratch project as Python

This is a wonderful moment: "Look, this is *your* project, in Python."

A Scratch project saved with **File**, **Save to your computer** is an `.sb3` file. An `.sb3` file is really a zip file. Inside are the costume pictures, the sounds, and one file called `project.json`, which holds all the scripts as text. Each block is stored with a name such as `motion_movesteps` (move steps) or `control_repeat` (repeat). Scratch has no "export as Python" button, but you can peek inside with Python itself. Upload the `.sb3` to Colab (the folder icon on the left, then the upload button), change the file name in line 3, and run:

```
1  import zipfile, json
2
3  with zipfile.ZipFile("my-project.sb3") as z:
4      project = json.loads(z.read("project.json"))
5
6  for sprite in project["targets"]:
7      print("Sprite:", sprite["name"])
8      for block in sprite["blocks"].values():
9          if isinstance(block, dict):
10             print("   ", block["opcode"])
```

For a square-drawing project it prints `event_whenflagclicked`, `control_repeat`, `motion_movesteps` and `motion_turnright`. The blocks may not come out in script order, but your child will recognise them.

**Which projects translate well?** Simple drawing projects (green flag, repeat, move, turn, pen, variables, ask, My Blocks) translate almost line by line into the Python in this book, and the Block Dictionary makes this a good activity to do together by hand. Scratch's stage and the turtle share the same coordinates, so the drawing comes out the same. Games built on picture costumes, sprites touching, clones or sound do not translate simply. For those, talk about the structure instead: a loop is a `for`, a My Block is a `def`. Rewriting one of their own projects with the Block Dictionary open is the real milestone of this book.

## References


- Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. Paper presented at the Annual Meeting of the American Educational Research Association, Vancouver, Canada. `https://scratched.gse.harvard.edu/ct/files/AERA2012.pdf`
- Dann, W., Cosgrove, D., Slater, D., Culyba, D., & Cooper, S. (2012). Mediated transfer: Alice 3 to Java. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*, 141–146.
- Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J., & Werner, L. (2011). Computational thinking for youth in practice. *ACM Inroads, 2*(1), 32–37. `https://doi.org/10.1145/1929887.1929902`
- Sentance, S., Waite, J., & Kallia, M. (2019). Teaching computer programming with PRIMM: A sociocultural perspective. *Computer Science Education, 29*(2–3), 136–176. `https://doi.org/10.1080/08993408.2019.1608781`
- Weintrop, D., & Wilensky, U. (2017). Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education, 18*(1), Article 3. `https://doi.org/10.1145/3089799`
- Weintrop, D., & Wilensky, U. (2019). Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. *Computers & Education, 142*, 103646. `https://doi.org/10.1016/j.compedu.2019.103646`



This is an independent workbook. It is not made by, endorsed by or connected with the Scratch Foundation, MIT or Google. Scratch is a project of the Scratch Foundation. Google Colab is a product of Google. ColabTurtlePlus, Thonny, Pygame Zero, the micro:bit Python editor and MakeCode Arcade belong to their own makers.

NO PEEKING UNTIL YOU HAVE TRIED!

# Answers

## Part 1: Point in a direction (the table)

| Scratch direction | Python code | Python direction | Which way? |
|---|---|---|---|
| 90 | `setheading(90 - 90)` | 0 | right |
| 0 | `setheading(90 - 0)` | 90 | up |
| -90 | `setheading(90 - -90)` | 180 | left |
| 45 | `setheading(90 - 45)` | 45 | up and right |
| 135 | `setheading(90 - 135)` | $-45$ | down and right |

Taking away a minus number is the same as adding, so $90 - (-90) = 180$. Python understands $-45$ too: it is the same direction as 315.

## Part 1: Translate it yourself

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
penup()
goto(-100, 0)
pendown()
pencolor("red")
pensize(4)
setheading(90 - 0)
forward(80)
right(90)
forward(80)
stamp()
```

The turtle draws a red line up from $(-100, 0)$, turns right, draws a line across, and stamps itself at $(-20, 80)$. Writing `setheading(90)` instead of `setheading(90 - 0)` also works.

If you added `shape("turtle")`, well done: your stamp looks like a turtle.

### Part 1: Two-line puzzle

```
setheading(90 - 180)
forward(100)
```

`setheading(-90)` works too, and so does `right(90)` instead of `setheading`, because the turtle starts facing right. A quarter turn clockwise makes it face down.

### Part 2, Draw shapes the short way: the shapes table

Pentagon: $360 \div 5 = 72$. Hexagon: $360 \div 6 = 60$. Octagon: $360 \div 8 = 45$. Decagon: $360 \div 10 = 36$. Almost a circle: $360 \div 36 = 10$.

The 36-sided shape looks most like a circle. With `pencolor("red")` inside the loop, the whole triangle comes out red: line 4 makes the pen blue, but the red line runs before every `forward`, so blue is never used. With `print("Ta-da!")` at the margin it prints once; with four spaces it is inside the loop and prints 3 times.

### Part 2, Stars: the star table

| Points | $180 \div$ points | Turn |
|---|---|---|
| 9 | 20 | 160 |
| 15 | 12 | 168 |
| 45 | 4 | 176 |

### Part 2, Square spiral: the missing line

The first time round, `i` is 0, so `forward(i * 5)` is `forward(0)`. The turtle doesn't move, so the first "line" has no length. With `range(60)` and `forward(i * 3)`, the longest line is $59 \times 3 = 177$ steps, close to the $39 \times 5 = 195$ steps of the first spiral, so the size stays about the same while the lines sit closer together.

### Part 2, Flower of squares

36 squares $\times$ 10 degrees $= 360$, and 12 squares $\times$ 30 degrees $= 360$: the turns between the squares must add up to one full circle. `hideturtle()` needs **no** spaces, so it runs once, after the whole flower.

### Part 2, Translate it yourself

Your code may look a little different and still be right. If the turtle draws the same picture, you did it!

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
pensize(4)
pencolor("purple")
for i in range(8):
    for j in range(6):
        forward(40)
        right(60)
    right(45)
penup()
print("Done!")
```

`forward(40)` and `right(60)` have 8 spaces (inside both loops). `right(45)` has 4 spaces. `penup()` and `print("Done!")` have none, because they come after the loops. You get 8 hexagons in a ring.

### Part 2, Two-line puzzle

```
for i in range(5):
    print(i + 1)
```

`i` goes 0, 1, 2, 3, 4, so `i + 1` goes 1, 2, 3, 4, 5.

## Part 3: Quick quiz, box or bust?

Python accepts `lives`, `player_1` and `bestTime`. `high score` has a space, `1st_place` starts with a digit, and `import` is a word Python already uses.

## Part 3, Mission: Score counter

`score += 10` does exactly the same as `score = score + 10`, so nothing changes. To count down, start with `score = 50` and use `score -= 10`: the last line says `Final score: 0`. With `lives -= 1` inside the loop, the lives go 2, 1, 0, then $-1$ and $-2$. Python doesn't stop at 0 unless you tell it to (you will learn how with **`if`** in Part 5).

**Super challenge:**

```
import time
score = 0
points = 10
print("Score:", score)
for i in range(5):
    time.sleep(1)
    score = score + points
    points = points + 10
    print("Score:", score)
print("Game over! Final score:", score)
```

The score goes 10, 30, 60, 100, 150. The final score is `150`.

## Part 3, Mission: Growing spiral

With `+ 8` the last line is 317 steps long, and the spiral only just fits on the stage. For the thicker pen, put `thick = 1` next to `size = 5`, and add `pensize(thick)` and `thick = thick + 0.2` inside the loop.

**Super challenge** (after the three starting lines):

```
speed(13)
pencolor("purple")
penup()
goto(-100, 100)
pendown()
size = 200
for i in range(40):
    forward(size)
    right(90)
    size = size - 5
```

### Part 3, Mission: Rainbow from a colour list

`colours.append("pink")` adds pink to the end of the list, so the loop goes round 7 times and pink is the bottom stripe. Red, white and blue stripes, from top to bottom, make the flag of the Netherlands. Try `["white", "blue", "red"]` for Russia.

**Super challenge:** lines 8 to 16 become

```
y = 60
length = 300
for colour in colours:
    pencolor(colour)
    penup()
    goto(-150, y)
    pendown()
    forward(length)
    y = y - 15
    length = length - 30
hideturtle()
```

### Part 3, Mission: Random story maker

For the places, add `places = ["in the bath", "on the bus"]` before the loop, `place = random.choice(places)` inside it, and change the print to `print(hero, action, thing, place)`. After the loop, **`len`**`(things)` is 4, and `"my homework"` is the last item, because `append` always adds to the end:

**You should see**

```
4
['a banana', 'the moon', 'a sock', 'my homework']
```

### Part 3: Translate it yourself

```
pets = []
pets.append("cat")
pets.append("dog")
pets.append("fish")
print(len(pets))
print(pets[1])
```

It prints 3 and then dog. Scratch's item 2 is Python's `[1]`. `pets = []` does the job of **delete all of pets**, because it starts a brand-new empty list.

### Part 3: Two-line puzzle

```
for word in words:
    count += 1
```

The loop goes round once for every item, and each time it adds 1 to `count`. That is exactly how **`len`** counts!

### Part 4: Predict it!

| Code | Shows | Why |
|---|---|---|
| `print(9 / 3)` | `3.0` | Divide always gives a decimal. |
| `print("4" + "4")` | `44` | They are strings, so `+` joins them. |
| `print("Dog" == "dog")` | `False` | Capital letters count in Python. |
| `print("pizza"[1])` | `i` | Python counts from 0, so `[1]` is the second letter. |
| `print(round(4.5))` | `4` | A .5 goes to the nearest even number. |
| `print(17 % 5)` | `2` | $17 = 3 \times 5 + 2$, so 2 is left over. |

### Part 4, Mission 1: Dice roller

For 100 rolls, change line 3 to `for i in range(100):` and line 8 to `str(total / 100)`. With two dice, the biggest total is $6 + 6 = 12$.

### Part 4, Mission 2: Times-table printer

For the 9 times table, change line 1 to `table = 9`. To go up to 20, change line 3 to `range(20)`. For square numbers, line 4 becomes `print(f"{n} x {n} = {n * n}")`. The `True` or `False` at the end tells you whether the answer is even.

### Part 4, Mission 3: Random dot art

With 7 colours, line 15 needs `random.randint(0, 6)`. For the top half, line 12 becomes `y = random.randint(0, 180)`.

### Part 4, Mission 4: Name machine

It says `i + 1` because `i` starts at 0, but people count letters from 1.

### Part 4: Translate it yourself

Your code may look a little different and still be right. If it prints three totals, you did it!

```
import random
for i in range(3):
    a = random.randint(1, 10)
    b = random.randint(1, 10)
    print("Total: " + str(a + b))
```

One real run printed `Total: 9`, `Total: 19` and `Total: 8`. An f-string works too: `print(f"Total: {a + b}")`.

### Part 4: Two-line puzzle

```
word = "Turtle"
print(word[0] + str(len(word)))
```

`word[0]` is the first letter, `T`. **`len`**`(word)` is 6, and **`str`**`( )` turns it into a string so it can be joined.

### Part 5, Mission "Say hello": challenges

The age question. **`str`**`( )` turns the number back into writing so it can be joined:

```
name = input("What's your name? ")
age = int(input("How old are you? "))
print("In 10 years you will be " + str(age + 10))
```

**Super challenge:** without **`int`**`( )`, `age + 10` tries to add a number to writing. The last line says `TypeError: can only concatenate str (not "int") to str`. A `TypeError` means writing and numbers got mixed up.

### Part 5, Mission "Guess my number": only 5 tries

```
import random
secret = random.randint(1, 10)
guess = 0
tries = 0
while guess != secret and tries < 5:
    guess = int(input("Guess 1 to 10: "))
    tries += 1
    if guess < secret:
        print("Too low")
    elif guess > secret:
        print("Too high")
if guess == secret:
    print("You got it in " + str(tries) + " tries!")
else:
    print("Out of tries! It was " + str(secret))
```

Typing `cat` gives `ValueError: invalid literal for int() with base 10: 'cat'`, because `int( )` can't turn the word `cat` into a number.

### Part 5, Mission "Quiz with a score": the final message

Put this at the very end, after the score is printed:

```
if score == 3:
    print("Quiz champion!")
elif score >= 1:
    print("Well done!")
else:
    print("Better luck next time!")
```

**Super challenge:** put `import time` and `start = time.time()` at the top, and `print(round(time.time() - start))` at the bottom.

### Part 5, Mission "Reaction timer": best of three

```
import time
import random
best = 100
for i in range(3):
    input("Press Enter to start ")
    print("Get ready...")
    time.sleep(random.randint(2, 5))
    start = time.time()
    input("GO! Press Enter! ")
    taken = time.time() - start
    if taken < best:
        best = taken
print("Your best: " + str(round(best, 2)) + " seconds")
```

`best` starts at 100 seconds, so your first go is sure to beat it.

### Part 5: Translate it yourself

Your code may look a little different and still be right. If it does the same thing, you did it!

```
answer = input("What's the password? ")
while answer != "banana":
    print("Wrong! Try again.")
    answer = input("What's the password? ")
print("Come in!")
```

Using `while not answer == "banana":` on line 2 is right too.

### Part 5: Two-line puzzle

```
age = int(input("How old are you? "))
print("Next year you will be " + str(age + 1))
```

### Part 6, Mission: Make your own block

**Triangle:**

```
def triangle(size):
    for i in range(3):
        forward(size)
        right(120)
```

**Colour input:**

```
def square(size, colour):
    pencolor(colour)
    for i in range(4):
        forward(size)
        right(90)
```

**No brackets:** a line with just `square` does nothing, and there is no error. Without the brackets you only say the block's name. You must call it, for example `square(60)`.

### Part 6, Mission: Shape machine

`flower(8, 4, 60)` draws 8 petals, and each petal is a square. With `animationOff()` the turtle jumps instead of walking, so the drawing appears faster. Colour input:

```
def flower(petals, sides, size, colour):
    pencolor(colour)
    for i in range(petals):
        polygon(sides, size)
        right(360 / petals)
```

Call it with four inputs, for example `flower(12, 6, 40, "red")`.

### Part 6, Mission: Answer machine

```
def half(n):
    return n / 2

def perimeter(width, height):
    return 2 * width + 2 * height

def is_even(n):
    return n % 2 == 0
```

`print(half(9))` shows `4.5`. `is_even(7)` is **`False`**, so the **`if`** part is skipped.

With `print(n * 2)` in line 2, `double` only **shows** the number and gives back `None`. So line 7 shows 8 and then `None`. Line 11 then tries `area(None, 10)`, and Python stops with a `TypeError`, because it can't multiply nothing by 10.

### Part 6, Mission: Sprite army

```
def turn_all(angle):
    for s in army:
        s.left(angle)
```

Call `turn_all(90)` before marching, and the army marches up. For the race, put **`import`** `random` at the top and use `s.forward(random.randint(10, 80))` in `march`. `army`

`[0]` is the first sprite made: the red one at the top (lists count from 0).

### Part 6, Translate it yourself

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))

def star(size):
    for i in range(5):
        forward(size)
        right(144)

pencolor("gold")
star(100)
star(50)
```

You should see a big gold star with a small one inside it, sharing one point.

### Part 6, Two-line puzzle

```
def triple(n):
    return n * 3
```

`triple(2)` is 6, and `triple(6)` is 18.

### Part 7, Costume parade: five costumes

If you add `"arrow"` to the costume list, the colour list needs a fifth colour too, for example `colours = ["green", "red", "blue", "orange", "purple"]`. Otherwise, when `n` is 4, `colours[4]` doesn't exist and Python stops with an `IndexError`. Line 17 doesn't need changing: `len(costumes)` is now 5 by itself.

### Part 7, Make music: Happy Birthday

```
G = 392
A = 440
B = 494
C2 = 523
tune = [G, G, A, G, C2, B]
beats = [0.75, 0.25, 1, 1, 1, 2]
```

Put these in place of lines 2 to 10 and keep the loop as it is. If it sounds too fast, change `0.4` in line 12 to `0.5`.

### Part 7, Turtle race: three racers

```
green = make_racer("green", 0)
```

goes after line 17. The loop and its inside become:

```
while (red.xcor() < 200 and blue.xcor() < 200
       and green.xcor() < 200):
    red.forward(random.randint(1, 20))
    blue.forward(random.randint(1, 20))
    green.forward(random.randint(1, 20))
```

The round brackets let the condition go over two lines.

### Part 7, Translate it yourself

```
from ColabTurtlePlus.Turtle import *
clearscreen()
initializeTurtle((480, 360))
import time

shape("turtle")
shapesize(3)
for i in range(3):
    hideturtle()
    time.sleep(0.5)
    showturtle()
    time.sleep(0.5)
print("Boo!")
time.sleep(2)
```

Size 300 % is `shapesize(3)`. "Say for 2 seconds" is a `print` and then a 2-second wait. If you used a speech turtle to write `Boo!` on the stage and `clear()` it after 2 seconds, that is even closer to Scratch.

### Part 7, Two-line puzzle

```
def cheer():
    print("Hooray!")
```

The third line, the broadcast, is `cheer()`. When you run all three, you see `Hooray!`.

## Big Projects: hints for "Level up"

There is no single right answer for a level up. If your program does what you wanted, it is right! These hints help if you get stuck.

**Animate your name.** Change `range(8)` to `range(20)` on line 20 and the number in `time.sleep()` on line 29. For growing letters, change the font on line 27 to `font=("Arial", 20 + frame * 5, "bold")`. For the rainbow, replace `random.choice(colours)` on line 24 with `colours[frame % 7]` (there are 7 colours in the list).

**Pen flower.** `petal(150)` makes bigger petals that may go off the stage. For a flower inside

a flower, copy lines 18–21 and use `petal(50)` in the copy. For the garden, this works (it replaces lines 17–31):

```
def flower(x, y):
    penup()
    goto(x, y)
    pendown()
    for p in range(petals):
        color(colours[p % 6])
        petal(50)
        left(360 / petals)

petals = int(input("How many petals? (3 to 20) "))
flower(-150, 0)
flower(0, 0)
flower(150, 0)
hideturtle()
```

**Maths quiz.** For 10 questions change `range(5)` on line 16 and the 5 in the messages (lines 22 and 24). For a seven-times-table quiz, line 17 becomes `a = 7`. For the sounds: `beep(523)` goes after `print("Correct!")` and `beep(196)` after the Oops line.

**Guess the number deluxe.** Halving 1000: 500, 250, 125, 63, 32, 16, 8, 4, 2, 1. That is 10 tries. Add `elif level == "4":` with `top = 1000` and `tries = 10` before the `else:`. To count wins, set `wins = 0` before the big loop, add `wins = wins + 1` inside `if won:`, and print it after the loop. For "Hot!": `if guess != secret and guess - secret < 6 and secret - guess < 6:`. For "too big": check `if guess > top:` straight after the `input` line, print a message, and add 1 back to `tries`.

**Choose-your-own story.** A third choice needs a different helper, or you can ask two questions in a row: first `choose("Row or swim?", ...)`, and inside the swim part, `choose("Swim or cross the bridge?", "swim", "bridge")`, which calls `bridge()`. Put `def bridge():` near the top with the other scenes, so Python has read it before the story starts.

**Virtual pet.** Add `energy = 7` at the top, a new `elif choice == "sleep":` that sets `energy = 10`, and `energy = energy - 2` in the play part. A mood message: `if happy >= 8:` then `print("(purring)")`. For winning: `if day == 15:` print your certificate, then `break`.

**Chase game.** Put `cat_step = 25` next to `score = 0`, use `cat.forward(cat_step)`, and add `cat_step = cat_step + 5` inside the star's `if`. A fast cat can jump right over you, so also change the catch test to `cat.distance(pos()) < cat_step + 5`.

A second cat: copy lines 11–16 with the name `cat2` and `cat2.goto(-200, -140)`, then copy lines 49–53 for `cat2`.

**Catch game.** For a wider start, line 23 becomes `x = random.randint(-160, 160)`; to keep the apple on the stage, use `drift = random.choice([-10, 0, 10])`. For the computer player, replace line 30 with

```
    spot = apple.xcor() + 4 * drift
```

The apple has 4 more steps to fall, and it drifts by `drift` on each step. The computer catches every apple! For the stamps, put `apple.stamp()` after line 35.

## The Bug Zoo: bug hunt

Every error below is the real last line from Python 3.13. Your fix may look a little different and still be right: if the program now does what it should, you caught the bug!

| Bug | The clue | The fix |
|---|---|---|
| 1 | `IndentationError: expected an indented block after 'while' statement on line 2` (the Wobbly Stack) | Line 3 needs four spaces in front, so both lines 3 and 4 are inside the loop. |
| 2 | `NameError: name 'Print' is not defined` (the Capital Copycat) | Use a small p: `print("Your score is", score)` |
| 3 | `TypeError: can only concatenate str (not "int") to str` (the Glue Goblin) | `input()` always gives text. Turn it into a number: `age = int(input("How old are you? "))` |
| 4 | `SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?` (the Equals Eel) | To ask a question, use two equals: `if password == "cat123":` |
| 5 | No error on line 2, but it prints `First: banana` (the Zero Hero). Then `IndexError: list index out of range` (the Off-by-One Owl). | Python counts from 0: use `fruits[0]` for the first and `fruits[2]` (or `fruits[-1]`) for the last. |

| | | |
|---|---|---|
| 6 | `ZeroDivisionError: division by zero` (the Zero Zombie) | Check first:<br>`if friends == 0:`<br>`  print("Nobody to share with!")`<br>`else:`<br>`  print("Each gets", sweets / friends)` |
| 7 | No error, but nothing is printed (the Silent Recipe). | Add the brackets to cook the recipe: `cheer()` |
| 8 | No error, but it prints 5 forever (the Forever Phantom). Press the stop button; you then see `KeyboardInterrupt`. | `n` never changes. Add `n = n - 1` inside the loop, after `print(n)`. |
| 9 | No error, but the turtle draws one long line of 200 steps instead of a square (the Lost Hooray's cousin). | `right(90)` has no spaces, so it is outside the loop and happens only once. Put four spaces in front of it. |
| 10 | `SyntaxError: unterminated string literal (detected at line 3)` (the Quote Thief) | Add the closing quote: `print("Nice to meet you!")`. Because it is a grammar bug, nothing runs, not even the question on line 1. |