

GRAPHIC GUIDE to Programming



GRAPHIC GUIDE TO pythonTM

with Processing.py 3

ANTONY LEES

TABLE OF CONTENTS

[Graphic Guide to Python with Processing.py 3 \(Graphic Guide to Programming\)](#)

[Chapter 1: Introduction to Python and Processing](#)

[Chapter 2: Programming Basics](#)

[Chapter 3: If Statements](#)

[Chapter 4: Loops](#)

[Chapter 5: Data Structures](#)

[Chapter 6: Functions](#)

[Chapter 7: Animation](#)

[Chapter 8: Colour](#)

[Chapter 9: Lines and Shapes](#)

[Chapter 10: Custom Shapes](#)

[Chapter 11: Object-Oriented Programming](#)

[Chapter 12: Transformation](#)

[Chapter 13: Image Manipulation](#)

[Chapter 14: Event Handling](#)

[Chapter 15: 3D Rendering](#)

[Chapter 16: Creating a Game](#)

[Epilogue](#)

[Appendix A: Pirate Animation Code](#)

[Appendix B: Mouse Controlled Space Game Code](#)

[Appendix C: Cat Game Code](#)

Graphic Guide to Python

with Processing.py 3

Written by Antony Lees

Cover Design by Louise Gillard

Published by Antony Lees

© 2024

All rights reserved. No part of this book may be reproduced or modified in any form, including photocopying, recording, or by any information storage and retrieval system, without permission in writing from the publisher.

First printed 2024

www.devoniant.com

“Python” and the Python Logo are trademarks of the Python Software Foundation



Acknowledgements

Book supervision provided by:

Miss Pandora Sparklepaws ('Pandy') – Assistant Supervisor (left)

HRH Sherbet Sparklepaws ('Sparkles'), Princess of the Floofs – Senior Supervisor (middle)

Miss Mittensia Sparklepaws ('Mittens') – Apprentice Supervisor (right)

PREFACE

Processing is a free, open-source, programming language based on the much larger Python language. It focusses on the visual output of computer programs, designed for non-programmers, and those more experienced, to create visual graphics and animation. Processing includes everything needed to create and view graphics including an IDE (Integrated Development Environment) that can be used to create programs and graphics.

As Python is not a visual language, it is often taught using text output like printing “Hello world!”. By using Processing, we can learn Python with interesting graphics, giving immediate visual responses to the code you write, which is much more exciting!

The book aims to teach the Python programming language to both non-programmers and experienced programmers alike, allowing those who have not worked on computer programs before to become competent in the language and anyone to create some visually stunning graphics and animation using Processing, regardless of prior experience.

You will soon discover that there are no real rules to creating graphics in Processing and that minor changes can yield unexpected results. So, my best advice is, try stuff out, see what happens, have fun!

THE BIG CAVEAT

At the time of writing, Python in Processing can only be used in version 3 of Processing, **not** 4. This book was written using version 3.5.4.

Acknowledgements

Preface

The Big Caveat

Section 1: Beginning python with processing

Chapter 1: Introduction to Python and Processing

Python vs Processing

The Processing Environment

Your First Programme

Saving Your Code

Breaking Down Your Code's Language

Catching Typos: The Sneaky Culprits

Screen Coordinates

Drawing Some Simple Shapes

Chapter 2: Programming Basics

Variables

Declaring Variables

Using Variable Values

Using the Console

Variable Naming Rules

[Variable Types](#)

[Operators](#)

[The Assignment Operator](#)

[Mathematical Operators](#)

[Shorthand Operations](#)

[Relational Operators](#)

[Precedence and Brackets: Keeping Order](#)

[Comments](#)

[Some Operator Code](#)

[Chapter 3: If Statements](#)

[The if Statement](#)

[The if-else Statement](#)

[Multiple if Statements](#)

[Logical Operators](#)

[Multiple Branches: if Statements](#)

[Nesting if Statements](#)

[Chapter 4: Loops](#)

[For Loops](#)

[Specifying a starting range value](#)

[Specifying the increment](#)

[Or Else.. What?](#)

[While Loops](#)

[Break](#)

[Continue](#)

[Nested Loops](#)

Chapter 5: Data Structures

[Behold, the List!](#)

[Conquering Lists with Loops](#)

[Dynamic Lists](#)

[Lists with different types of elements](#)

[Checking list contents](#)

[Multi-Dimensional Lists](#)

[Tuples](#)

[Sets](#)

[Dictionaries](#)

Chapter 6: Functions

[Built-in Functions](#)

[Writing your own functions](#)

[Default arguments](#)

[Return Values](#)

[Multiple Return Statements](#)

[Global and Local Variables](#)

[Global Variables](#)

Chapter 7: Animation

[Static Mode](#)

[Active Mode](#)

[setup\(\)](#)

[draw\(\)](#)

[Active Mode Controls](#)

[loop\(\) and noLoop\(\)](#)

[Frame Rate](#)

[Animation](#)

Chapter 8: Colour

[Processing and Colour](#)

[Background Colour](#)

[Fill Colour](#)

[Colour Modes](#)

[HSB](#)

[Hue](#)

[Saturation](#)

[Brightness](#)

[Colour Functions](#)

[Blending Colour](#)

[ADD](#)

[SUBTRACT](#)

[MULTIPLY](#)

[Linear Interpolation](#)

[Chapter 9: Lines and Shapes](#)

[Lines](#)

[The line\(\) Function](#)

[Line Thickness](#)

[Line Shape](#)

[Shapes](#)

[Rectangles](#)

[Placing Rectangles](#)

[Squircles](#)

[Quadrilaterals](#)

[Ellipses](#)

[Arcs](#)

[Modes](#)

[Triangles](#)

[Shapes Animation](#)

[Chapter 10: Custom Shapes](#)

[Vertices](#)

[Simple Custom Shapes](#)

[Custom Regular Shapes](#)

[Complex Shapes](#)

Section 2: Advanced concepts

Chapter 11: Object-Oriented Programming

[Classes](#)

[Methods](#)

[Using Objects](#)

[Inheritance](#)

[Inherited Methods](#)

[Extending a Subclass](#)

[Lists of Objects](#)

[A Longer Example](#)

[Pirate Ship](#)

[The Hull](#)

[Sails](#)

[Jolly Roger](#)

[Portholes](#)

[Movement](#)

[A Fleet of Pirate Ships](#)

[Waves](#)

Chapter 12: Transformation

[Translating](#)

[Scaling](#)

[Rotation](#)

[The Transformation Matrix](#)

[Resetting the Transformation Matrix](#)

[Remembering the Transformation Matrix](#)

Chapter 13: Image Manipulation

[Graphics Formats](#)

[Manipulating Pixels](#)

[Using Images](#)

[Exception handling](#)

[Images as Pixels](#)

[Image Manipulation Functions](#)

[Masking](#)

[Tinting](#)

[Filters](#)

[Blending](#)

[Blend Modes](#)

[Darken modes](#)

[Lighten Modes](#)

[Contrast Modes](#)

[Comparative Modes](#)

[Chapter 14: Event Handling](#)

[Keyboard Events](#)

[Keyboard Event Actions](#)

[keyPressed](#)

[keyPressed\(\)](#)

[keyTyped\(\)](#)

[keyReleased\(\)](#)

[Key Event Variables](#)

[key](#)

[keyCode](#)

[Mouse Events](#)

[Mouse Event Actions](#)

[mousePressed](#)

[mousePressed\(\)](#)

[mouseReleased\(\)](#)

[mouseClicked\(\)](#)

[mouseWheel\(\)](#)

[Mouse Event Variables](#)

[mouseButton](#)

[mouseX and mouseY](#)

[Mouse Movement Functions](#)

[mouseMoved\(\)](#)

[mouseDragged\(\)](#)

[A Mouse-controlled Game](#)

[Chapter 15: 3D Rendering](#)

[Rendering Modes](#)

[3D Mode](#)

[3D Space: The z-coordinate](#)

[Predefined 3D Shapes](#)

[Box](#)

[Sphere](#)

[Custom 3D Shapes](#)

[Lights](#)

[Directional Light](#)

[Ambient Light](#)

[Chapter 16: Creating a Game](#)

[Drawing a Cat](#)

[Cat Body](#)

[Cat Head](#)

[Cat Ears](#)

[Cat Tail](#)

[Cat Movement](#)

[The Game Board](#)

[Tile Types](#)

[Combining Cat and Board](#)

[Game Play](#)

[Score Board](#)

[Life Counter](#)

[Game Over!](#)

[3D Game Board](#)

[Enhancing Game Play](#)

[Scrolling Game Board](#)

[Grace Period](#)

[Game Speed](#)

[Over To You](#)

[Epilogue](#)

[Appendices](#)

[Appendix A: Pirate Animation Code](#)

[Appendix B: Mouse Controlled Space Game Code](#)

[Appendix C: Cat Game Code](#)

SECTION 1: BEGINNING PYTHON WITH PROCESSING

CHAPTER 1: INTRODUCTION TO PYTHON AND PROCESSING

Python vs Processing

In this book you will be learning both Python and Processing. Python is a general-use programming language that has multiple applications and is widely used in industry, especially for data analytics. Processing is a graphics programming language built on top of Python, allowing you to write Python code that creates visually interesting graphics, hopefully making your Python journey a lot more interesting than printing out text on a console. Python can be used without Processing, and Processing can be used in languages other than Python.

Processing makes a great learning tool for learning Python because you can actually see the results of your code. Plus, it's fun. Now that we have that clear, let's look at the Processing environment we will use to write our code and view the results.

The Processing Environment

Processing empowers graphic artists, even absolute coding novices, to craft stunning computational art with breathtaking ease. Don't be fooled by its simple interface—it unlocks immense creative potential. You can conjure complex visuals without months of programming knowledge.

Let's delve into the key elements of Processing to guide you before you truly dive in. This environment, referred to as an IDE (Integrated Development Environment), is called the PDE (Processing Development Environment) in Processing-speak. In Processing, your artistic creations are

called "sketches" which are visual computer programmes. The code that brings them to life is known as "sketch code" but we will just call this code.

Ready to create amazing computational art?

- Head over to processing.org and download the Processing software for your system (Windows, Mac, Linux). *Make sure you use version 3.5.3 not version 4.* You may have to click a link to get older versions
- Unzip the downloaded file
- Double-click the Processing application to embark on your artistic journey

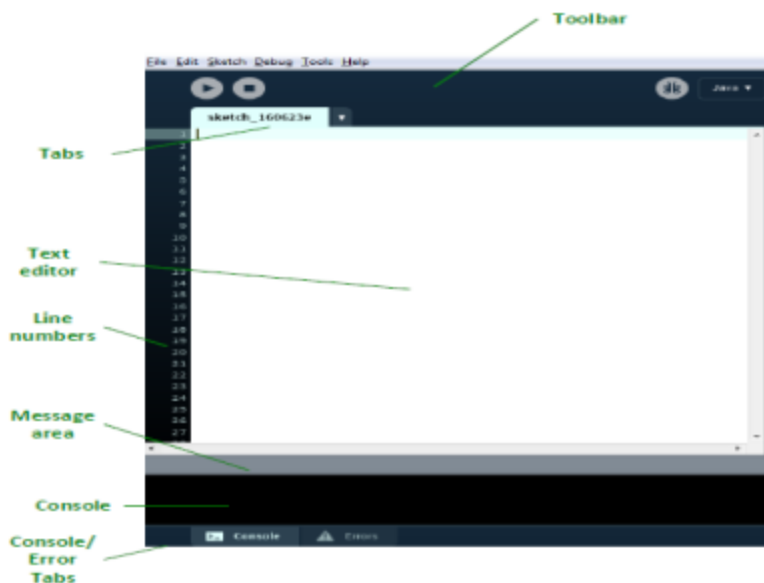


Figure 1: The Processing Environment

Upon opening, you'll be greeted by a welcoming interface which showcases the Processing interface, composed of six key components:

- **Toolbar:** Your handy companion for managing and running your code.

- **Tabs:** Each tab acts as a canvas for your unique code creations.
- **Text Editor:** Craft and view your code within this central workspace.
- **Line Numbers:** Keep track of your code with numbered lines, with the current line highlighted for easy reference.
- **Message Area:** Processing keeps you informed here, displaying useful messages.
- **Console/Error Tabs:** Switch between the Console for general output and the Errors tab for troubleshooting.

Understanding the Toolbar

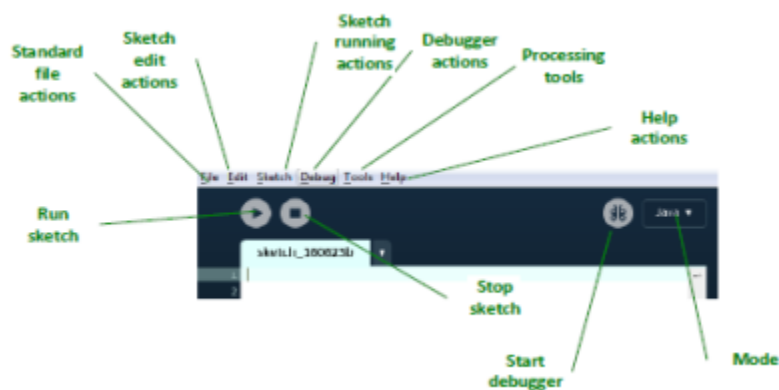


Figure 2: Processing Environment Toolbar

While the toolbar's functions might seem intuitive, let's take a closer look at each button for clarity:

- **Run:** Press this to bring your code to life! It compiles and runs your code.
- **Stop:** Need to hit the brakes? This button halts a running programme.
- **Debug:** If something goes wrong, this button opens the debugger for troubleshooting.

- **Mode:** Changes the language you use for writing your programme. The default is Java, but we'll adjust this soon.

Navigating the Top Menu

The menu bar houses helpful actions, but don't worry, we won't delve into every option. If you need full details, the Processing website has your back!

Here's a quick overview:

- **File:** Manage your programmes, from opening and saving to printing and setting preferences.
- **Edit:** Need to cut, copy, or paste code? This is your go-to section.
- **Sketch:** Control your programme's (sketch's) life cycle, including running and importing libraries.
- **Debug:** Debugging tools become your allies when encountering code issues.
- **Tools:** For advanced users, this section offers less frequently used utilities.
- **Help:** Stuck? Get guidance and explore resources right here.

Note: The menu layout might vary slightly depending on your operating system (the example shown is from Windows 10). But fret not, it's likely similar to what you're used to seeing in other programs.

Your First Programme

Before we can start writing our first programme, we need to set Processing to use Python mode. First, click the 'Mode' button and click 'Add Mode'. This will display the Contribution Manager.

Choose the 'Modes' tab and highlight Python Mode for Processing 3 by clicking it. If it has a tick next to it, you are good to go. If not, click the Install button at the bottom and wait for it to install. Once installed you can close the Contribution Manager window.

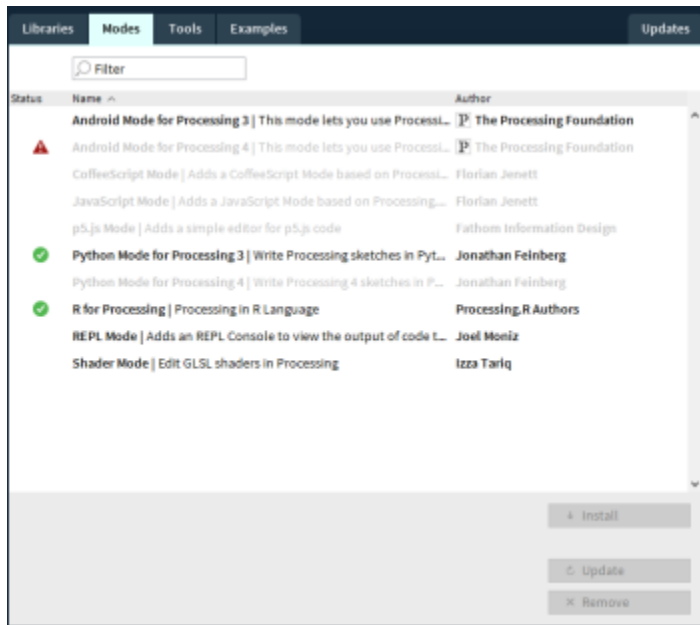


Figure 3: Contribution manager

Click the 'mode' button again and choose 'Python'. Processing should re-open with Python mode showing on the mode button.



Figure 4: Processing showing Python mode

Let's create our first running programme! Get started by opening Processing and focussing on the white text editor. In that space, type this magical line:

```
circle(20, 20, 20)
```

Remember, every character matters, so copy it exactly! Python cares about spaces, so this must be written on the very left. Now, hit the run button (it looks like a play symbol). If you typed it correctly, something amazing happens!

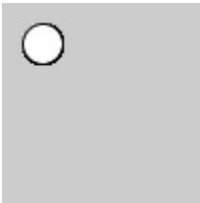


Figure 5: Hello world

A new window pops up, displaying a little circle. We'll call this our 'hello world' circle as the first program people often write is to print out some text that says 'hello world'. Close this window - it's called the display window, where your graphic creations will come to life.

Congratulations! You've just created your first programme, a moment cherished by programmers worldwide as the "hello world" milestone. Seeing this confirms everything is working perfectly!

Saving Your Code

Let's make sure your first masterpiece doesn't vanish! Click on "File" and then "Save" to give it a name, like "hello_world." Processing automatically adds a ".pde" extension, like a signature that says, "This is a special Processing file!"

Breaking Down Your Code's Language

Ready for a quick code anatomy lesson? That line you typed is like a sentence Processing understands:

- **circle:** This is a command, telling Processing to display something on the screen.
- **(and):** These are parentheses, acting like a cozy hug to hold the information Processing needs to do its job.
- **20, 20, 20:** These are called arguments, which describe where to draw the circle.

Don't worry if you make a mistake—Processing is like a patient teacher, offering helpful error messages to guide you back on track. For example, if you forget a bracket or quote mark, it'll kindly point out what's missing.

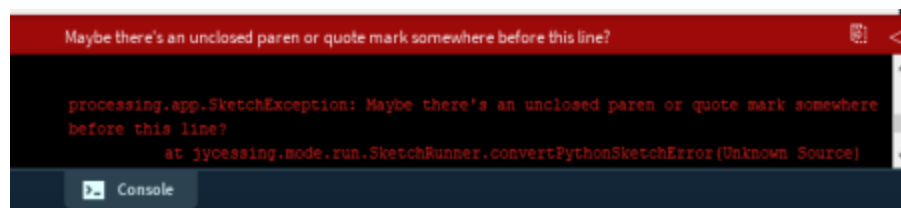


Figure 6: Error console

Catching Typos: The Sneaky Culprits

Remember our "hello world" code? Even with just one line, mistakes happen! While Processing often throws helpful error messages, some typos can be trickier to spot.

For example, try misspelling "circle" as "circl" in your code. You won't see an error message, but... nothing will happen either! This requires you to detective-mode and carefully review the line to find the culprit.

Don't worry, even experienced coders face this! The important thing is to practice reading your code with eagle eyes and learning from each small

slip-up. Soon, you'll be like Processing itself, catching those sneaky typos in no time!

Screen Coordinates

You have seen that, when you press run, you see a display window. This is the screen that will be used for displaying graphics. Of course it isn't very exciting at the moment because we haven't told it to display much. Unless we change it, the display will be a default size, that is the size that Processing uses unless told otherwise. The default is 100 wide and 100 high. But 100 what?

Computer displays are made up of dots, called pixels, which you will have experienced, even if you didn't know it, when looking at TV and screen display sizes or with camera resolutions.

You can control the size of the display window using `size()` which defines the number of pixels along the x- and y- axes, in that order. So a size of 600, 400 will create a window 600 pixels wide and 400 pixels high. This would give a resolution of 600 x 400, or 240,000, pixels.

The displayable area is like a grid, with the number of pixels measured from the top-left point, position (0, 0) and increasing along each axis. The coordinates of any single pixel can be determined by providing the x- and y- coordinates in relation to the starting point at position (0, 0). As examples, the bottom right of a 600 x 400 display is at position (600, 400) and the middle is at position (300, 200).

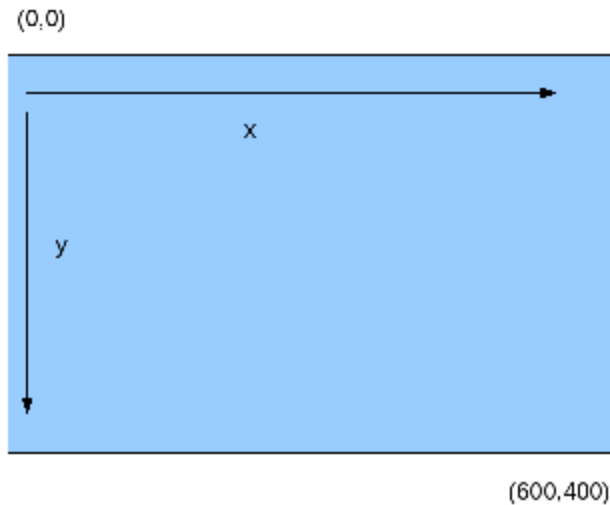


Figure 7: Coordinates of the Processing screen

Try to change the size of the display by putting:

```
size (600, 400)
```

in your code and pressing run. You should see that the size changes.

We can use these coordinates to draw at specific points on the screen. So, for example, we can draw a dot (called a point) by using point function in the form `point(x, y)`. For example, the code:

```
size(600, 400)
```

```
point(300, 200)
```

will draw a point in the middle of the screen.

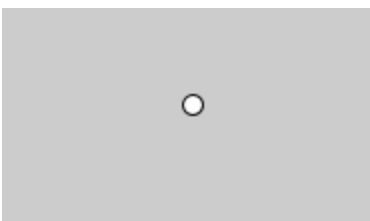


Figure 8: Circle in the centre of the screen

Of course it is very difficult to see! Let's draw a circle instead

```
size(600, 400) # screen size 600 x 400
```

```
circle(300, 200, 10) # circle
```

This draws a circle at the centre coordinates; x-coordinate 300, y-coordinate 200, because the screen is 600 wide by 400 high. The circle is 10 pixels wide because we specified it. So, the line of code:

```
circle(300, 200, 10) # circle
```

means

□ Draw a circle

- At x-coordinate 300
- At y-coordinate 200
- 10 pixels wide
- '# circle' is what we call a comment and is just for information

Any set of coordinates can be used to draw a shape. Provided the coordinates are within the range of those of the screen, then the shape will be visible. If coordinates are used that are not within that range, for example a negative number or a number greater than the greatest x or y position, the shape will not be visible.

Try changing the size of the screen by changing the numbers in `size(600, 400)` to something else and press run. What happens? The circle is no longer in the centre, right?

We could, of course, just work out where the centre now is and change the circle as well, but that's annoying, and it would be really tedious if we had lots of shapes to change! Luckily there is a solution to this.

Processing's built-in variables `height` and `width` can be used find out what the limit of the coordinates are. Therefore, the point:

```
point(width, height)
```

is at the bottom-right of the screen so we can work out coordinates in relation to the width and height. I will cover how to do this in the next chapter.

```
point(0, 0)
```

will always be the top-left though.

Drawing Some Simple Shapes

Let's create something more visually exciting! Copy and paste this code into Processing's text window, and then hit the run button with fearless curiosity:

```
size(640,580)
```

```
rect(0, 0, width/3, height/3)
```

```
ellipse(width/2, height/2, width/3, height/3)
```

```
triangle(width, 0, width, height, width-width/3, height)
```

If Processing complains, don't fret! It's common to make tiny typos. Double-check the code for accuracy, and try again.

Here's what you created:

- **size(640, 580)** This is the size of the display window that you can use for drawing on
- **rect(), ellipse(), triangle()** These draw shapes you observed—a rectangle, circle, and triangle, each taking their designated positions on the canvas.

Now, unleash your curiosity! Try altering the numerical values within the code and witness how the shapes transform. Fear not—if a change displeases you, simply change it back and run it again. Embrace the joy of experimentation!

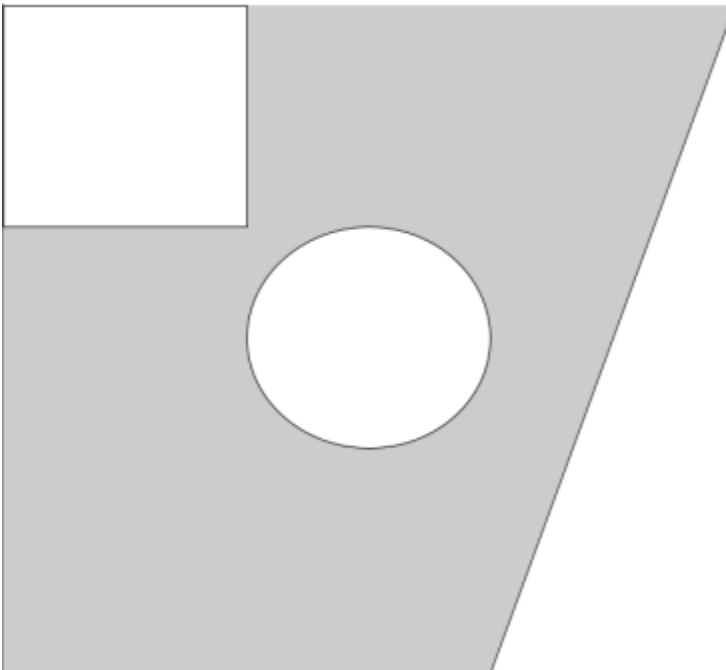


Figure 9: Some shapes

CHAPTER 2: PROGRAMMING BASICS

Variables

Imagine your computer program as a room full of whiteboards. Each variable is like an individual whiteboard where crucial information is stored. When we want to find that information, we find the relevant whiteboard and the information is there. Their contents can change throughout the program, just like rubbing it out and writing a new piece of information.

Declaring Variables

Before using a whiteboard (variable), you need to tell Python about it. This "declaration" usually happens at the start of your program. Here's how:

```
wheel_size = 20
```

This creates a variable named "wheel_size" and assigns the value "20" (the size of a wheel in our example). This first assignment is called "initialisation."

Using Variable Values

Want to use what's on the whiteboard? You can use it later in your code like this:

```
wheel_size = 20
```

```
circle(20, 40, wheel_size)
```

This will draw a circle at x coordinate 20, y coordinate 40 and the circle will be the size we have stored in wheel_size (so, 20)

Run this code and you will see a circle, representing a wheel, on the screen.

If you change the value of `wheel_size` and run it again, it will change the size of the circle. For example change it so the code says

```
wheel_size = 50
```

When you run it, you'll get a bigger wheel!

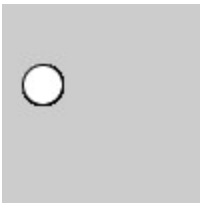


Figure 10: Using a variable to draw a circle

Using the Console

If you ever need to check what the value of a variable is, or help you find out what is happening in your code, you can use the console. The console is the black area below where you write your code

```
wheel_size = 50
```

```
circle(20, 40, wheel_size)
```

```
print(wheel_size)
```

Run this code, and you'll see the number "20" appear in the Processing window, showing the value stored in your `wheel_size` variable.

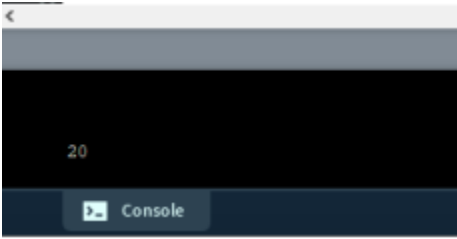


Figure 11: The Processing console

Variable Naming Rules

- Python takes names seriously! It distinguishes between uppercase and lowercase letters (so "Name" and "name" are different variables)
- Python programmers use "snake_case" for names: all lowercase with underscores between each word (e.g., "number_of_wheels"). It's called snake case because the shape looks a bit like a snake. You may also see examples of camelCase (because it has humps)
- Variable names can only start with letters or underscores, and can only contain letters, numbers, and underscores
- Avoid using reserved keywords like "print" as variable names
- While technically possible, we highly discourage having similar-looking names like "my_Name" and "my_name" - it's a recipe for confusion!

Processing lets you choose descriptive names for your variables. Don't settle for generic names like "a" or "size"! Choose clear and meaningful names like "wheel_size" to make your code easier to understand later. Think of it like labelling your files for better organisation!

Variables are your key to storing and managing data in your Processing programs. Use them wisely and creatively to unlock your coding potential!

Variable Types

Unlike some other programming languages, Python does not specify the ‘type’ to variables as such. For example, we did not need to declare that the `number_of_wheels` variable would contain a number.

However, Python does *know* that it contains a number. In this case a specific type of number called an *integer* which means a whole number with no decimal points. There are other data types too.

Type	Description	Examples
Integer	Whole number	10
		2.6
Floating point number	Number with decimal places	10.0
Boolean	Can only be true or false	True
		False
String	Text	“Hello”
		‘Hello’

Table 1: Variable types

Operators

Just like a toolbox for different tasks, operators fall into categories based on their functions. We'll explore each category along with its unique tools (operators).

The Assignment Operator

You've already met the "=" operator. This operator assigns values or results to variables, like saying this whiteboard (called `wheel_size`) has the number 20 written on it.

```
wheel_size = 20
```

Here, "=" stores the number 20 in the `wheel_size` variable. Whenever you use this variable, you'll retrieve the value 20 unless you update it.

You can update variables repeatedly to change their value:

```
wheel_size = 20
```

```
circle(20, 40, wheel_size)
```

```
wheel_size = 30 # get a bigger wheel
```

```
circle(40, 40, wheel_size)
```

Notice how we assign values multiple times. This changes the value stored in that variable.

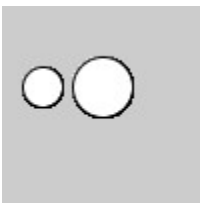


Figure 12: Two circles

You can even copy values between variables:

```
wheel_size = 20
```



```
tyre_size = wheel_size
```

```
circle(20, 40, tyre_size)
```

This will copy the value of `wheel_size`, which is 20, to the variable `tyre_size`.

Each variable holds its own value, independent of others.

Mathematical Operators

The mathematical toolbox contains all the mathematical tools you might need. The operators can be used on plain numbers, such as $10 + 2$, or on variables that contain numbers, such as $x + y$

+ Addition $x + y$

- Subtraction $x - y$

* Multiplication $x * y$

/ Division x / y

% Modulus $x \% y$

** Exponentiation $x ** y$

// Floor division $x // y$

Table 2: Basic mathematical operators

We will describe each one:

- Addition (+) - adds 2 numbers together
- Subtraction (-) - subtracts the second number from the first
- Multiplication (*) - multiplies 2 numbers. Note the symbol for this operator is *
- Division (/) - divides the first number by the second. Be careful not to divide by zero as that will cause an error (dividing by zero results in an infinity answer and Python doesn't know how to deal with infinity!)
- Modulus (%) - divides the first number by the second then provides the remainder. If these are fully divisible (eg 10 / 2) then the result will be 0. Note that % does not mean percentages
- Exponentiation (**) - the first number to the power of the second
- Floor division (//) - the first number divided by the second and then rounded down

Mathematical operators can be used to reassign the value of variables based on the result of the mathematics. For example:

```
x = x + 1
```

This calculates the current value of x, adds 1 to it, then assigns the result back to the variable x.

So in the situation:

```
x = 1
```

```
x = x + 1
```

The variable x will have the value 2 at the end of this code.

Note that this will only work with number types. If you try to add a string it will cause an error.

Shorthand Operations

= `x = 5` `x = 5`

+= `x += 3` `x = x + 3`

-= `x -= 3` `x = x - 3`

*= `x *= 3` `x = x * 3`

/= `x /= 3` `x = x / 3`

%= `x %= 3` `x = x % 3`

//= `x //= 3` `x = x // 3`

**= `x **= 3` `x = x ** 3`

Table 3: Shorthand Operators

These operators streamline code when updating variable values but are simply shorter versions of code. For example: `x += 5` is shorthand for `x = x + 5`.

These operators combine assignment and the corresponding operation (add, subtract, multiply, divide etc).

Relational Operators

`==` Equal `x == y`

`!=` Not equal `x != y`

`>` Greater than `x > y`

`<` Less than `x < y`

`>=` Greater than or equal to `x >= y`

`<=` Less than or equal to `x <= y`

Table 4: Relational Operators

These operators compare values and return true or false depending on the result. Something that can only be true or false is called a Boolean value.

For example: `x < 10` checks if the value in `x` is less than 10. Similarly, `x > y` checks if the value in `x` is greater than the value in `y`.

These can be combined such that `x >= y` checks if the value in `x` is greater than or equal to `y`. So in the example:

```
x = 6
```

```
y = 10
```

```
print(x > y)
```

the result 'False' will be printed.

The Boolean result can also be stored in a variable for later use, such as:

```
x = 6
```

```
y = 10
```

```
result = x > y
```

```
print(result)
```

Precedence and Brackets: Keeping Order

Python follows specific rules when evaluating multiple operators (like BODMAS in mathematics). This allows you to understand, and specify if needed, the order that operators are used, and is the cause of many an internet argument.

For example, what is the result of this:

```
x = 6
```

```
y = 2
```

```
result = x * 2 + y
```

The correct answer is 14. You can check this by printing the ‘result’ variable if you wish. If you thought the answer was 24, it might be because you didn’t apply the order of precedence.

The order is as follows:

1. Brackets
2. Order (powers)
3. Division and Multiplication
4. Addition and Subtraction

Based on this order, the above should calculate $x * 2$ before adding y because multiplication comes before addition. If you want to control the order, you can use brackets:

```
x = 6
```

```
y = 2
```

```
result = x * (2 + y)
```

This will result in 24 because brackets come before multiplication/ you can add as many brackets as you like although it can get confusing!

Comments

Just like adding notes to drawings, use comments to explain your code for future reference. Comments in Python is any text after a `#` until the end of the line. You can put the `#` at the start of a line, and then the entire line is a comment, or after some code.

You should aim to:

- Write meaningful comments that clarify your thoughts and actions.
- Avoid redundant comments that simply repeat the code's meaning.

Some Operator Code

Operators are your essential tools for performing calculations, comparisons, and logic in Processing. Master them, and your code will become more powerful and easier to understand!

As an early example of drawing using what we have learned, and some other concepts we will learn later, we have created some code for you to

look through below

```
# set screen size

size(600, 600)

# Set the background colour

background(220)

# Define variables for size

width = 200

height = 150

# Draw a rectangle in the centre

fill(100, 200, 0) # Green colour

rect((width - width/2), (height - height/2), width, height)

# Draw a circle inside the rectangle

fill(255, 0, 0) # Red colour

ellipse(width/2, height/2, width/3, height/3)

# Draw a triangle below the rectangle

fill(0, 0, 255) # Blue colour

triangle(width/4, height + height/4, 3*width/4, height + height/4, width/2, height + 2*height/4)
```

You should be able to work out what this does from the comments. Can you see some examples of the operators in action? We have used the assignment operator to set initial values of width and height, then used these variables to draw our shapes.

Instead of working out the exact values, we have used operators to work them out based on the width and height variables. What do you think will happen if you change the values of these variables? Try it and find out.

What you should find is that the shapes all resize! It's like magic, right? Just by changing these values, the size of all the shapes changes because we based them on the initial values. This makes our code much more flexible than typing in the exact sizes for the shapes each time.

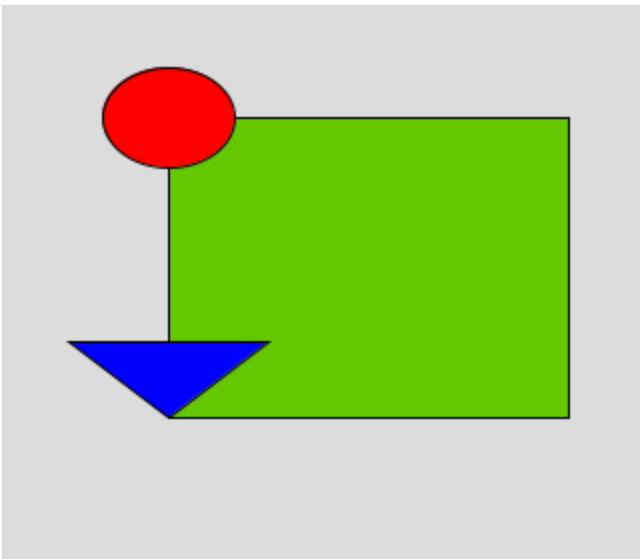


Figure 13: Coloured shapes



Chapter 3: If Statements



The if Statement

Until now, our Processing code has flowed smoothly from top to bottom, like reading a story. But what if you want your programmes to react or change based on certain conditions? That's where conditional statements come in!

Imagine a switch with two positions: True and False. The if statement acts like that switch, controlled by a boolean (true/false) value. We call each route through our code a branch:

- If the condition is True, the indented code below it is run
- If the condition is False, that code is skipped, and continues after the if block.

Here's a basic example:

```
is_day_time = True
```

```
if is_day_time == True:
```

```
    background(0, 0, 255) # blue skies!
```

Here we are setting our `is_day_time` variable to be a True boolean value and then using the if statement to determine if it is day time (True). If it is, we set the background to a blue colour.

Note the indentation for the background statement. This is important as Python uses indentations to decide what it should be doing. In this case the

indent is saying that the background should only be changed if `is_day_time` is true.

The if-else Statement

What if your programme needs different actions depending on whether a condition is true or false? Enter the if-else statement, your two-in-one decision maker!

Let's imagine you want to set the background colour if it is not daytime:

```
is_day_time = True
```

```
if is_day_time == True:
```

```
    background(0, 0, 255) # blue skies!
```

```
else:
```

```
    background(0, 0, 0) # dark
```

If `is_day_time` is true, the code after the 'if' runs, painting the sky a dazzling blue. Otherwise, the code after the 'else' takes over, turning the screen into a dark night.

You can also adjust the `is_day_time` value to see how the if-else statement controls the background colour based on different conditions. Have fun experimenting!

Multiple if Statements

Not just one, but two... or more! Sometimes, a single condition isn't enough. Imagine leaving the house. You wouldn't just check if the door is open, right? You also need the key! That's where multiple conditions in if statements come in.

Logical Operators

and	Returns True if both statements are true	$x < 5$ and $x < 10$
or	Returns True if one of the statements is true	$x < 5$ or $x < 4$
not	Reverse the result, returns False if the result is true	$\text{not}(x < 5 \ \&\& \ x < 10)$

Table 5: Logical Operators

Think of logical operators like tools that combine boolean values (true/false) to make smarter decisions. Here are some key ones:

- and: Both conditions must be true. Like needing both the door open and the key.
- or: Only one condition needs to be true. Maybe you have a back door unlocked.
- not: Reverses the condition. Useful for checking if you DON'T have the key.

You can combine these operators and boolean variables to create intricate checks. Here are some examples:

- if $x < 10$: Simple check if x is less than 10.
- if $x == 25$ and $y < 10$: Checks if x is 25 AND y is less than 10 (both need to be true).
- if $\text{not}(\text{boolean_variable})$: Executes if the boolean_variable is False (e.g., you don't have the key).

While you can create very complex conditions, remember that readability is important . Overly complex checks can be hard to understand and debug. Keep it clear! Consider breaking down complex logic into smaller, more manageable conditions.

Multiple Branches: if Statements

We've seen if statements with two paths: follow the "true" door or the "false" door. But what if you have more options? Enter multiple branches!

Imagine, from our previous example, that we have three conditions to draw our sky colour: day time, night time and dawn. It no longer makes sense to use a boolean variable because it can only have 2 values; True or False.:

```
time_of_day = "dawn"

if time_of_day == "day":

    background(0, 0, 255) # blue skies!

elif time_of_day == "dawn":

    background(255, 125, 0) # orange

else:

    background(0, 0, 0) # dark
```

Where elif means else-if. So if is not "day" it will next check if it is "dawn". If it neither (else) it is assumed it must be dark.

We now have 3 potential conditions. When the time_of_day variable is:

- "day" we draw a blue background
- "dawn" we draw an orange background
- anything else we assume it must be night and draw a black background

There is an obvious flaw here of course, it could also be dusk. So, we can use our multiple conditions to fix this.

```
time_of_day = "dusk"

if time_of_day == "day":

    background(0, 0, 255) # blue skies!
```

```
elif time_of_day == "dawn" or time_of_day == "dusk":
```

```
background(255, 125, 0) # orange
```

```
else:
```

```
background(0, 0, 0) # dark
```

Now dusk and dawn are treated the same, assuming this is what we wanted to do, because we are saying that if the time_of_day is either “dusk” or “dawn” then draw an orange background.

Nesting if Statements

Imagine you have a complex decision to make, based on several factors. That's where nested if statements come in - they let you create layers of conditions within your code.

A nested if statement is an if statement inside another if statement. Consider this example:

```
time_of_day = "day"
```

```
weather = "not sunny"
```

```
if time_of_day == "day":
```

```
if weather == "sunny":
```

```
background(0, 0, 255) # blue skies!
```

```
else:
```

```
background(100, 100, 100) # grey skies
```

```
elif time_of_day == "dawn" or time_of_day == "dusk":
```

```
background(255, 125, 0) # orange
```

else:

```
background(0, 0, 0) # dark
```

We are now checking 2 things – the time of day and the weather. To read this you need to work through the decision branches:

- If the time_of_day is “day” then you go into that branch

- If then, the weather is sunny, background is blue. If not (else) it is grey

- If the time_of_day is either “dawn” or “dusk” then the background is orange

- In all other cases of time_of_day the background is black

Let’s look at a more complex example using the hours of the day to decide on the sky colour:

```
hour_of_day = 18
```

```
weather = "sunny"
```

```
size(200, 200)
```

```
if hour_of_day > 6 and hour_of_day < 18: # day time
```

```
if weather == "sunny":
```

```
background(0, 0, 255) # blue skies!
```

```
fill(255, 255, 0) #yellow
```

```
circle(30, hour_of_day * 10, 50)# draw the sun
```

```
else:
```

```
background(100, 100, 100) # grey
```

```
elif hour_of_day == 6 or hour_of_day == 18: # dusk or dawn
```

```
background(255, 125, 0) #orange
```

```
fill(255, 80, 0) # darker orange
```

```
circle(30, 190, 50) # draw the sun
```

```
else:
```

```
background(0, 0, 0) #dark
```

Here we have combined many of the concepts we have used so far. We have:

- 2 variables for the hour of the day and the weather
- An if statement with multiple branches that works differently depending on the hour of the day, using multiple conditions such as `hour_of_day > 6` and `hour_of_day < 18`, that determines the background colour of the sky
- A nested if statement that determines the weather, but only if the if statement that surrounds it is true (ie it is day time)
- A mathematical operator that determines the location of the sun using `hour_of_day * 10`

If you try changing the hour of day and the weather you should be able to see some differences in what is displayed. Try to work out what will happen in a given situation and then press run and see if you are right!

image

Figure 14: The sun at dusk



Chapter 4: Loops



Loops are when you want code to repeat. We'll explore two types of loops:

1. **for loops:** Execute code a specific number of times, ideal for tasks with a known repetition count.
2. **while loops:** Repeat code as long as a condition is true, perfect for situations where the number of repetitions isn't predetermined.

Each time the loop goes around it is called an iteration.

For Loops

Imagine you want to draw 10 circles on your screen. Writing the same `circle()` function 10 times would be tedious, right? That's where for loops come in! They're like having a helpful assistant who repeats a task for you a specific number of times.

Here's how it works

```
size(300, 100)
```

```
# Set the number of circles
```

```
num_circles = 10
```

```
# Start the for loop
```

```
for i in range(num_circles):
```

```
# This code will be repeated 10 times
```



```
# Change the color for each circle
```

```
fill(255 - i * 25, 0, i * 25)
```

```
# Draw a circle with different positions
```

```
circle(i * 20 + 30, 50, 20)
```

If you run this you will see that there are 10 circles whilst we only wrote the `circle()` statement once because there were 10 iterations.



Figure 15: Circles

You should be able to see the for loop statement in the code but we will explain the code:

```
num_circles
```

defines how many times the loop runs (10 in this case).

```
for i in range(num_circles):
```

This is the core of the loop. A for loop is a bit like an if statement. In this case it is saying to keep running the code inside the loop whilst the variable `i` is in the range of the number of circles (which is 10).

You may be thinking ‘but we never declared the variable `i`!’ and you’d be right. Sort of. The for loop contains the declaration of the variable, so you don’t need to declare it first. In this case it will declare a variable called `i` and give it the value 0 because, confusingly, loops start at 0 and not 1, and loops 10 times.

This means that each time the loop runs it will give the variable a new value (0, 1, 2, ...etc) until it reaches the range value `num_circles` when the loop will stop. The code inside the for loop can use the variable `i` if needed. In this case it is used to determine the colour and location of the circle:

```
# Change the color for each circle
```

```
fill(255 - i * 25, 0, i * 25)
```

```
# Draw a circle with different positions
```

```
circle(i * 20 + 30, 50, 20)
```

Try changing the value of `num_circles` and see how this changes the display.

Remember: Indentation is crucial in Python! Make sure the lines inside the loop are indented correctly.

Specifying a starting range value

It is possible to change the starting number for the loop variable (`i`)

```
size(300, 100)
```

```
# Set the number of circles
```

```
num_circles = 10
```

```
# Start the for loop
```

```
for i in range(4, num_circles):
```

```
# This code will be repeated 10 times
```

```
# Change the color for each circle
```

```
fill(255 - i * 25, 0, i * 25)
```

```
# Draw a circle with different positions
```

```
circle(i * 20 + 30, 50, 20)
```

The important line here is:

```
for i in range(4, num_circles):
```

Notice that we have added a number before `num_circles`. This is the starting value for `i`. So now the for loop will run between the range 4 and `num_circles`.



Figure 16: Circles with starting range

Specifying the increment

The amount by which the loop variable, in this case `i`, is increased each time is called the increment. By default this is 1 so that the variable is increased by 1 each time. We can also specify this if we want to:

```
# Set the number of circles
```

```
num_circles = 10
```

```
# Start the for loop
```

```
for i in range(4, num_circles, 2):
```

```
# This code will be repeated 10 times
```

```
# Change the color for each circle
```

```
fill(255 - i * 25, 0, i * 25)
```

```
# Draw a circle with different positions
```

```
circle(i * 20 + 30, 50, 20)
```

The important line here is

```
for i in range(4, num_circles, 2):
```

Notice that we have added a number after num_circles. This is the increment value for i. So now the for loop will run between the range 4 and num_circles but will increment by 2 each time. This gives the effect of leaving a gap in our circle example.



Figure 17: Circles with increment of 2

Experiment with combinations of these values to see what they do. As a reminder these are:

```
for i in range(starting_number, end_number, increment_value):
```

Or Else.. What?

Remember we said a for loop is like an if statement? Well, we can add else statements to for statements too!

```
size(300, 100)
```

```
# Set the number of circles
```

```
num_circles = 10
```

```
# Start the for loop
```

```
for i in range(4, num_circles, 2):
```

```
# This code will be repeated 10 times
```

```
# Change the color for each circle
```

```
fill(255 - i * 25, 0, i * 25)
```

```
# Draw a circle with different positions
```

```
circle(i * 20 + 30, 50, 20)
```

else:

```
square(50, 20, 20)
```

The else statement is run when the for loop has finished, so it will always run regardless of how many times it loops round. Note that you cannot use `i` in the else part because it belongs to the for part of the loop as that is where it was declared.



Figure 18: For loop else square

While Loops

We've seen how for loops repeat actions a fixed number of times. But what if we want to keep repeating until something specific happens? That's where while loops come in! Think of them like a persistent friend who keeps asking "Are you done yet?" until you give a specific answer.

Imagine this scenario:

- You want to draw stars falling from the top of the screen until they all reach the bottom.
- You don't know exactly how many stars you need because the display screen might be different sizes

A while loop is perfect for this!

```
size(200, 200)
```

```
# Initial position of the first star
```

```
star_y = 0
```

```
background(0, 0, 0) #black sky
```

```
# Keep drawing stars while they're above the bottom
```

```
while star_y < height:  
  
    # Move the star down a bit  
  
    star_y += 10  
  
    # Draw a star at the current position  
  
    fill(255, 255, 255) # white stars  
  
    ellipse(width // 2, star_y, 10, 10)
```

In this piece of code

- `star_y` keeps track of the star's current vertical position.
- `while star_y < height`: This is the condition that controls the loop.
- The loop keeps running as long as `star_y` (the star's position) is less than the height of the screen.
- Inside the loop:
 - We draw a star at the current `star_y` position.
 - We increase `star_y` by 10 to move the star down.
 - When the condition `star_y < height` becomes false (all stars are below the screen), the loop stops.



Figure 19: Falling stars in a while loop

Of course, we could have used a for loop for this because we do know how big the display window is as we set it using `size(200, 200)` at the top. But try changing the height of the window, using the second argument (the second 200) and you will see that it still works no matter how large the

window is! This wouldn't be the case with a for loop, we would have to work out how many stars we needed and change the loop.

While loops repeat until the condition, the boolean that checks whether to keep running, becomes false. In our example, when `star_y` hits the bottom of the screen. This means you don't need to know the number of times it will loop beforehand.

Think carefully about your condition to avoid infinite loops! Make sure the condition eventually becomes false so the loop ends. If you ever accidentally create a loop that runs forever, you can stop the programme using the square stop button at the top.

As with for loop, we can optionally have an else statement that runs at the end of the loop.

```
size(200, 200)
```

```
# Initial position of the first star
```

```
star_y = 0
```

```
background(0, 0, 0) #black sky
```

```
# Keep drawing stars while they're above the bottom
```

```
while star_y < height:
```

```
# Move the star down a bit
```

```
star_y += 10
```

```
# Draw a star at the current position
```

```
fill(255, 255, 255) # white stars
```

```
ellipse(width // 2, star_y, 10, 10)
```

```
else:
```

```
square(height/2, width/2, 20) # a square
```

This will still run even if the condition is never true ie the boolean that decides whether to loop is always false. Let's say, for example, we messed up our boolean logic like this:

```
while star_y > height:
```

This would never draw any stars because `star_y` is always less than the height of the screen. The square will still be drawn though! Try it and see.

Break

The `break` statement is used to stop the loop early before the Boolean is false, if needed. It would usually be used inside an `if` statement. For example:

```
if star_y > height/2:
```

```
    break
```

This would make the loop stop when the `if` statement condition is reached. In this case, it will only draw stars until `star_y` is greater than `height/2`, otherwise it will break, which ends the loop. This means it will only draw them in the top half of the screen. You could just change the loop variable of course, but this is just an example.



Figure 20: Half the stars

Continue

The `continue` statement is a little like `break()` but instead of ending the loop it skips the rest of that iteration and goes on to the next one. For example:

```
if star_y % 15 == 0:
```



```
continue
```

This code uses an if statement to check a modulus (remainder). In this case, if the modulus of the star_y value and 15 is zero (ie if it divides exactly by 15) then we skip the iteration of the loop. If the modulus is not exactly 0, the iteration will continue as normal.

```
size(200, 200)
```

```
# Initial position of the first star
```

```
star_y = 0
```

```
background(0, 0, 0) #black sky
```

```
# Keep drawing stars while they're above the bottom
```

```
while star_y < height:
```

```
# Move the star down a bit
```

```
star_y += 10
```

```
if star_y % 15 == 0:
```

```
continue
```

```
# Draw a star at the current position
```

```
fill(255, 255, 255) # white stars
```

```
ellipse(width // 2, star_y, 10, 10)
```

If you try this you will see that there are now gaps in our stars.

 image

Figure 21: Stars with gaps

Nested Loops

Just as we could have nested if statements, we can also have nested loops. So, a loop within a loop. They can be complicated though, so use them sparingly, but sometimes they are necessary. For example, perhaps we want to access every coordinate on the whole screen!

Firstly, let's just deal with the x-coordinate:

```
size(256, 256)

for x in range(width):

    stroke(x)

    point(x, height//2)
```

This loop ranges from 0 to the width of the screen drawing a coloured point at the x-coordinate that represents the current value of x in the loop. The colour is also based on the x value, so we get a line of dots that start dark (color(0)) and end at white (color(255)).



Figure 22: Pixels drawn in a loop

The points were all drawn at a specific y-coordinate, height//2, the middle of the screen. However, we could also use a loop to access all the y-coordinates and do something similar:

```
size(256, 256)

for y in range(height):

    stroke(y)

    point(width//2, y)
```

This would draw a line of points using the y-coordinates.

So, what if we wanted to do this for every x- and y-coordinate? We could draw a vertical line for every value of x, or draw a horizontal line for every value of y. Seems tedious, but there must be a simpler way, right? Of course there is! There's nested loops.

A nested loop, like a nested if statement, is a loop inside a loop. So we could combine the two loops we just wrote into one nested loop:

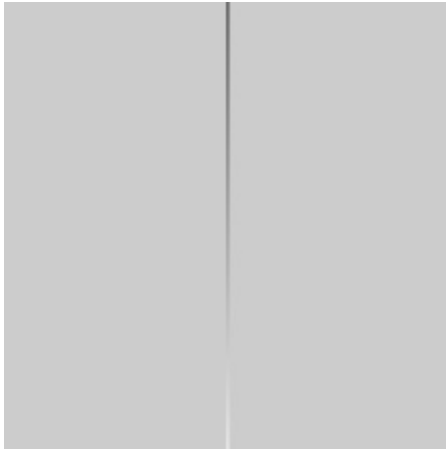


Figure 23: y-coordinate pixels drawn in a loop

```
size(256, 256)
```

```
for x in range(width): # loop 1
```

```
for y in range(height): # loop 2
```

```
stroke(x)
```

```
point(x, y)
```

This may take a bit of explanation, so bear with me.

The first loop (marked loop 1) is basically the same as before. It ranges from every value from 0 to 255, which we have called x.

The second loop (marked loop 2) is also basically the same as before. It ranges from every value from 0 to 255, which we have called y.

We have simply put the second loop inside the first - a nested loop. As loop 2 is inside loop 1 (note the indenting), the code in loop 2 has access to the code in loop 1. So, we can make use of both x and y inside loop 2. This means that we can draw a point at both x- and y-coordinates using the loops.

Loop 1 will range across the x-coordinates but, because loop 2 is inside it (ranging across the y-coordinates) it will, for each x value, do each y value. So, for every value of x, it will loop through every value of y. This covers every coordinate on the screen!

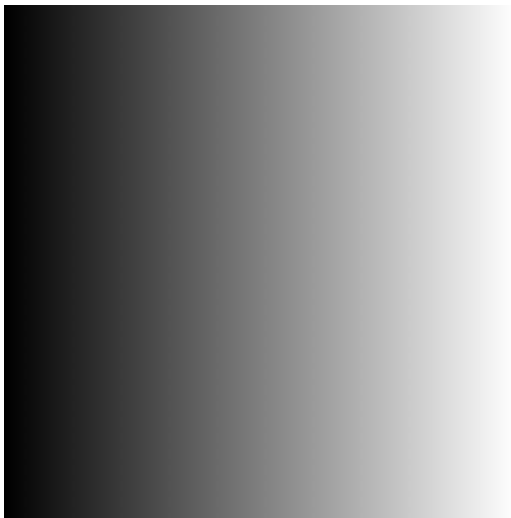
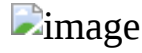


Figure 24: Coloured pixels in a nested loop



Chapter 5: Data Structures



Imagine that, with our previous falling stars example that instead of drawing them all one after another, you want to specify the coordinate to draw them, so you need to know where to draw them on the screen. Keeping individual variables for each star (like `star_y_1`, `star_y_2`, etc.) gets messy, especially if you have a lot of them. Wouldn't it be easier to have one place to store them all?

For this we can use data structures, there are several types.

	Ordered	Changeable	Allows duplicates
List []	✓	✓	✓
Tuple ()	✓	✗	✓
Set {}	✗	✗ *	✗
Dictionary {}**	✓	✓	✗

Table 6: Data structure types

* sets cannot be changed, but elements can be added and removed

**** dictionaries use a different way of creating elements and have a specific use**

Behold, the List!

Instead of having one variable for each time, think of a list as a flexible container that holds multiple values of the same data type. Instead of separate variables, you can store all your star locations in a list called `stars_y` (as it will be y-coordinates of the stars we draw):

```
# list of star locations
```

```
stars_y = [25, 37, 56, 96, 135, 185, 12]
```

Accessing Data: Like Picking Items from a Shelf

Each value in the list has a unique position, called an index. Imagine the star list as a shelf with numbered compartments. We call each value an element which are, confusingly, numbered from 0. So the first star at `stars_y[0]`, the second at `stars_y[1]`, and so on.

If we put these into a table, we can perhaps visualise what this looks like.

Element [0] 25

Element [1] 37

Element [2] 56

Element [3] 96

Element [4] 135

Element [5] 185

Element [6] 12

Table 7: Elements in a one-dimensional list

These are all contained in the one list variable we have called `stars_y`.

```
size(200, 200)
```

```
background(0, 0, 0) #black sky
```

```
# list of star locations
```

```
stars_y = [25, 37, 56, 96, 135, 185, 12]
```

```
circle(width//2, stars_y[0], 4)
```

```
circle(width//2, stars_y[1], 4)
```

```
circle(width//2, stars_y[2], 4)
```

```
circle(width//2, stars_y[3], 4)
```

```
circle(width//2, stars_y[4], 4)
```

```
circle(width//2, stars_y[5], 4)
```

```
circle(width//2, stars_y[6], 4)
```

This code creates the list with the stars, and then uses each star location from the list, using the index of each one, to draw a circle on the screen. Remember that the index numbers start from 0 and end at 6, because there are 7 values in the list. Attempting to access an index that doesn't exist will not work.

image

Figure 25: Star locations in an list

I guess this is OK for our 7 stars, but it would get pretty tedious to write if we had lots! Instead we can use a loop we've already seen: the for loop.

Conquering Lists with Loops

We've seen for loops before and they make it super easy to access all the elements in a list:

```
size(200, 200)

background(0, 0, 0) #black sky

# list of star locations

stars_y = [25, 37, 56, 96, 135, 185, 12]

for y in stars_y:

    circle(width//2, y, 4)
```

Here we have used a for loop to draw the stars in the exact same location but with far fewer lines of code. It also means that we can easily add another element to the list and the loop code doesn't need to change.

The important parts of this are:

□ `for y in stars_y:` – this is similar to the for loops we have already seen where:

- `'y'` is the loop variable that we can use in the loop. Each time the loop goes around it will be given a new value from the list
- `'in stars_y'` is the range, but this time it is the list we want to use
- `circle(width//2, y, 4)` draws a circle at the coordinates `width//2` (the middle) and `'y'`

Dynamic Lists

So far, we have declared, initialised and filled the list with one statement, for example:

```
# list of star locations
```

```
stars_y = [25, 37, 56, 96, 135, 185, 12, 13, 300]
```

Something we can do is add values to the list as the program is running. For example, maybe, instead of typing all the values in to our code (known as hardcoding) we want to calculate them:

```
size(200, 200)
```

```
background(0, 0, 0) #black sky
```

```
# list of star locations
```

```
stars_y = []
```

```
number_of_stars = 10
```

```
for i in range(number_of_stars):
```

```
    y = height/number_of_stars*i
```

```
    stars_y.append(y)
```

```
for y in stars_y:
```

```
    circle(width//2, y, 4)
```

Of course we could have done this with one loop, but we have separated them here to demonstrate the concept of adding values to a list. Note that:

- The list starts with no elements `stars_y = []`
- We add the values to the list as we calculate them using `stars_y.append(the_value_we_want_to_add)`

Lists with different types of elements

It is possible to add different types of elements to a list, although you have to think carefully before doing this because it becomes difficult to know what you'll get back out! For example:

```
stars = [1, 45, "yes I'm a star", 36.7]
```

is a valid list but now you have to remember what each element is for. So, it is possible, but not necessarily recommended.

Checking list contents

Having calculated the coordinates, we no longer really have an easy way of knowing what the contents of the list is. It is possible to check if a specific element is inside a list using an if statement like this:

```
if 25 in stars_y:
```

```
    fill(255, 0, 0) #red
```

This checks to see if the value 25 is in the list `stars_y`. Of course, this does rely on knowing the exact value we want to find. What if we want to make the stars red if they are the bottom half of the screen? In that case we can nest an if statement inside the loop:

```
size(200, 200)
```

```
background(0, 0, 0) #black sky
```

```
# list of star locations
```

```
for i in range(number_of_stars):
```

```
    y = height/number_of_stars*i
```

```
    stars_y.append(y)
```

```
for y in stars_y:
    if y > height/2:
        fill(255, 0, 0) #red
    else:
        fill(255) #white
    circle(width//2, y, 4)
```

Here we check the y coordinate of each star to see if it is in the bottom half of the screen ($> \text{height}/2$). If it is we use red, otherwise we use white.



Figure 26: Choosing colours in a loop

It is also possible to:

- replace elements in the list: `stars_y[1] = 45` # change element at index 1 to 45
- remove them: `stars_y.remove(220)` # remove element 220
- sort them: `stars_y.sort()` # sort into order
- copy them: `new_stars = stars_y.copy()` # copy list to a new list
- join them: `new_stars = stars_1 + stars_2` # create a new list as a combination of 2 other lists

and many other things!

Multi-Dimensional Lists

The lists we have seen up to now have been one-dimensional lists which are basically a list of items of a similar type. We can think of this as one column of values.

Element (0)

Element (1)

Element (2)

Table 8: A one-dimensional list

A two-dimensional list is best described as a table of tables, having rows and columns of items of a similar type, a bit like a spreadsheet.

Representing this as a table shows that it will have four rows and two columns. Each element is a list that has 2 elements in this example.

Element (0) Element (0,0) Element (0,1)

Element (1) Element (1,0) Element (1,1)

Element (2) Element (2,0) Element (2,1)

Table 9: A two-dimensional list

Employing two-dimensional lists in computer graphics can be quite useful as a table can be used for storing coordinate data. The use of programming constructs such as loops allows for each element of the list to be processed individually. The difference for a two-dimensional list is that, when we access an element in the list, we get another list! So, we need to access those as well.

You already know that a one-dimensional list is like this:

```
stars_y = [] # a list with no elements
```

or

```
stars_y = [25, 37, 56] # a list with 3 elements
```

We can visualise the elements in a table with 1 column.

Element (0) 25

Element (1) 37

Element (2) 56

Table 10: Elements in a one-dimensional list

A two dimensional list declaration is like this:

```
stars = [[], [], []] #2-dimensional list with 3 empty elements
```

or

```
stars = [[25, 67], [120, 46], [94, 32]] #2-dimensional list with 3 elements
```

If we visualise these elements in a table, there would be 2 columns.

Element (0) 25 67

Element (1) 120 46

Element (2) 94 32

Table 11: Elements in a two-dimensional list

The difference here is that the second list has two values: one to control the x-coordinate and another to control the y-coordinate. In theory, it doesn't really matter whether the first or second is considered to be the x or y-coordinate but the convention is to list the x-coordinate first. What is really important though is that they are used consistently throughout the code.

	x	y
Element (0)	25	67
Element (1)	120	46
Element (2)	94	32

Table 12: Coordinates in a two-dimensional list

This means now that, in this example, we have the coordinates for 3 stars, because there are 3 elements in the list. Each time we access an element we will be accessing a list containing 2 elements which we have decided are the x and y coordinates respectively.

If we create this list and extract the first element, we can print it out and see that it contains another list:

```
# list of star locations
```

```
stars = [[25, 67], [120, 46], [94, 32]] #2-dimensional list with 3 elements
```

```
star_0 = stars[0] # extract element 0
```

```
print(star_0)
```

The output on the console shows the contents of element 0, which contains the list of 2 elements.



Figure 27: Output of printing a list

This would allow us to retrieve the coordinates of each star and use them to draw it, so we can just use a loop as we did before. If we put the code above into a loop, instead of extracting one element at a time we can see that the contents of each element:

```
size(200, 200)

background(0, 0, 0) #black sky

# list of star locations

stars = [[25, 67], [120, 46], [94, 32]] #2-dimensional list with 3 elements

for star in stars: # loop through the list

    print(star) # we get another list!
```

The code here is similar to what we did before. We created a list (albeit, this time it is a list of lists) and loop through it. We will call this the outer list as it contains lists inside it. We are simply printing the contents at this point because each element is another list.



Figure 28: Contents of the outer list

Think about this code for a moment:

```
for star in stars: # loop through the list

    print(star) # we get another list! LINE 2
```

On the line where we have written LINE 2 we have access to the contents of the outer list. The first time it loops this will be element 0, so the element we now have, that we are printing out, is the first inner list:

```
[25, 67]
```

Of course it will then loop again and access the second element, which is the second inner list:

```
[120, 46]
```

And then the third. We can see this in the output that we printed out.

If we just had one of these lists, for example:

```
star = [25, 67]
```

We could just extract the values of the elements and use it to draw a circle (star) right? Let's do that!

```
circle(star[0], star[1], 4)
```

This would draw a circle using the first element as the x-coordinate (star[0]) and the second element as the y-coordinate (star[1]). So all we need to do is plug this into our loop:

```
size(200, 200)
```

```
background(0, 0, 0) #black sky
```

```
# list of star locations
```

```
stars = [[25, 67], [120, 46], [94, 32]] #2-dimensional list with 3 elements
```

```
for star in stars: # loop through the list
```

```
circle(star[0], star[1], 4)
```


This will draw all 3 stars using the 3 outer elements which in turn contain the 2 elements for the coordinates. So, we are directly using the elements in the inner list to draw circles by accessing the elements.

Now try adding some more inner lists and draw more stars! The sky's the limit!



Figure 29: Stars drawn using 2-dimensional lists

So, lists have an order (the elements are in the order of index number) and can be altered (we can add, remove and change the elements). It will also let you put duplicate values in as elements (so we could have the element 25 twice for example).

Tuples

Tuples are designed to be like lists you can't change. Once created, you can't add or remove any of the elements, so you would use this for something you didn't expect to change.

We have seen that we can change elements in a list and that it can contain duplicates. For example:

```
size(200, 200)
```

```
background(0, 0, 0) #black sky
```

```
# list of satellite locations
```

```
satellites = []
```

```
x = 2
```

```
while x < width:
```

```
satellites.append(x)
```

```
x += (x/2)

print(satellites)

for satellite_x in satellites: # loop through the list

circle(satellite_x, height//4, 4)
```

image

```
fill(0, 0, 255)

circle(width//2, height, width) # planet
```

This code calculates a set of x-coordinates for a series of ‘satellites’, adds them to a list and then displays them (and prints the values in the list). We can try adding a duplicate, such as:

```
satellites.append(2)
```

We will see that the list contains the value ‘2’ twice because lists allow duplicates. We won’t see the circle of course because it will just draw it over the top of the first.

image

Figure 30: Satellites drawn from a list

If we recreate this using tuples (which uses round brackets, rather than square), we see that this isn’t possible:

```
size(200, 200)

background(0, 0, 0) #black sky

# TUPLE of satellite locations
```

```
satellites = ()

satellites.append(2)

x = 2

while x < width:

    satellites.append(x)

    x += (x/2)

print(satellites)

for satellite_x in satellites: # loop through the list

    circle(satellite_x, height//4, 4)
```

image

```
fill(0, 0, 255)

circle(width//2, height, width) # planet
```

Note the round brackets when we create the satellites variable. This is now a tuple, not a list. However, running this results in the error “'tuple' object has no attribute 'append'” which is because you can't change a tuple by adding an element to it; once created they cannot be changed.

```
size(200, 200)

background(0, 0, 0) #black sky

# TUPLE of satellite locations

satellites = (10, 20, 40, 80, 160)
```

```
print(satellites)

for satellite_x in satellites: # loop through the list

    circle(satellite_x, height//4, 4)

fill(0, 0, 255)

circle(width//2, height, width) # planet
```

You also cannot change any of the elements already inside it (which you can with lists). For example:

```
satellites[0] = 5
```

Will result in the error “'tuple' object does not support item assignment” which means you can't set the value of an element inside it.



Figure 31: Satellites drawn from a tuple

You can, however, have duplicates. So, if we add 10 twice, you should see that it contains 10 twice.

So, tuples are like lists, but you can't change them. This is like a fail-safe that stops you accidentally altering something you didn't expect to change. More importantly, if you were working in a team, it tells other programmers that you didn't expect it to change and to stop trying to mess about with it!

Sets

Sets are like lists that can't change (sort of) and don't allow duplicates. They are written with curly brackets (braces). However, there is some oddity with creating them if you want to start with an empty set as we did with the list.

There are two ways to create a set:

1. Start with at least one element in it. For example, `satellites = {10, 20}`. However, if you only want one element, you still need the comma! For example, `satellites = {10,}`
2. Start with an empty set using the `set()` function. For example, `satellites = set()` You cannot start with empty braces (eg `satellites = {}`)

We fully appreciate that this is a bit confusing. Let's use our previous list example but with a set:

```
size(200, 200)
```

```
background(0, 0, 0) #black sky
```

```
# SET of satellite locations
```

```
satellites = set()
```

```
satellites.add(2)
```

```
x = 2
```

```
while x < width:
```

```
satellites.add(x) # note ADD not append
```

```
x += (x/2)
```

```
print(satellites)
```

```
for satellite_x in satellites: # loop through the list
```

```
circle(satellite_x, height//4, 4)
```

image

```
fill(0, 0, 255)
```

```
circle(width//2, height, width) # planet
```

There are a few things to note here.

Firstly, we used the `set()` function to create an empty set. We could also have used:

```
satellites = {2,}
```

Since we immediately add the value 2 to the set. Note also the use of the `add()` function, rather than `append()`. Sets use `add()` instead. But hold on, didn't I say sets can't be changed? I did, but I also said 'sort of'. Sets can have elements added and removed, but you can't change the value of the elements. You cannot access set elements using the index number, so you can't use the line:

```
satellites[0] = 5
```

So, sets can have elements added and removed, but you can't change the value of the elements in them.

Lastly, sets do not allow duplicates. When we added the value 2 to a list, it allowed it to be included twice. Sets do not. They also don't complain, they will just ignore it if you try to add it again.

So, sets are useful when you don't want to allow duplicates.

Dictionaries

Dictionaries are the odd-ones-out in the Python data structure types because they are quite different. They still hold multiple values, but this time in what is known as key-value pairs. This means that each element has a 'key' (something used to define it, like a variable name) and a 'value' (the value that corresponds to the key).

For example:

```
colours = {  
    "planet_colour": color(20, 255, 180)  
}
```

Here the key is the string “planet_colour” and the value is the colour color(20, 255, 180). This would allow us to access the colour we wanted by looking it up from the dictionary called colours. For example:

```
fill(colours["planet_colour"])
```

This will access the value of the key “planet_colour” and use it in the fill() function.

You can more than one dictionary entry, such as:

```
colours = {  
    "planet_colour": color(20, 255, 180),  
    "satellite_colour": color(255, 255, 0)  
}
```

We can then use this in our code example:

```
size(200, 200)  
background(0, 0, 0) #black sky  
colours = {  
    "planet_colour": color(20, 255, 180),  
    "satellite_colour": color(255, 255, 0)  
}
```

```

# SET of satellite locations

satellites = set()

x = 2

while x < width:

    fill(colours["satellite_colour"])

    satellites.add(x) # note ADD not append

    x += (x/2)

print(satellites)

for satellite_x in satellites: # loop through the list

    circle(satellite_x, height//4, 4)

    fill(colours["planet_colour"])

    circle(width//2, height, width) # planet

```

Dictionaries allow duplicate values, but not duplicate keys. Of course, we could just have used variables but dictionaries are useful when you have some key-value pairs that are all about one thing so you can store them together.

So, you have now seen the four data structure types available in Python: lists, tuples, sets and dictionaries. They all have different uses but are available should you need them.

Ordered Changeable Allows duplicates

List []	✓	✓	✓
---------	---	---	---

Tuple ()	✓	✗	✓
Set {}	✗	✗ *	✗
Dictionary {}**	✓	✓	✗

Table 13: Reminder of data structure types

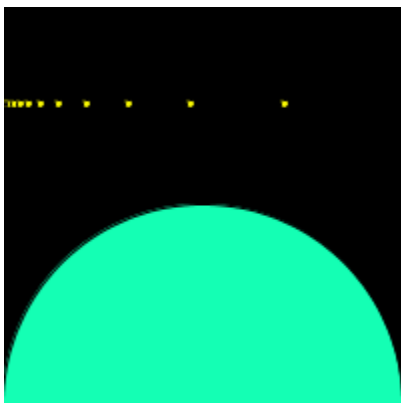


Figure 32: Satellites using a dictionary



Chapter 6: Functions



Built-in Functions

We have seen functions before but just not described exactly what they were although you will by now understand what they do. Some examples of functions we have used are:

`size(600, 400)`

`circle(30, 30 20)`

`background(0)`

You will recognise these by now as letting you change the size of the display screen, draw a circle and change the background colour. They are what we call built-in functions, because they were provided for us. Functions perform a specific action and usually have a relevant name that tells you what it will do. The brackets are known as arguments or parameters, which are information we need to give the function so it knows what to do. For example, we would say the size function has 2 arguments – one for the width and one for the height. Some functions may not have any arguments and in that case they would just have empty brackets, such as the `sort()` function we saw when dealing with lists.

We will see many more built-in functions but let's write some of our own!

Writing your own functions

Up to this point all our coding has been linear: that is to say our programmes start at the top and work their way right through until they get to the bottom. There are times when we've almost repeated the same pieces of code within the same programme. Wouldn't it be great if there was a way that you could just write that bit once and then tell the computer to use it again and again? This is where functions come in.

A function is like a mini piece of code that can be used over and over by the main programme code. Every function has a name and, optionally, has one or more arguments passed to it. The basic structure of a function is:

```
def function_name (argument_list):
```

```
// function content
```

This is made up of several parts:

- the keyword `def` is short for define and means we are about to create a function
- the function name largely follows the normal naming conventions that you would expect for any identifier except it should start with a verb eg `do_something`
- the round brackets enclose any arguments that the function will be expecting. These are information the function needs to work. If no information is needed, the brackets are still included but will be empty
- the colon `:` marks the beginning of the code that comprises the function, this can range from a simple statement of just a single line to a complex routine of many lines. The function code should be indented to show it belongs to the function

If you ever find yourself writing a function with more than five or six arguments though, then your function is most likely trying to do too much and should be examined to see if it can be broken down into two or more simpler functions. This is because the more complicated a function is, the harder it is to work out where any issues may arise and the easier it is for an error to slip through unnoticed.

Let us look at writing an example. Let's say we find we are repeatedly drawing a paw print.

We could of course, write the code several times and just change the coordinates, but this involves more code and, more importantly, more margin for error when creating or changing the code.

Instead we could create a function that does this for us.

```
def paw():  
  
    circle(width/2 - 12, height/2, 20)  
  
    circle(width/2 + 12, height/2, 20)  
  
    circle(width/2 - 35, height/2 + 15, 20)  
  
    circle(width/2 + 35, height/2 + 15, 20)  
  
    circle(width/2, height/2 + 40, 40)
```



Figure 33: Paw print

This code defines a function called `paw()` that has no arguments. If we wanted to use this function (known as 'calling') we can now do so, like we would with any of the built-in functions. However, it will only work in the programme where we defined it, and it must be defined before (ie above) any call to the function.

We would use it like this:

```
size(400,400)  
  
def paw():  
  
    circle(width/2 - 12, height/2, 20)
```

```
circle(width/2 + 12, height/2, 20)

circle(width/2 - 35, height/2 + 15, 20)

circle(width/2 + 35, height/2 + 15, 20)

circle(width/2, height/2 + 40, 40)

paw() # call the paw function
```

Of course, at the moment, this doesn't help us because it will always just draw the paw print in the same place, so repeatedly calling it will draw them over the top of the previous one. What we want is to be able to specify where to draw the paw. We need arguments.

```
size(400,400)

def paw(x, y):

    circle(x - 12, y, 20)

    circle(x + 12, y, 20)

    circle(x - 35, y + 15, 20)

    circle(x + 35, y + 15, 20)

    circle(x, y + 40, 40)

paw(50, 20) # call the paw function
```

Lets examine the differences between this version and the previous one.

- The paw() function has 2 arguments, x and y so it became paw(x, y)
- We changed the circle() functions calls to use the new x and y arguments rather than height and width
- We changed the paw() function call to be paw(50, 20) so that it now provides (or 'passes') the 2 arguments (x and y) it now expects, in that order.

What we have defined here is a function called `paw()` that needs 2 arguments: `x` and `y`. When we call this function, in this example using `paw(50, 20)`, it finds the function called `paw()` and the `x` argument becomes the first value we gave it (50) and the `y` argument becomes the second value (20).



Figure 34: Argument values

This now means we can draw paw prints wherever we want!

```
size(400,400)

def paw(x, y):

    circle(x - 12, y, 20)

    circle(x + 12, y, 20)

    circle(x - 35, y + 15, 20)

    circle(x + 35, y + 15, 20)

    circle(x, y + 40, 40)

paw(50, 20) # call the paw function

paw(100, 200)

paw(300, 100)

paw(300, 300)
```

Here we just add more calls to the `paw()` function and give it different argument values. This draws the paw prints at the location we specify with hardly any extra code!



Figure 35: Using a function to draw multiple paw prints

This is a very powerful aspect of functions and the design of the argument list. It should be noted, however, that a single function should be limited to performing one specific task—if a function is designed to draw a shape, do not expand it to also calculate some complex mathematical sum: that should be the job for a different function. Sometimes it is all too tempting to add in a bit of extra code because ‘it’s doing something else with the same arguments’, but this generally just leads to errors that are difficult to track down and makes the code harder to understand. Imagine if the `circle()` function also draw a square! You would be very confused. In that case you should use two different function calls to achieve your squarey-circle.

We could even extend our example with a third argument for the size of the paw print

```
size(400,400)
```

```
def paw(x, y, paw_size):
```

```
    circle(x - (paw_size * 0.6), y, paw_size)
```

```
    circle(x + (paw_size * 0.6), y, paw_size)
```

```
    circle(x - (paw_size * 1.75), y + (paw_size * 0.75), paw_size)
```

```
    circle(x + (paw_size * 1.75), y + (paw_size * 0.75), paw_size)
```

```
    circle(x, y + (paw_size * 2), paw_size * 2)
```

```
paw(50, 20, 20) # call the paw function
```

```
paw(100, 200, 20)
```

```
paw(300, 100, 20)
```

```
paw(300, 300, 20)
```

Our `paw()` function now needs 3 arguments:

- x – the x coordinate to draw the paw
- y – the y coordinate to draw the paw
- paw_size – the size of the paw. Note that we had to use ‘paw_size’, not ‘size’, because ‘size’ is already a word that Processing understands, known as a key word



Figure 36: Three argument values

Try changing the coordinates and sizes of the paw prints, to check you understand how this works. Try adding more calls to the paw() function or even better, use it in a loop for loads of paw prints!



Figure 37: Paw print function in a loop

Default arguments

It is possible to specify a ‘default’ value for arguments so that, if you don’t specify the value, it will use the default one instead. For example, we might want to specify the size of the paw print if it is not specified in the call to the function.

We do this by specifying the default value in the function definition like this:

```
def paw(x, y, paw_size=10):
```

This tells the function that, if we don’t specify the paw_size value then use 10. But if we do specify the value, use that instead. That makes either of these function calls successful:

```
paw(100, 100, 20) # paw size 20
```

```
paw(200, 200) # use default paw size
```


Return Values

Return values are exactly as they sound: they are values that are returned by the function. We would usually use these if the function was expected to work something out and give us an answer.

Functions that do not return a value are useful – these can be considered ‘orders’: Do this! Functions that return a value can be considered a request for information: Tell me the answer to this. This is achieved by use of return values.

One simple example is a function that calculates something for us, such as the width of our paw print given the specified size.

```
def paw_width(paw_size):  
  
    return paw_size * 3.5
```

This function calculates the width of the paw given the paw size and returns that value. How do we use this returned value though? We can assign it to a variable (remember those?) perhaps:

```
each_paw_width = paw_width(10) # find width of paw size 10
```

This will call the `paw_width()` function, providing the argument value of 10. The `paw_width` function will calculate the value for us (`paw_size * 3.5`) and return it. This is then stored in the variable we created `each_paw_width` which now holds the value 35.

You can print this out to verify it for yourself if you wish:

```
each_paw_width = paw_width(10) # find width of paw size 10  
  
print(each_paw_width)
```

It is also possible to chain function calls together. For example, this code does the same thing as the code above:

```
print(paw_width(10))
```

Which we appreciate can be confusing to read at first sight! This is just the `paw_width()` function call and the `print()` function call together. If we ran this, this is what would happen:

1. Call the `paw_width` function with the value 10
2. `paw_width` function calculates the width, given the value 10 as its argument
3. `paw_width` function returns the calculated value 35
4. `print()` function prints the returned value

If it is easier for now though, keep using variables to store the values and then use them in the next function.

Multiple Return Statements

As a last example if using function, let's use these functions together. One function that you might find you often need is whether a shape will be drawn off the edge of the screen, so let's create a function that tells us the answer!

```
def is_on_screen(x, paw_size):
```

```
    one_paw_width = paw_width(paw_size) # the function we just wrote
```

```
    if x + (one_paw_width/2) > width:
```

```
        return False
```

```
    elif x - (one_paw_width/2) < 0:
```

```
        return False
```

```
    else:
```

```
        return True
```

This new function uses the function we just wrote to find out the width of the paw print and then uses it to calculate if the paw print will be printed off the screen or not. It does this using more than one return statement, but only one will get used at a time:

- If the first boolean condition is true, it will return False
- If the first boolean condition is false, but the second is true, it will return False
- If the first and second conditions are false, it will return True

This will then determine if the paw print will be drawn off the screen. We can then use this function to decide whether to bother drawing it!

```
def paw(x, y, paw_size=10):  
  
    on_screen = is_on_screen(x, paw_size) # will this be on screen?  
  
    if on_screen:  
  
        circle(x - (paw_size * 0.6), y, paw_size)  
  
        circle(x + (paw_size * 0.6), y, paw_size)  
  
        circle(x - (paw_size * 1.75), y + (paw_size * 0.75), paw_size)  
  
        circle(x + (paw_size * 1.75), y + (paw_size * 0.75), paw_size)  
  
        circle(x, y + (paw_size * 2), paw_size * 2)
```

We can use the new function in our existing one to decide whether to draw the paw print or not. If it will be on the screen, then we draw it. If not, we do nothing.

Here is all the code:

```
size(400,400)  
  
def paw(x, y, paw_size=10):
```

```

on_screen = is_on_screen(x, paw_size) # will this be on screen?

if on_screen:

    circle(x - (paw_size * 0.6), y, paw_size)

    circle(x + (paw_size * 0.6), y, paw_size)

    circle(x - (paw_size * 1.75), y + (paw_size * 0.75), paw_size)

    circle(x + (paw_size * 1.75), y + (paw_size * 0.75), paw_size)

    circle(x, y + (paw_size * 2), paw_size * 2)

def paw_width(paw_size):

    return paw_size * 3.5

def is_on_screen(x, paw_size):

    one_paw_width = paw_width(paw_size) # the function we just wrote

    if x + (one_paw_width/2) > width:

        return False

    elif x - (one_paw_width/2) < 0:

        return False

    else:

        return True

paw(400, 20, 10) # call the paw function

```

Try changing the argument values for the paw() function call to check that it will only draw the paw prints if they will be on the screen. You've written your first functions!

Global and Local Variables

The last thing we want to look at with functions is something known as variable scope. Variables have a home and can only live where they are told. We call this scope. For example in our paw print program we created a variable called `one_paw_width` in `is_on_screen()` function

```
def is_on_screen(x, paw_size):  
  
    one_paw_width = paw_width(paw_size) # the function we just wrote  
  
    if x + (one_paw_width/2) > width:  
  
        return False  
  
    elif x - (one_paw_width/2) < 0:  
  
        return False  
  
    else:  
  
        return True
```

This is an example of what we call a local variable. A local variable has the scope of the function it is declared in. So, if we try to use it outside the function then it will not be recognised.

If you put this line:

```
print(one_paw_width)
```

In another function, or not in a function at all, you will get an error message.

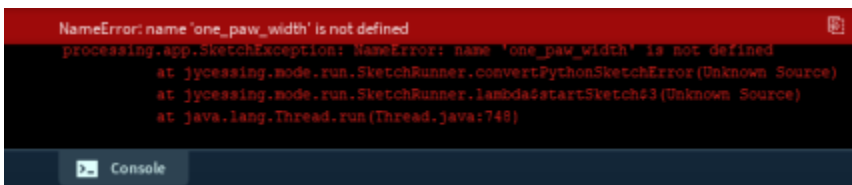


Figure 38: Variable scope error

The same applies to the variable names we use in functions such as the one we wrote to draw paw prints:

```
def paw(x, y, paw_size=10):
```

```
# etc
```

In this function, `x`, `y` and `paw_size` are local variables and can only be used inside the function. Didn't I say they were called arguments? Yes I did, they can be used interchangeably but technically the difference is:

- A variable is the name we give to hold a value
- An argument is the actual value we give to the variable

What if you have a variable that you want to access in more than one function, or outside a function? You have 2 choices:

- Pass the variable as an argument to the function
- Use a global variable

We have already looked at passing values to functions, and you can do the same with variables as they represent values. For example, we made this function:

```
def is_on_screen(x, paw_size):
```

```
    one_paw_width = paw_width(paw_size)
```

```
# etc
```

Which has 3 arguments, and we call it like this:

```
paw(100, 20, 10) # call the paw function
```

So, the argument `x` becomes 100, `y` becomes 20, `paw_size` becomes 10. Inside the `is_on_screen()` function we then use the local variable `paw_size` in another call to the `paw_width()` function. In this case we are passing on

the value of the variable to the other function `paw_width()`. There is no limit to how many times it could be passed on.

Global Variables

Global variables are variables that can be used anywhere in your program once declared. If we declare a variable outside of a function, it is said to have global scope. For example we might want to create a loop for displaying our paw prints

```
number_of_paws = 20 # global variable

for i in range(number_of_paws):

    paw(i * 20, i * i, 10) # call the paw function
```

Here the `number_of_paws` variable is a global variable that can be used inside or outside functions in our program because it was declared in the scope of the entire program, not inside a function.

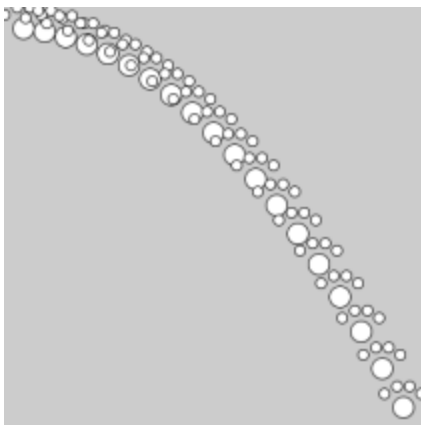


Figure 39: Paws in a loop



Chapter 7: Animation



Have you noticed that all our programmes so far have been like snapshots; unchanging pictures on the screen? That's because we've been using static mode, one of the ways Processing can run code.

Static Mode

Think of static mode like taking a photograph. You write your code, Processing draws the graphics once, and then it freezes the image on the screen. It's a simple approach, but it limits what you can create.

Active Mode

Active mode differs from static mode because the display screen is drawn continuously, rather than just once. This means that we can change aspects of what we draw to make your graphics come alive!

Unlike static mode, active mode uses two functions: `setup()` and `draw()`

`setup()`

Think of `setup()` as your animation's blueprint. Here, you define crucial things like window size, initialise variables, or pre-draw parts of the program that won't change. It runs only once at the beginning. This is used similarly to creating your own function, except it has to be called `setup()`:

```
def setup():
```

```
# Set window size, initialize variables, etc.
```



```
size(400, 200)

background(220) # Set background colour

# Your other essential setup tasks here
```

draw()

This is where the magic happens! `draw()` is a continuous loop that redraws your programme constantly. Think of this like a stop-motion movie. Each time the drawing is changed slightly and this creates the impression of movement. This allows you to update your animation each time, creating movement and interactivity.

```
def draw():

# Update values, draw shapes, respond to user input, etc.

fill(0) # Set filling colour for shapes

circle(0, height//2, 50) # Draw a circle

# Your animation logic and updates here
```

If you run this though, you will see that there is no animation and the circle just sits in the middle of the screen whereas I said it would be animated. Can you see why?



Figure 40: Stationary circle

It's because the circle is always being drawn in the same place! So, although this is animated, each frame is being drawn exactly the same. To create the illusion of movement, we need to move the circle a little each time the frame is drawn by altering it in some manner. Let's alter the x-coordinate for the circle to use a variable instead of hardcoding it:

```
x = 0 # global variable

def setup():

    size(400, 200)

    background(220) # Set background colour

    def draw():

        global x # use the global variable

        fill(0) # Set filling colour for shapes

        circle(x, height/2, 50) # Draw an ellipse

        x+=1
```

This allows us to use the x and y variables to draw the circle. So, we can now change those values each time the frame is drawn by adding 1 to x using `x+=1` which should mean that the circle is drawn progressively from left (because that is x-coordinate 0) to right. Notice the use of the global variable and that we had to say that we intended to change the global variable inside the function using:

```
global x # use the global variable
```

If we didn't do this, it would look for a local variable called x and wouldn't find one. This allows you to use local and global variables with the same name if you wish to.

If you run this you might see that it doesn't quite do what we had hoped. We get an increasing bar rather than a circle that moves.

image

Figure 41: Overlapping circles

The reason for this is simple, but not obvious until you realise what is happening. Let's look at the draw() function that we wrote

```
def draw():
```

```
    global x # use the global variable LINE 1
```

```
    fill(0) # Set filling colour for shapes LINE 2
```

```
    circle(x, height/2, 50) # Draw an ellipse LINE 3
```

```
    x+=1 # LINE 4
```

We told the draw function to:

- Use the global x variable (LINE 1)
- Fill the circle with the colour black (LINE 2)
- Draw the circle (LINE 3)
- Add 1 to the global variable x (LINE 4)

And this is exactly what happened, right? The draw function loops around and around until stopped. So, each time it looped it drew a circle. At no point did we tell it get rid of the previous circle. So, it just drew them over the top of each other!

The easiest way to fix this is to clear the screen between each frame using the background() function. If we move our background(220) to the draw function it will clear the screen each time, then draw a circle, giving us the effect of moment:

```
x = 0
```

```
def setup():
```

```
    size(400, 200)
```

```
def draw():
```

```
    background(220) # Set background colour
```

```
global x # use the global variable  
  
fill(0) # Set filling colour for shapes  
  
circle(x, height/2, 50) # Draw an ellipse  
  
x+=1
```



Figure 42: Moving circle

Active Mode Controls

loop() and noLoop()

What if you want to pause the action or resume it later? Here's where loop() and noLoop() come in!

Think of noLoop() as a "pause" button. Call it, and draw() stops looping, freezing your animation like a paused movie. The scene you see on the screen is the last frame drawn before the pause.

```
if x >= width//2:
```

```
noLoop()
```

This will stop the circle from moving if it reaches the middle of the screen.

To get things moving again, just call loop(). This is like pressing "play" on your paused movie so that the programme starts looping again, and your animation continues from where it left off.

Frame Rate

Remember how Active Mode draws frames constantly, creating smooth animations? But what if you want to slow things down or speed them up? That's where frameRate() comes in!

Adjusting Animation Speed

Imagine `frameRate()` as a dial controlling the animation's tempo. By default, it's set to 30 frames per second (fps), like a normal movie. Increase the value (e.g., `frameRate(60)`) for a faster animation. Decrease it (e.g., `frameRate(10)`) for a slower, more deliberate feel.

Experimenting with Speed

Try adding `frameRate(5)` to your sketch. Observe how the background colour changes at a slower pace, creating a more sluggish animation. This demonstrates how changing the frame rate directly affects the speed of your visuals.

There are some useful built-in variables that go along with this:

- `frameRate` - the current frame rate within your code you can use the variable.
- `frameCount` – the number of frames have been drawn so far

Animation

Let's animate our circle to be a little more exciting by animating the circle as a bouncing ball which will bounce off the sides of the window.

First of all, let's have it bounce sideways. We can do this with one new global variable at the top and 2 if statements in `draw()`:

```
go_right = True # should ball go right (or left)
```

```
global go_right
```

```
# check if ball hits the edge
```

```
if x >= width: # if we hit the right side
```

```
go_right = False
```

```
elif x <= 0: # if we hit the left side
```

```
go_right = True
```

```
# send ball in a specific direction
```

```
if go_right:
```

```
x+=1
```

```
else:
```

```
x-=1
```

Adding this into your program will make the ball bounce off the left and right sides and, because it keeps animating until you stop it, the ball will just keep bouncing!

Let's just add a little more to make it bounce off the top and bottom edges. We basically just need to replicate what we have done with the x-coordinate with a new y-coordinate variable:

```
y = 0
```

```
go_up = False
```

```
and some new if statements:
```

```
global go_up
```

```
if y >= height: # if we hit the bottom
```

```
go_up = True
```

```
elif y <= 0: # if we hit the top
```

```
go_up = False
```

```
# send ball in a specific direction
```

```
if go_up:
```

```
y-=1
```

```
else:
```

```
y+=1
```

Then just change it so the circle is drawn at the x- and y-coordinates:

```
global y
```

```
fill(0) # Set filling colour for shapes
```

```
circle(x, y, 50) # Draw an ellipse
```

The entire code as we have developed it is below, but you should see that now the ball bounces of any of the four sides!

image

Figure 43: Bouncing ball animation

```
x = 0
```

```
y = 20
```

```
go_right = True # should ball go right (or left)
```

```
go_up = False
```

```
def setup():
```

```
# Set window size, initialize variables, etc.
```

```
size(400, 200)
```

```
# Your other essential setup tasks here
```

```
def draw():
```

```
background(220) # Set background color

global x # use the global variable

global y

# Update values, draw shapes, respond to user input, etc.

fill(0) # Set filling colour for shapes

circle(x, y, 50) # Draw an ellipse

# Your animation logic and updates here

global go_right

# check if ball hits the edge

if x >= width: # if we hit the right side

    go_right = False

elif x <= 0: # if we hit the left side

    go_right = True

# send ball in a specific direction

if go_right:

    x+=1

else:

    x-=1

global go_up

if y >= height: # if we hit the bottom
```



```
go_up = True
```

```
elif y <= 0: # if we hit the top
```

```
go_up = False
```

```
# send ball in a specific direction
```

```
if go_up:
```

```
y-=1
```

```
else:
```

```
y+=1
```


CHAPTER 8: COLOUR

Processing and Colour

The `color()` function (note the American spelling) works somewhat like printer ink such that we can combine colours from a set of 3 primary colours: Red, Green, Blue (RGB).



Figure 44: Printer ink cartridges

We specify the colour in the `color()` function using either one argument, or three:

```
color(grey)
```

```
color(red, green, blue)
```

All of the argument values are in the range 0 (no intensity) to 255 (full intensity). In the case of the one-argument version, this is just a scale of grey from 0 (black) to 255 (white).

The three-colour function specifies the amount of red, green, blue in that order. So, for example `color(255, 0, 255)` will be a purple colour as it has maximum red and maximum blue but no green.

That means that there are 256 variations for red times 256 for green times 256 for blue... that's a staggering 256 x 256 x 256, or 2 to the power of 24, possible colours! Written out, that's a whopping 16,777,216 colours at your fingertips which is more than the human eye can differentiate. That's an incredible palette to paint your digital creations with!

Red Green Blue Colour

0	0	0	Black
0	0	255	Blue
0	255	0	Green
255	0	0	Red
255	255	0	Yellow
255	0	255	Magenta
0	255	255	Cyan
255	255	255	White

Table 14: Major RGB colours

There are a few ways you can use colour

Background Colour

You might like to try some colour variations by setting them as a `background()` argument, for example:

```
background(100, 100, 250)
```

Fill Colour

Fill colour allows you to set the infill the colour of shapes (as opposed to the outline):

```
size(200, 200)
```

```
fill(100, 200, 150)
```

```
square(10, 10, 50)
```

The fill colour will be used for all shapes drawn after it until the colour is changed:

```
size(200, 200)
```

```
fill(100, 200, 150) #colour 1
```

```
square(10, 10, 50)
```

```
square(100, 10, 50)
```

```
fill(50, 170, 250) #colour 2
```

```
circle(50, 100, 50)
```

```
circle(150, 100, 50)
```

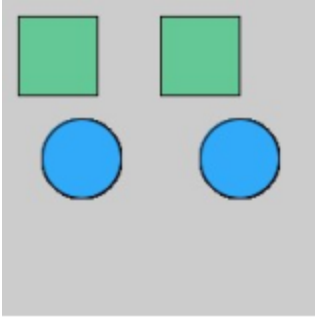


Figure 45: Coloured shapes

Stroke Colour

As you can see the shapes have a black outline. This is the *stroke colour*. Stroke colour is also used for outlines, and also points (dots) and lines.

For example, we can change the program we were just using to have white lines instead of black:

```
size(200, 200)

stroke(255) # white lines

fill(100, 200, 150) #colour 1

square(10, 10, 50)

square(100, 10, 50)

fill(50, 170, 250) #colour 2

circle(50, 100, 50)

circle(150, 100, 50)
```

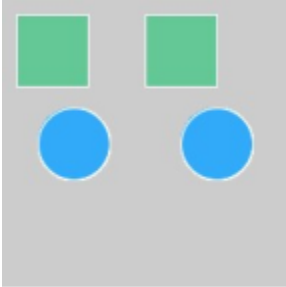


Figure 46: Changing the stroke colour

You can also turn off these lines using `noStroke()` instead of specifying the `stroke()` colour. Use the `stroke()` function to turn it back on.

Of course you can try changing any of the values for R, G and B to between 0 and 255 and see what happens!

Red Green Blue Colour

128	128	128	Mid Grey
100	100	250	Mid Blue
20	60	20	Olive Green
255	100	0	Orange
0	0	40	Navy Blue
160	240	230	Eggshell Blue
70	0	10	Crimson

255 210 210 Pink

Table 15: More RGB colours

Alpha Transparency

There is also another aspect to colour which we haven't yet mentioned – transparency, known as the alpha value. This is added as an extra argument but is optional. This means there are actually 4 ways to create colours:

```
color(grey) # no transparency
```

```
color(red, green, blue) # no transparency
```

```
color(grey, alpha)
```

```
color(red, green, blue, alpha)
```

The value is relative to the range of the colour so for the RGB model, where the maximum value is 255, then 255 is totally opaque and is 0 totally transparent.

Let's first try it without the alpha value using two shapes:

```
size(200, 200)
```

```
fill(0, 0, 40) #navy blue
```

```
circle(width//2, height//2, 30)
```

```
fill(255, 100, 0) # orange
```

```
circle(width//2, height//2, 20)
```

Here we draw two circles at the same point. The order matters, so we are drawing the larger, blue circle first, then the smaller, orange circle on top of

it, because that is the order of our code.



Figure 47: Opaque circles in size order

If we draw them in the opposite order, you will no longer be able to see the smaller circle because the larger circle is covering it up, much like a solar eclipse:

```
size(200, 200)

fill(255, 100, 0) # orange

circle(width//2, height//2, 20)

fill(0, 0, 40) #navy blue

circle(width//2, height//2, 30)
```

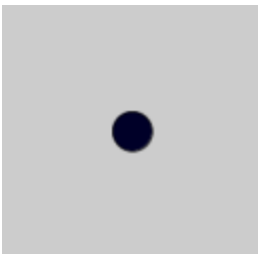


Figure 48: Opaque circles in reverse size order

The orange circle is still being drawn, we just can't see it because of the circle drawn over the top of it. If we increase the transparency (or rather,

decrease the opacity) of the circle though, we will be able to see the circle underneath as well:

```
size(200, 200)

fill(255, 100, 0) # orange

circle(width//2, height//2, 20)

fill(0, 0, 40, 125) # transparent navy blue

circle(width//2, height//2, 30)
```

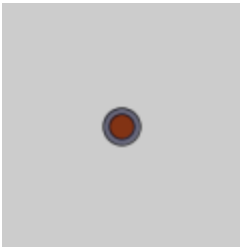


Figure 49: Transparent circle

Colour Modes

Though the RGB colour scheme is more widely known, *Processing* allows colours to be described using two quite distinct numerical models: RGB and HSB (Hue, Saturation, Brightness) using the `colorMode()` function:

```
colorMode(mode)
```

The default mode is RGB so there is no need to specify this unless you have changed it.

```
colorMode(RGB)
```

Let's first create a program that shows off the range of colours using RGB colour mode

```
def setup():  
  
    size(200, 200)  
  
    colorMode(RGB) # You can experiment with changing this to HSB  
  
    def draw():  
  
        # Nested loops for each pixel  
  
        for x in range(width):  
  
            for y in range(height):  
  
                # Calculate color based on coordinates  
  
                r = x  
  
                g = y  
  
                b = r + g # Combine values for more interesting variation  
  
                # Set stroke colour  
  
                stroke(color(r, g, b))  
  
                # Draw a point at the current coordinates  
  
                point(x, y)
```

This program uses two for loops to draw points (dots) on the screen using different colours. Notice that we have used a nested loop. The reason we are doing this is because we want to make sure we draw a coloured point on every pixel on the display screen.

- The first loop is the x-coordinate horizontal range
- The second, nested, loop is the y-coordinate vertical range

This means that for *each* horizontal line, we need to cover the entire range of vertical pixels. The loop will therefore start at x-coordinate 0, and work downwards through all the y-coordinates, before moving on to x-coordinate 1 etc



Figure 50: RGB coloured pixels

Consider the coordinates of the display screen. The first loop is the x-coordinates, so starts at $x = 0$.



Figure 51: Coordinates on the screen

For every x-coordinate, in this case, 0, the nested loop will loop through every y-coordinate. So (0,1), (0,2) etc all the way to (0,200).

The outer loop will move onto the next x-coordinate, $x = 1$, and then the inner loop will loop through all the y-coordinates again, (1,1), (1,2) etc all the way to (1,200).

This way we cover every coordinate on the screen.

The effect of this is to cover every pixel on the screen and colour it in **HSB**

We can now easily see the difference in the colour mode by simply applying the color mode to HSB:

colorMode(RGB)

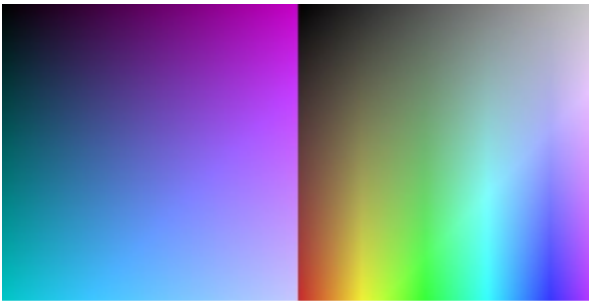


Figure 52: RGB (left) versus HSB (right) colour mode

As you can see, once you've set the colour mode, you can use colour exactly as before. The difference is that, instead of red-green-blue, we now have to think in terms of hue-saturation-brightness. These also have the range 0 (least) to 255 (most).

Hue

In the context of colour theory and visual representations, a **hue** refers to the basic, pure colour itself, independent of its brightness or intensity. It's essentially the "what" of the colour, like red, green, blue, yellow, etc.

Think of a rainbow. Each distinct colour band on the rainbow represents a different hue. From the red end to the violet end, you encounter a progression of hues along the spectrum.



Figure 53: Hue values

We can adapt our previous program to show the full range of hues

```
def setup():
    size(255, 200)
    colorMode(HSB)

    def draw():

        # Nested loops for each pixel

        for x in range(width):

            for y in range(height):

                # Calculate color based on coordinates

                h = x # hue based on x coord

                s = 255 # max saturation

                b = 255 # max brightness

                # Set stroke color

                stroke(color(h, s, b)) # HSB

                # Draw a point at the current coordinates

                point(x, y)
```

This program will fill the y-coordinates with gradually increasing hue value so that you get the entire range. Note that we have set the saturation and brightness to their maximum value.

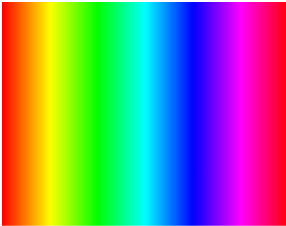


Figure 54: Full range of hues

Saturation

The saturation represents the strength of the colour. For example, if the hue is 255 (which is red) then how *much* red is the saturation.

0	50	100	150	200	250
0	50	100	150	200	250

Figure 55: Saturation levels

If we change our colour program around a bit by setting the hue to 255 (red) and making the saturation based on the x-coordinate:

`h = 255 # hue`

`s = x # saturation based on x coord`

We can see the effect of the saturation.



Figure 56: Range of saturation values

Brightness

Brightness is the measure of, well, how bright the colour is. Brightness 0 will always be black.



Figure 57: Brightness levels

If we switch around saturation and brightness so that brightness is based on the x-coordinate

$s = 255$ # max

$b = x$ # brightness based on x coord

We can see the effect of the increasing brightness

The question then is, should you use RGB or HSB? RGB is arguably easier to understand because it's roughly how printers and paint works – combine

colours to make other colours. HSB, however, lets you have more control over the colours as the colour itself is controlled by only one of the elements – hue – giving you another two to change the amount of colour (saturation) and the brightness.



Figure 58: Range of brightness levels

Colour Functions

Let's now look at a couple of built-in for manipulating colours - `blendColor()` and `lerpColor()`.

Blending Colour

Want to create new colours by combining existing ones? Look no further than the `blendColor()` function! It offers a simple yet powerful way to merge two colours into a brand new one.

Imagine blending red and green. What do you get? As you might expect, it's yellow! This function works like a mixing brush, combining colours based on a chosen blending "mode." Let's look at **ADD** first

ADD

Remember our bouncing ball animation? Let's bring that back!

It originally looked like this

```
x = 0

go_right = True # should ball go right (or left)

def setup():

    # Set window size, initialize variables, etc.

    size(400, 200)

    # Your other essential setup tasks here

def draw():

    background(220) # Set background colour

    global x # use the global variable

    # Update values, draw shapes, respond to user input, etc.

    fill(0) # Set filling colour for shapes

    circle(x, height/2, 50) # Draw an ellipse

    # send ball in a specific direction

    global go_right

    # check if ball hits the edge

    if x >= width: # if we hit the right side

        go_right = False

    elif x <= 0: # if we hit the left side

        go_right = True

    if go_right:
```

```
x+=1
```

```
else:
```

```
x-=1
```

Let's add to this a little so we have coloured backgrounds and a ball colour that combines colours.

First let's add some background colours. We can't use `background()` for this because that sets the entire background and we want to make it different colours, so we'll draw some squares. We are going to use global variables for the colours so that we can use them later, using the `color()` function which allows us to create variables for colours. So add some global variables for these:

```
left_colour = color(120, 60, 125)
```

```
right_colour = color(25, 125, 60)
```

And replace the `background(220)` line in the draw function:

```
fill(left_colour)
```

```
square(0, 0, 200)
```

```
fill(right_colour)
```

```
square(200, 0, 200)
```

Adding this into our sketch gives us a lovely, coloured background for our bouncing ball.



Figure 59: Coloured background using squares

All we need to do now is change the ball colour to be a combination of an original colour (I chose blue) and the background it is currently in:

```
global ball_colour
```

```
global left_colour
```

```
global right_colour
```

```
new_ball_colour = ball_colour
```

```
if x < 200:
```

```
new_ball_colour = blendColor(ball_colour, left_colour, ADD)
```

```
else:
```

```
new_ball_colour = blendColor(ball_colour, right_colour, ADD)
```

```
fill(new_ball_colour)
```

The ADD mode in blendColor() will add the current ball colour (a light purple colour) to the background colour.



Figure 60: Blended colour addition

The entire code is below:

```
x = 0

go_right = True # should ball go right (or left)

left_colour = color(120, 60, 125)

right_colour = color(25, 125, 60)

ball_colour = color(120, 120, 255)

def setup():

    size(400, 200)

    noStroke()

    def draw():

        global ball_colour

        global left_colour

        global right_colour

        global x

        fill(left_colour)

        square(0, 0, 200)

        fill(right_colour)

        square(200, 0, 200)

        new_ball_colour = ball_colour

        if x < 200:
```

```
new_ball_colour = blendColor(ball_colour, left_colour, ADD)
```

```
else:
```

```
new_ball_colour = blendColor(ball_colour, right_colour, ADD)
```

```
fill(new_ball_colour)
```

```
circle(x, height/2, 50)
```

```
# send ball in a specific direction
```

```
global go_right
```

```
# check if ball hits the edge
```

```
if x >= width: # if we hit the right side
```

```
go_right = False
```

```
elif x <= 0: # if we hit the left side
```

```
go_right = True
```

```
if go_right:
```

```
x+=1
```

```
else:
```

```
x-=1
```

It is now very easy to investigate the effects of the other blending modes but just changing ADD to something else

SUBTRACT

SUBTRACT, as you might expect, subtracts one colour from another.

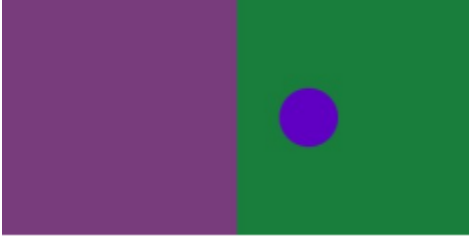


Figure 61: Blended colour subtraction

MULTIPLY

MULTIPLY, again, as you'd expect, multiplies the colours. This will also result in a darker colour.

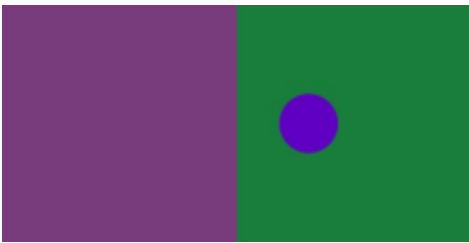


Figure 62: Blended colour multiplication

There are many other modes that you might like to play with. We investigate all of these when we look at image manipulation.

- DARKEST – darkest colour is the one displayed
- LIGHTEST – lightest colour is the one displayed
- DIFFERENCE – SUBTRACT colours from the colour underneath
- EXCLUSION – similar to DIFFERENCE but with less effect
- SCREEN – MULTIPLY with inverse colours
- OVERLAY – MULTIPLY dark colours and SCREEN light colours
- HARD_LIGHT – SCREEN when more than 50% grey, else

MULTIPLY

- SOFT_LIGHT – Like OVERLAY but with less effect
- DODGE – Lightens light colours only
- BURN – Darkens dark colours only

Linear Interpolation

Linear interpolation (aka lerp) is to find a colour between two other colours. To use this you specify the amount of interpolation which means which side to favour and by how much, from 0.0 (most left colour) to 1.0 (most right colour) with 0.5 being the mid-way point.

For example a colour mid-way between black and white is a mid-grey:

```
black = color(0)
```

```
white = color(255)
```

```
c = lerpColor(black, white, 0.5)
```

The colour mid-way between red and yellow is orange:

```
red = color(255, 0, 0)
```

```
yellow = color(255, 255, 0)
```

```
c = lerpColor(red, yellow, 0.5)
```

Changing the interpolation value in this example will either make it *more* red or *more* yellow.

We can try this easily with our bouncing ball code by changing `blendColor()` to `lerpColor()`:

```
if x < 200:
```



```
new_ball_colour = lerpColor(ball_colour, left_colour, 0.5)
```

```
else:
```

```
new_ball_colour = lerpColor(ball_colour, right_colour, 0.5)
```

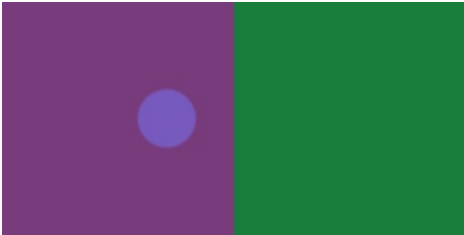


Figure 63: Linear interpolation of colours

CHAPTER 9: LINES AND SHAPES

Lines

Drawing a line may seem like a basic task, and with Processing, it is! But behind the scenes, there's a hidden world of complexity at play. Let's peel back the curtain and explore how lines are actually drawn on digital displays.

Remember those tiny squares called pixels? Every image on your screen is built from them, and they're the smallest unit we can directly control. This works perfectly for lines parallel to the X and Y axes – think vertical and horizontal lines. But what about diagonals?

Here's the challenge: pixels are square, not diagonal. So, how do we represent a line with any other angle? Imagine drawing a diagonal line across a grid of squares. Choosing which pixels to light up to best represent the intended line becomes a tricky calculation.

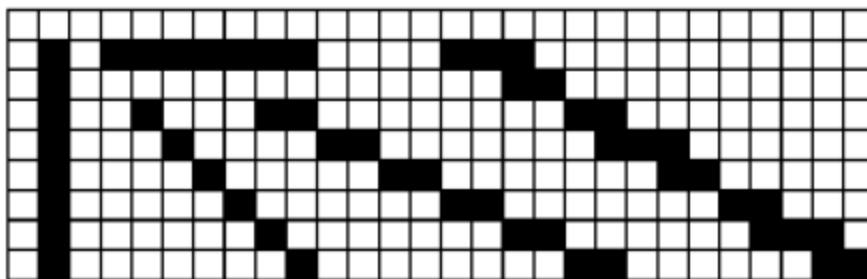


Figure 64: Lines drawn over pixels at different angles

Thankfully, Processing comes to the rescue!

The line() Function

Lines are specified by two sets of coordinates; the start and end points). Processing then works its behind-the-scenes magic to represent the line accurately so that you don't need to worry about how it does it:

```
line(x1, y1, x2, y2)
```

Where x1, y1 are the start coordinates and x2, y2 are the end coordinates. But don't let its simplicity fool you. Under the hood, line() does some clever calculations to ensure your line looks smooth, regardless of its angle. Need a diagonal line? No problem! Just tell line() your starting and ending points and it takes care of the rest.

For example the following lines will produce the lightning bolt shape:

```
line(25,25,12,50)
```

```
line(12,50,32,55)
```

```
line(32,55,25,75)
```

```
line(25,75,50,50)
```

```
line(50,50,25,42)
```

```
line(25,42,25,25)
```



Figure 65: Lightning bolt made of lines

Line Colour

We have already seen how to change line colour using the `stroke()` function:

```
stroke(255, 255, 20) # yellow
```

```
line(25,25,12,50)
```

```
line(12,50,32,55)
```

```
line(32,55,25,75)
```

```
line(25,75,50,50)
```

```
line(50,50,25,42)
```

```
line(25,42,25,25)
```



Figure 66: Yellow lines

Line Thickness

Line thickness can be controlled using the `strokeWeight()` function which sets the ‘weight’ of the line. This can be any value, including decimal values such as 0.5. If you don’t set it then the default value of 1 is used.

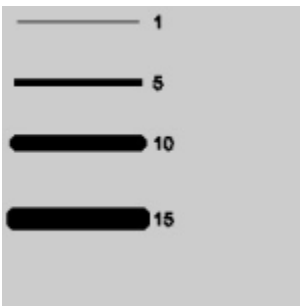


Figure 67: Line weights

As with `stroke()`, the weight is applied until it is changed:

```
strokeWeight(5)
```

```
line(10, 50, 90, 50)
```

Line Shape

The end shape of lines can also be changed using the `strokeCap()` function, for example:

```
strokeCap(SQUARE)
```

The options are:

- ROUND – rounded ends (the default)
- SQUARE – square ends
- PROJECT – ends that project beyond the end of the line

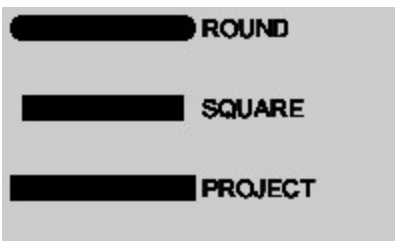


Figure 68: Line ends

Shapes

We have already used some of the shapes that are available, but let's look at some additional features and some new shapes!

Rectangles

You can create rectangles using the `rect()` function. Similar to creating squares, we need to specify the coordinates and the lengths of the sides:

```
rect(x, y, a, b)
```

Here `a` is the horizontal side and `b` is the vertical side. For example:

```
rect(20, 20, 100, 60)
```

Will draw a rectangle that is longer than it is tall.



Figure 69: Wide rectangle

Whereas:

```
rect(20, 20, 60, 100)
```

Will draw a rectangle that is taller than it is long.



Figure 70: Tall rectangle

What happens if we make the rectangle have the same length sides? Is that a square? Or a rectangle? Yes! It's both. A square is just an equilateral rectangle (ie a rectangle with equal sides).

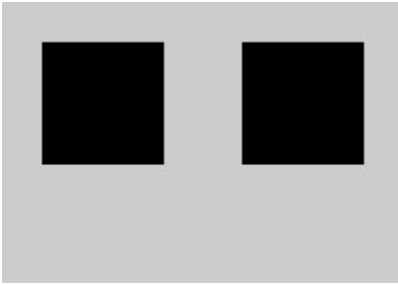


Figure 71: A square and an equilateral rectangle

So, why have a `square()` function if we can just use `rectangle()`?

```
rect(20, 20, 60, 60)
```

```
square(120, 20, 60)
```

Convenience, and readability. In reality, a square is just an equilateral rectangle. There is no need to write the length twice if they are always the same, which reduces typing errors, and it is clearer from the code that we mean a square.

Placing Rectangles

When we draw a rectangle in the middle of the screen, what happens?

```
rect(width//2, height//2, 60, 60)
```

If you try this, you will see that the rectangle is not in the middle! But why? We said to draw it at `width/2` and `height/2` which is the middle of the screen.

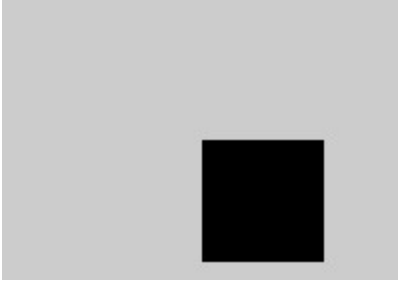


Figure 72: 'Centred' rectangle

If you look closely, you should see that the top-left corner of the rectangle is actually the middle of the display screen, not the centre of the rectangle. This is known as CORNER placement, which is the default method of placing it.

If you wanted to change this so that the coordinates you specify are the middle of the rectangle, you can use the CENTER placement by using the `rectMode()` function:

```
rectMode(CENTER)
```

We can see the difference this makes by drawing two identically sized rectangles using the same coordinates but different placements.

```
def setup():  
  size(200,140)  
  
  def draw():  
  
    fill(0, 75)  
  
    rectMode(CORNER)  
  
    rect(width//2, height//2, 60, 60)  
  
    fill(255, 0, 0, 75)
```

```
rectMode(CENTER)
```

```
rect(width//2, height//2, 60, 60)
```

As you can see, code for the rectangles are identical, it is the `rectMode()` that changed where they were placed.

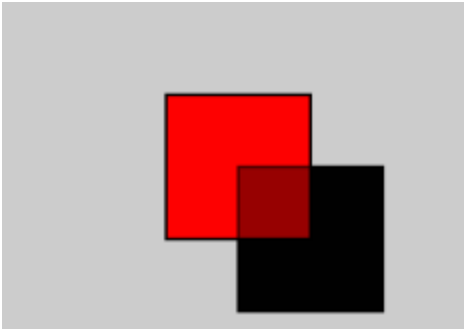


Figure 73: CENTER and CORNER rectangles

There are two other placement modes that complete the set:

- `CORNERS(x, y, a, b)` – this doesn't specify the size at all, but instead the coordinates of the corners. Here `x, y` are the top-left corner of the rectangle and `a, b` are the bottom-right corner
- `RADIUS(x, y, a, b)` – this is like `CENTER` except that `a` and `b` are the distance from the centre, rather than the overall size. This effectively means that the size will be double what you specify

Let's draw all four using the same values to see what happens:

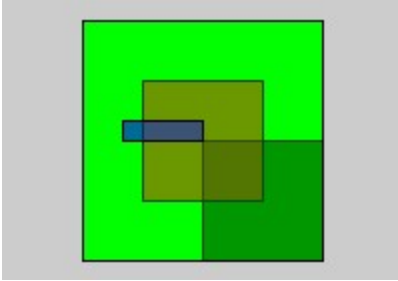


Figure 74: Different rectangle placements

```
def setup():
    size(200,140)

    def draw():

        fill(0, 75)

        rectMode(CORNER)

        rect(width//2, height//2, 60, 60)

        fill(255, 0, 0, 75)

        rectMode(CENTER)

        rect(width//2, height//2, 60, 60)

        fill(0, 255, 0, 75)

        rectMode(RADIUS)

        rect(width//2, height//2, 60, 60)

        fill(0, 0, 255, 75)

        rectMode(CORNERS)

        rect(width//2, height//2, 60, 60)
```

Squircles

Alright, there isn't really a squircle() function, but there is a way of drawing a rounded rectangle and it's really simple to do! We can provide an additional argument to the rect() function that defines the radius of the corner. There are 2 versions of this:

```
rect(x, y, a, b, r1, r2, r3, r4) # specify each corner
```

```
rect(x, y, a, b, r) # all corners the same
```

Try changing this on the code we just wrote and you should see that the larger number you give for the corners, the more rounded they become.



Figure 75: A squircle

Quadrilaterals

There's a lot we can do with rectangles and squares, but what if we wanted to draw some other four-sided shapes such as a rhombus or kite? Well, there is no specific function for these shapes, but there is the quad() function that lets you draw any four-sided shape, or quadrilateral.

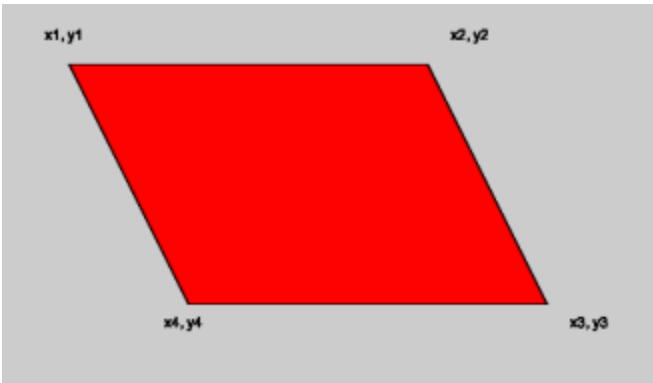


Figure 76: Quadrilateral coordinates

The `quad()` function is used by simply specifying the corners using coordinates:

```
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```

Of course you could still use this to draw squares and rectangles, but `square()` and `rect()` are just convenience functions for drawing common types of quadrilateral.

```
def setup():
```

```
  size(800,800)
```

```
  def draw():
```

```
    fill(255, 0, 0, 75)
```

```
    quad(100, 100, 400, 100, 500, 300, 200, 300) # parallelogram
```

```
    fill(255, 255, 0, 75)
```

```
    quad(400, 100, 600, 300, 400, 700, 200, 300) # kite
```

This code draws two quadrilateral shapes, a parallelogram and a kite.

There is no need for the shapes to be ‘proper’ named shapes though. You can supply any coordinates and have it draw any four-sided shape.

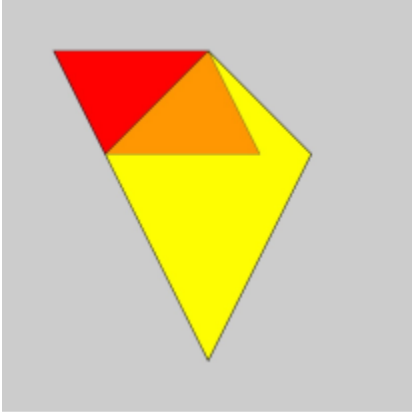


Figure 77: Parallelogram and kite quadrilaterals

Ellipses

Ellipses are ovals but, just as a square is an equilateral rectangle, a circle is an equilateral oval. They are created using the `ellipse()` function:

```
ellipse(x, y, a, b)
```

You can think of an ellipse as being an oval drawn inside a rectangle and so they otherwise work exactly the same as `rect()` but draws an oval instead:

```
def setup():
```

```
  size(400,200)
```

```
  def draw():
```

```
    rectMode(CORNER)
```

```
    fill(255) # white
```

```
    rect(50, 50, 300, 100)
```

```
ellipseMode(CORNER)

fill(255, 255, 0) # yellow

ellipse(50, 50, 300, 100)
```

Note the use of `ellipseMode()` instead of `rectMode()` but otherwise the code for the rectangle and the ellipse are essentially the same with the same coordinates.

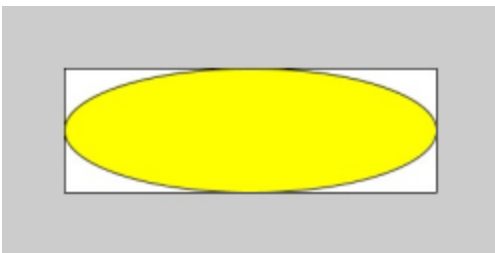


Figure 78: Ellipse in a bounding rectangle

The options for `ellipseMode()` are the same as `rectMode()`. To show this, we can just change our previous rectangle code and change `rect()` to `ellipse()`:

```
def setup():

size(200,140)

def draw():

fill(0, 75)

#rectMode(CORNER)

#rect(width//2, height//2, 60, 60)

ellipseMode(CORNER)

ellipse(width//2, height//2, 60, 60)

fill(255, 0, 0, 75)
```

```
# rectMode(CENTER)

# rect(width//2, height//2, 60, 60)

ellipseMode(CENTER)

ellipse(width//2, height//2, 60, 60)

fill(0, 255, 0, 75)

# rectMode(RADIUS)

# rect(width//2, height//2, 60, 60)

ellipseMode(RADIUS)

ellipse(width//2, height//2, 60, 60)

fill(0, 0, 255, 75)

#rectMode(CORNERS)

#rect(width//2, height//2, 60, 60)

ellipseMode(CORNERS)

ellipse(width//2, height//2, 60, 60)
```

Notice that we have left the previous rectangles in the code but turned them into comments, called commenting-out, so they do not run.

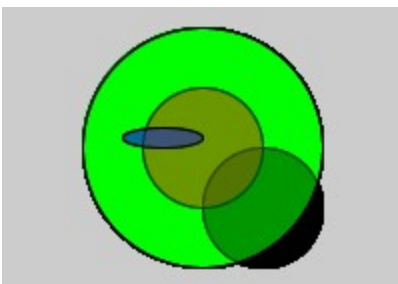


Figure 79: Ellipse placement

Of course it seems odd for an ellipse to have corners, but that's because the ellipse is basically drawn in what would be the rectangle.

If we combine the rectangles and ellipses, by 'commenting-in' the code again, we can see both at the same time and then the corners make a bit more sense.

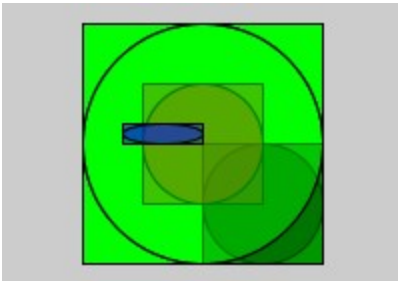


Figure 80: Ellipses and their bounding rectangles

Arcs

Arcs are potentially the most complicated of the pre-defined shapes. The `arc()` function helps you draw parts of an ellipse like a slice of pizza. Just like with ellipses, you provide coordinates (x, y) for the centre, along with two values (a, b) to define the ellipse's width and height.

The `arc()` function has extra arguments to control the slice itself. You can specify the starting and ending angles, defining how much of the ellipse's edge you want to draw. The angles are measured in radians measures angles in radians – like tiny slices of a mathematical pie. A half circle is pi (π) radians, so a full circle is 2π radians, and a quarter $\pi/2$:

`arc(x, y, a, b, start, end, mode)`

The coordinates x and, y , and the a and, b arguments are the same as for an ellipse. The others:

- start – the angle to start drawing the arc from
- end – the angle to stop drawing the arc, clockwise from the start
- mode – optional. One of OPEN, CHORD, PIE, or nothing. Not specifying the mode will use a combination of PIE and OPEN. We will look at the other modes later

But what is π ? Pi, represented by π , is a mathematical value used in calculation of angles. Luckily for us we don't really need to know the actual value most of the time because there are built-in variables for PI, TWO_PI, HALF_PI and QUARTER_PI. If you need some other value you can use mathematical operators with π such as $\text{PI}/8$.

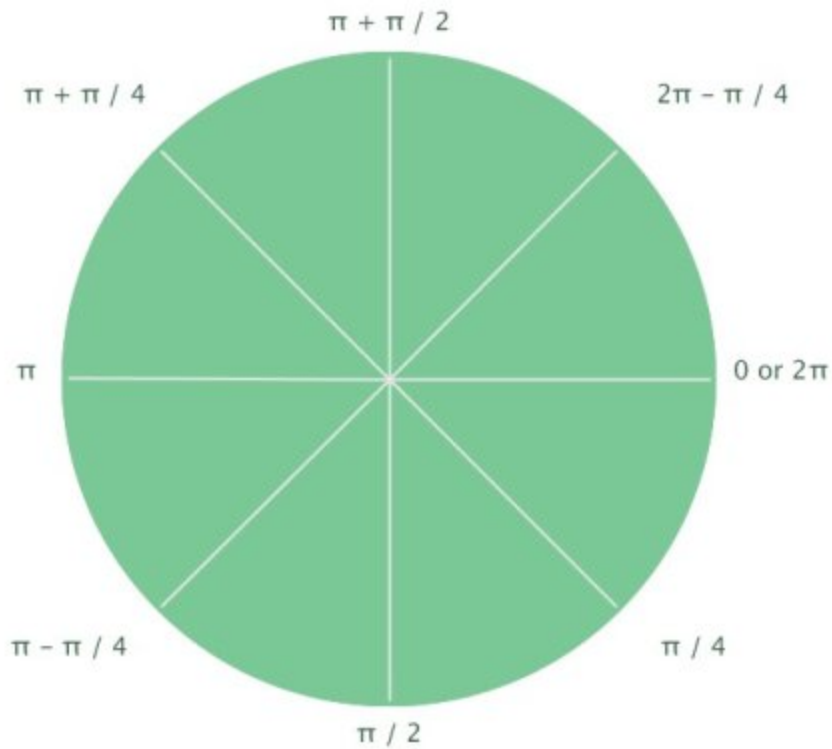


Figure 81: Radians of an ellipse

Think of an ellipse like a pizza, or a pie, with the slices represented by an amount of π . If we wanted the slice between 0 and $\pi/4$ we would specify these as the start and end of the arc

```
def setup():
    size(400, 400)

    def draw():

        fill(100, 170, 100)

        ellipseMode(CENTER)

        ellipse(width//2, height//2, 300, 300)
```

```
fill(250, 100, 100)
```

```
arc(width//2, height//2, 300, 300, 0, QUARTER_PI) # between 0 and  $\pi/4$ 
```

You can try changing these start and end points to see which parts of the pie you get. Of course, it doesn't just work with circles, it works for any shape ellipse.

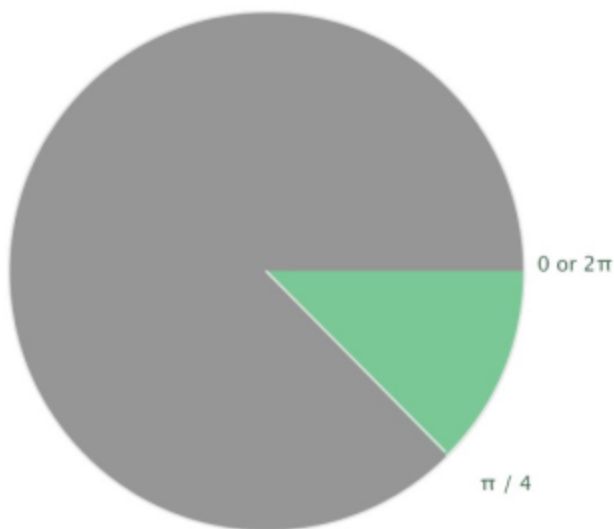


Figure 82: A small piece of pie

You can similarly use an arc from an oval.

```
def setup():
```

```
size(400, 400)
```

```
def draw():
```

```
fill(100, 170, 100)
```

```
ellipseMode(CENTER)
```

```

ellipse(width//2, height//2, 300, 150) # oval

fill(250, 100, 100)

arc(width//2, height//2, 300, 150, 0, QUARTER_PI) # between 0 and PI/4

# between PI + PI/4 and PI + PI/2

arc(width//2, height//2, 300, 150, PI+QUARTER_PI, PI+HALF_PI)

```

Here we are using an oval and drawing two arcs at different points.



Figure 83: Arcs of an oval

Arcs also use `ellipseMode()` to determine the coordinates points in the exact same way as ellipses do. Yes we know this is confusing!

Modes

The optional ‘mode’ determines how the arc is drawn. We have so far been using the default mode by not specifying it:

```
arc(x, y, a, b, start, end, mode)
```

We will illustrate these by drawing two ellipses, one smaller than half, and one larger:

```
def setup():
```

```

size(400, 200)

def draw():

fill(250, 100, 100)

arc(width//4, height//2, 150, 75, 0, QUARTER_PI)

fill(100, 100, 250)

arc(width//1.5, height//2, 150, 75, 0, PI + QUARTER_PI)

```

As you have seen, using the default mode is like cutting a piece, however large, out of a pie. Note, however, that there is no line drawn along the straight edge.

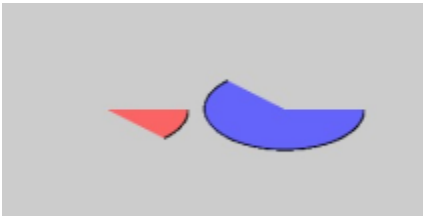


Figure 84: Ellipses in default mode

PIE

PIE mode is very similar to the default mode. The only difference is that it draws a border along the straight edge as well as the curved edges

```

def setup():

size(400, 200)

def draw():

fill(250, 100, 100)

arc(width//4, height//2, 150, 75, 0, QUARTER_PI, PIE)

```

```
fill(100, 100, 250)
```

```
arc(width//1.5, height//2, 150, 75, 0, PI + QUARTER_PI, PIE)
```

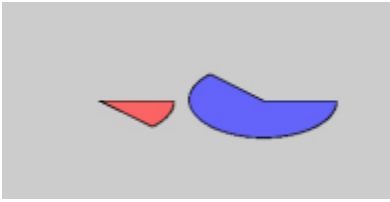


Figure 85: Arcs in PIE mode

OPEN

OPEN mode creates an arc that directly cuts from one point on the outer edge to the other. It creates a border around the curved edge but no border on the straight edges.

Changing PIE to OPEN mode to our ellipses results in some very different arcs.

You can imagine this like selecting to points on the edge of a pie and then cutting straight across it without going into the middle and back.

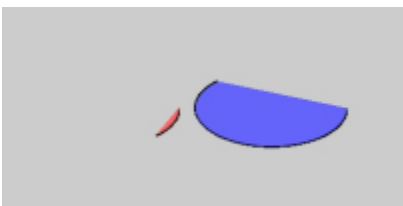


Figure 86: Arcs in OPEN mode

CHORD

CHORD mode is basically the same as OPEN mode but will draw the border on the straight edge as well. Change OPEN to CHORD in the code above and try it for yourself.

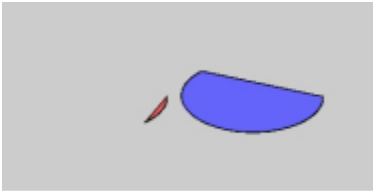


Figure 87: Arcs in CHORD mode

For a bit more interest, let's make a small animation using arcs.

```
n = 0

def setup():
    size(400, 400)

    def draw():
        global n
        fill(100, 100, 250)

        arc(width//2, height//2, 300, 300, 0, n)

    n += 0.01
```

Here we draw an arc using a variable as the end point. It will start at 0 and then gradually increase until it gets back around to 2π which is the same point as 0.



Figure 88: Animated arc

Triangles

The last built-in shape to look at are triangles which has a `triangle()` function. There is no `triangleMode()` function because you just tell it the coordinates of the three corners and it fills in the rest:

```
triangle(x1, y1, x2, y2, x3, y3)
```

This simplicity makes triangles very easy to understand as you just supply the coordinates. The order of the sets of x, y, coordinates don't matter because the lines are just drawn between them:

```
def setup():  
    size(400, 250)  
  
    def draw():  
        fill(60, 124, 200)  
  
        triangle(200, 200, 50, 50, 350, 200)
```

Drawing triangles doesn't really get any more complicated than this. The code draws a blue triangle.

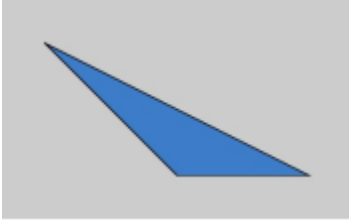


Figure 89: Blue triangle

Shapes Animation

Instead, let's do something more interesting with shapes.

```
x1 = 200
```

```
y1 = 50
```

```
x_direction = 1
```

```
y_direction = 1
```

```
x_change_speed = 1
```

```
y_change_speed = 1
```

```
def setup():
```

```
    size(400, 400)
```

```
    background(0)
```

```
def draw():
```

```
    global x1
```

```
    global y1
```

```
    global x_direction
```

```
    global y_direction
```

```
# change direction based on current x value
```

```
if x1 >= width:
```

```
x_direction = -x_change_speed
```

```
if x1 <= 0:
```

```
x_direction = x_change_speed
```

```
# change direction based on current y value
```

```
if y1 >= height:
```

```
y_direction = -y_change_speed
```

```
if y1 <= 0:
```

```
y_direction = y_change_speed
```

```
# change x and y coordinates
```

```
x1 += x_direction
```

```
y1 += y_direction
```

```
stroke(51, 255, 51)
```

```
fill(0)
```

```
triangle(200, 50, 300, 150, x1, y1)
```

The code draws a green triangle as changing coordinates which are contained within the size of the screen. As the draw() function repeats, it overlays triangles on top of other making some interesting patterns. Try changing some x_change_speed variable for different patterns.

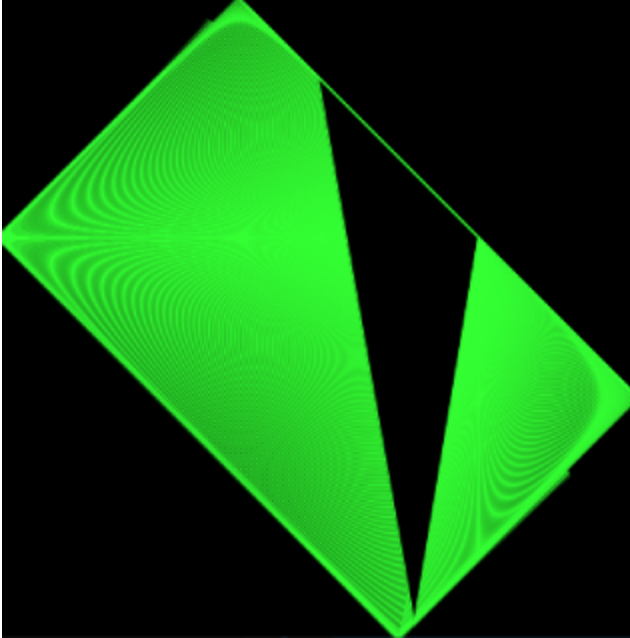


Figure 90: Overlaid triangles

CHAPTER 10: CUSTOM SHAPES

We've seen that it is easy to draw shapes using the built-in functions, but what if you want to draw a shape there isn't a function for? For that we need to define our own shapes using vertices (singular: vertex).

Vertices

A vertex is like a corner. Each vertex defines a point where the edge of the shape meets. We define these using the `vertex()` function by defining the corners (vertices) of the shape and Processing will fill in the lines to create a shape.

```
x = 250
```

```
y = 50
```

```
def setup():
```

```
  size(500, 350)
```

```
  def draw():
```

```
    strokeWeight(3)
```

```
    fill(255)
```

```
    global x
```

```
    global y
```

```
    beginShape()
```

```
    vertex(x, y) #1
```

```
    vertex(x + 200, y + 200) #2
```

```
vertex(x - 200, y + 200) #3
```

```
endShape()
```

It often makes sense to draw shapes from a defined point, rather than specify the exact coordinates of each vertex, as this can make it easier to visualise and means you can change the starting point, which we've called x and y, and everything else will change with it.

Note the use of `beginShape()` and `endShape()`. If you forget these, it just won't draw anything!

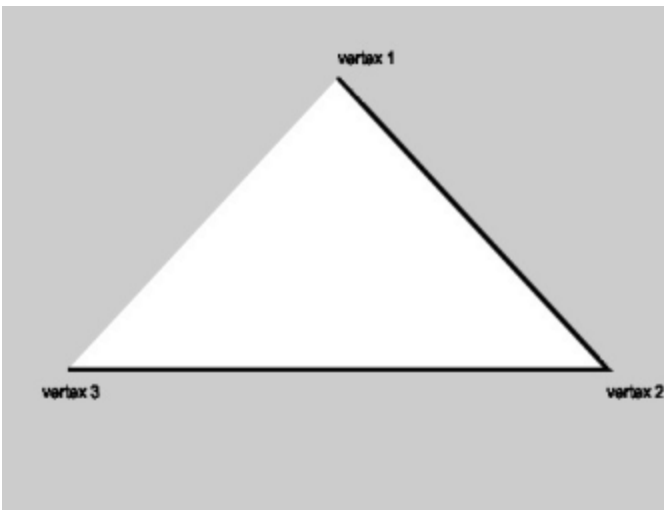


Figure 91: Custom triangle

You might also notice that one of the sides is missing its border. This is because we specified the three points and it just filled in the last line in. Really there are four vertices here because we also wanted a line from vertex 3 to vertex 1. We don't need to add another vertex though, we can just tell it to fill in the last line properly using `endShape(CLOSE)`.

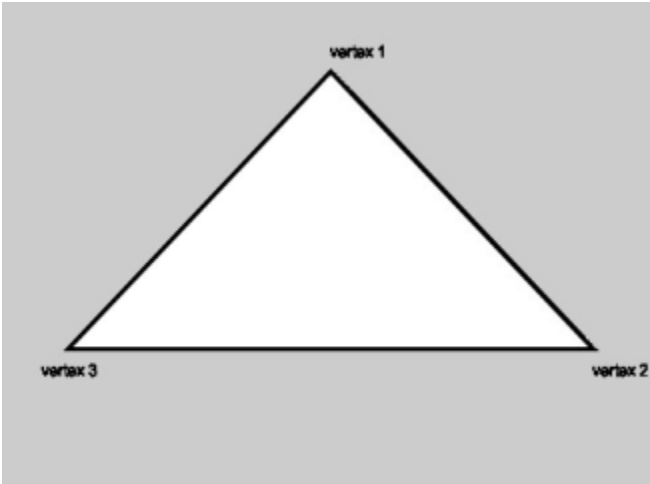


Figure 92: Closed custom triangle

```
beginShape()
```

```
vertex(x, y) #1
```

```
vertex(x + 200, y + 200) #2
```

```
vertex(x - 200, y + 200) #3
```

```
endShape(CLOSE) # close shape
```

This will effectively close the shape by adding the line from the last vertex (vertex 3) to the first (vertex 1).

Of course we could just use `triangle()` for this, so let's draw a more complex shape

Simple Custom Shapes

Let's take the triangle we just drew and add a fourth vertex to it:

```
beginShape()
```

```
vertex(x, y) #1
```

```
vertex(x + 200, y + 200) #2
```



```
vertex(x, y + 150) #3
```

```
vertex(x - 200, y + 200) #4
```

```
endShape(CLOSE)
```

We've added a vertex between vertex 2 and 3 to create a four-sided shape.

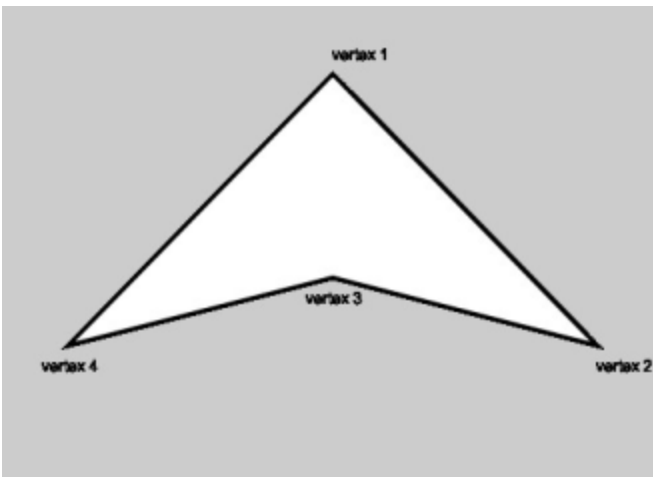


Figure 93: Custom four-sided shape

Again, we could have done this using the `quad()` function. Adding more vertices will enable you to create more complex shapes that are not possible using the built-in shapes such as `triangle()` and `quad()`:

```
beginShape()
```

```
vertex(x, y) #1
```

```
vertex(x + 50, y + 100) #2
```

```
vertex(x + 200, y + 200) #3
```

```
vertex(x, y + 150) #4
```

```
vertex(x - 200, y + 200) #5
```

```
vertex(x - 50, y + 100) #6
```

```
endShape(CLOSE)
```

The shape now has six sides, which you can draw anywhere on the screen simply by changing the x and y variables.

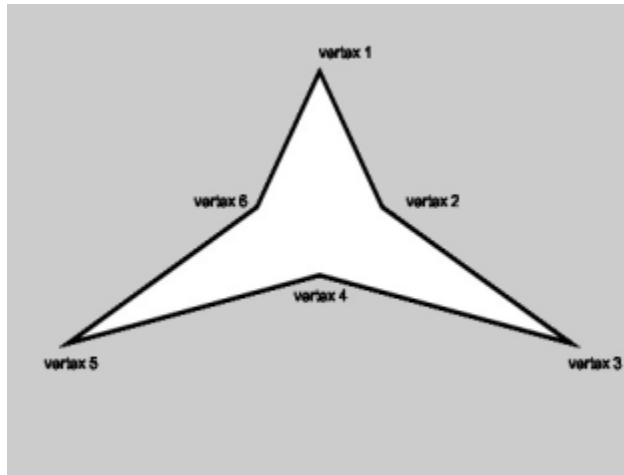


Figure 94: Custom six-sided shape

Custom Regular Shapes

A regular shape is one that has equal sides and angles, such as an equilateral triangle or a square, but also a hexagon or a decagon, for example, for which there are no built-in functions

Let's create our own function to draw a hexagon. With a bit of maths we can work out the angles for the vertices and use this to draw our shape:

```
def draw_hexagon(x, y, radius):
```

```
beginShape()
```

```
for i in range(6):# 6 sides
```

```
vertex(x + radius*cos(2.0*PI*i/6), y + radius*sin(2.0*PI*i/6))
```

```
endShape(CLOSE)
```

This function allows us to specify the starting x and y coordinates of the hexagon and also the size using the radius argument. The `cos()` and `sin()` functions are built-in functions for working with angles using trigonometry.

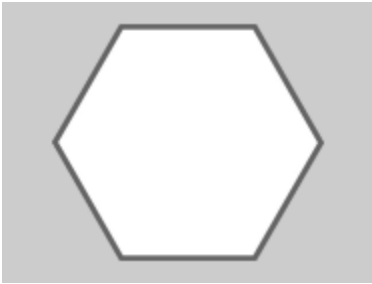


Figure 95: Custom hexagon

If we plug this into a programme we can draw ourselves a hexagon!

```
def setup():  
  
    size(640, 480)  
  
    strokeWeight(9)  
  
    stroke(100)  
  
    def draw():  
  
        draw_hexagon(width/2, height/2, height/2)  
  
    def draw_hexagon(x, y, radius):  
  
        beginShape()  
  
        for i in range(6):# 6 sides  
  
            vertex(x + radius*cos(2.0*PI*i/6), y + radius*sin(2.0*PI*i/6))  
  
        endShape(CLOSE)
```

Do you notice anything about this code? We repeatedly use the number 6? The same number as the number of sides in a hexagon. We could extend this function to draw any regular shape!

```
def draw_regular_shape(x, y, radius, sides):  
  
    beginShape()  
  
    for i in range(sides):# sides  
  
        vertex(x + radius*cos(2.0*PI*i/sides), y + radius*sin(2.0*PI*i/sides))  
  
    endShape(CLOSE)
```

All we have done here is create a new function called `draw_regular_shape` with an additional argument called *sides* for the number of sides. Then replaced all the number 6s with the *sides* argument. We can now draw regular shapes with any number of sides! Try it!

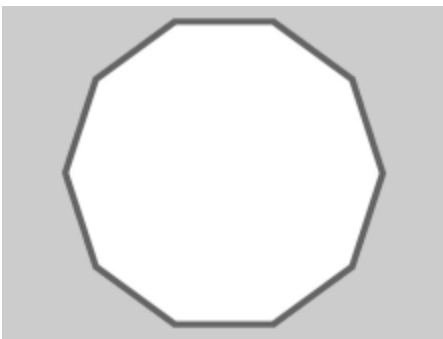


Figure 96: Custom regular shape (decagon)

Let's use this for something more interesting like creating a beehive using hexagons. We already have our code for drawing regular shapes, which includes hexagons, so we just need to draw a lot of them!

We could do this in a loop, which would draw them all at once, or we could rely on the repeated draw() function, either would work. Let's use a loop first.



Figure 97: Hexagons in a partially drawn beehive

```
x = 0

y = 25

def setup():

    size(640, 480)

    strokeWeight(4)

    def draw():

        global x

        global y

        while x < width: # only if on screen

            draw_regular_shape(x, y, 25, 6) # draw hexagon

            x += 45
```

This loop will draw a hexagon using the x and y coordinates then increases the x coordinate so they are drawn across the screen.

Not quite right though is it? We want to draw an extra row underneath but have it interlocking, so let's add some code to do that next

```
up = True

x = 0

y = 25

def setup():

    size(640, 480)

    strokeWeight(4)

    def draw():

        global x

        global y

        global up

        while x < width: # only if on screen

            if up: # draw up and down

                y -= 25

            else:

                y += 25

            draw_regular_shape(x, y, 25, 6) # draw hexagon

            x += 45

            up = not up # switch up and down

    def draw_regular_shape(x, y, radius, sides):

        beginShape()
```

```

for i in range(sides):# sides

vertex(x + radius*cos(2.0*PI*i/sides), y + radius*sin(2.0*PI*i/sides))

endShape(CLOSE)

```

This code now understands that we want to interlock the hexagons by switching between drawing them up a row and down a row. Each time it loops, the y coordinate is changed using the boolean *up* variable which is switched each time using *not up* which switches the boolean to the opposite value.

Note that we have used x and y as both local and global variables. Remember that variables have *scope*. When we create a local variable with the same name as a global variable, the function has a variable with that name, say x, in local scope, so it will use that one.

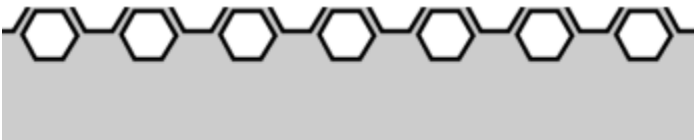


Figure 98: Interlocking hexagons

This is starting to look more like a beehive. We just need to fill the rest of the screen and colour it in!

```

up = True

x = 0

y = 25

def setup():

size(640, 480)

```

```
strokeWeight(4)

fill(238,186,25)

background(79,51,27)

stroke(237,140,0)

def draw():

    global x

    global y

    global up

    while x < width: # only if on screen

        if up: # draw up and down

            y -= 25

        else:

            y += 25

        draw_regular_shape(x, y, 25, 6) # draw hexagon

        x += 45

        up = not up # switch up and down

    else: # reset x and y

        x = 0

        y += 25

    def draw_regular_shape(x, y, radius, sides):

        beginShape()

        for i in range(sides):# sides
```



```
vertex(x + radius*cos(2.0*PI*i/sides), y + radius*sin(2.0*PI*i/sides))
```

```
endShape(CLOSE)
```

This is the entire code needed to draw our beehive. We have added an `else:` statement that resets the `x` and `y` coordinates each time a row is drawn. This will start on the left again a bit further down. We've also added some beehive colours.

It now really looks like a beehive! Now try changing *while* to *if*. What happens?

You might expect it to draw only one hexagon because there is no loop. Except the `draw()` function repeats, so each time it draws one hexagon. This gives an interesting effect of adding the hexagons one-by-one.

So why have loops at all? If the `draw()` function is repeated, each time it runs it will do whatever code you have written in there. If this is a loop it will do everything at once. It all depends on the effect you want.

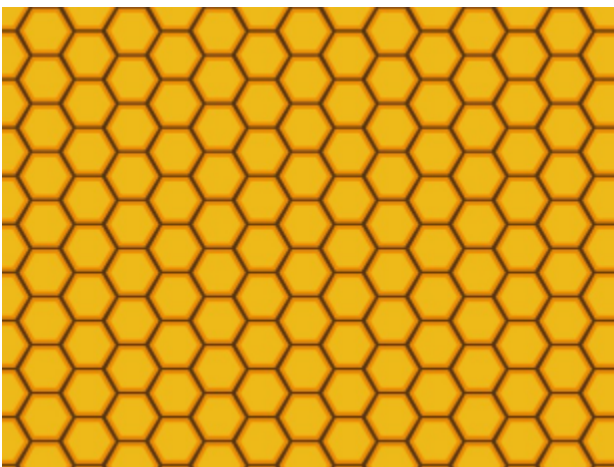


Figure 99: Hexagons in a beehive

Complex Shapes

Let's use both a loop and the draw() function to draw a something more complex. For this we are going make use of a function we haven't looked at before; random().

The random() functions gives you a random floating point number by specifying the boundary of the number you want. For example:

```
random(255)
```

Will give you a number between 0 and 255. Whereas:

```
random(100, 200)
```

Will give you a random number between 100 and 200.

```
def setup():
```

```
size(400, 400)
```

```
background(0)
```

```
def draw():
```

```
fill(random(255), random(255), random(255)) # random colour
```

```
x = width//2 # reset x
```

```
y = height//2 # reset y
```

```
sides = random(2, 7) # random number of sides
```

```
beginShape()
```

```
for i in range(int(sides)):
```

```
x += random(-50, 50) # random x
```

```
y -= random(-50, 50) # random y
```

```
vertex(x, y)
```

```
endShape(CLOSE)
```

This code draws random shapes with random colours. Let's walk through the code:

```
fill(random(255), random(255), random(255)) # random colour
```

This line fills the shapes with a random colour. We've used the `random()` function three times here, inside the `fill()` function. Remember that the `fill` function uses three colours for red, green blue, for example:

```
fill(255, 145, 32)
```

We have simply replaced each number with `random(255)` so that each becomes a random number. As this is inside the `draw()` function it will fill with a random colour each time.

```
sides = random(2, 7) # random number of sides
```

This line sets a variable called *sides* as a random number between 2 and 7. Each time the `draw()` function runs it will generate a new random number.

```
beginShape()
```

```
for i in range(int(sides)):
```

```
    x += random(-50, 50) # random x
```

```
    y -= random(-50, 50) # random y
```

```
    vertex(x, y)
```

```
endShape(CLOSE)
```

This code creates the shape. Notice that we have put a for loop between `beginShape()` and `endShape(CLOSE)`. This is fine and means that the loop will generate random numbers for x and y and then draw a vertex at those coordinates, and will do this the number of times we have told the loop to run, which is the random number of sides we just generated.

That means that each time the `draw()` function runs, it will generate new random colours, number of sides and then draw a shape. Every time you run it the shapes will be different!



Figure 100: Random shapes

SECTION 2: ADVANCED CONCEPTS

CHAPTER 11: OBJECT-ORIENTED PROGRAMMING

So far, we have been writing our code in a style that is known as *procedural programming*. Procedural programming is a way of writing code that is like a set of instructions that, although may be split-up using functions, can be read as a series of steps.

Object-Oriented Programming (also called OOP) instead uses the idea that programs contain ‘objects’ which are essentially ‘things’ that do a specific task or a set of related tasks and that other parts of the program don’t need to know how each object performs these tasks. For example, you might have some code that draws a complex shape such as a frog and, instead of putting that as a function in your program, you have some separate code that allows you to create frogs that know how to draw themselves. A real-life example might be that you employ a mechanic to fix your car. You don’t need to know *how* they fixed it, just that they did when you asked them to.



Figure 101: Using an object

The advantage of OOP is that you create code that can be shared by other programs and you, as the programmer, don’t need to know how to draw frogs if all you want is to use some code that does it.

The disadvantage of OOP is that it is more difficult to understand and involves writing more code the first time.

When we write the object-oriented code, we specify how the objects will work as *classes*.

Classes

Classes are the way of defining what objects will do. You could think of them like a template that you can create objects from. So if we had a template for drawing a frog, we could draw as many frogs as we want, and each frog will be an object that came from the template (class).

Let's start creating a class for our frogs. A class is declared like this:

```
class Frog:
```

This tells your programme that it can create Frog objects. They are usually named as a singular noun starting with a capital letter. Of course it doesn't do anything at the moment and, actually, Python will not let us create a class that does nothing.

If you are intentionally creating a class that does nothing because, for example, you will complete it later you can add the `pass` command:

```
class Frog:
```

```
    pass
```

This also works with functions that currently have no contents. You only need this if you are not intending to complete the rest of the class yet. But let's make it do something by adding a function to it. Technically functions inside classes are called methods but they are often used interchangeably.

Methods

Methods are named like functions, usually a verb that describes what it will do. We are going to create a method that allows our Frog class to know how to display itself, so we'll just call it *display*.

You can declare a method like this:

```
def display(self):
```

This mostly looks like a function that you've seen before apart from the *self* part. In Python *self* refers to the object. This is because each object is a copy of the class, and each class can have more than one object. So *self* refers to 'the current object that is being used while the programme is running',

Here is our whole Frog class so far:

```
class Frog:
```

```
# displays the frog
```

```
def display(self):
```

```
x = width//2
```

```
y = height//2
```

```
strokeWeight(2)
```

```
fill(0, 128, 0) # colour
```

```
# body
```

```
ellipse(x, y, 90, 45)
```

```
# feet
```

```
fill(107, 142, 35)
```

```
ellipse(x - 30, y + 15, 45, 23)
```

```
ellipse(x + 30, y + 15, 45, 23)
```

```
# Eyes
```

```
fill(255, 255, 255)
```

```
ellipse(x - 15, y - 23, 23, 30)
```

```
ellipse(x + 15, y - 23, 23, 30)
```

```
fill(0, 0, 0)
```

```
ellipse(x - 15, y - 23, 8, 8)
```

```
ellipse(x + 15, y - 23, 8, 8)
```

```
# Mouth
```

```
arc(x, y - 2, 30, 5, 0, PI)
```

This code declares that there is a class called Frog, and that each Frog object has a display() method that draws a frog on the screen. If you copy this into Processing though, you will find it doesn't do anything yet, and that is because we didn't create any objects, only defined the class which objects will be a copy of. So, we have the template, but no actual frogs.

Using Objects

We create objects from our class a bit like creating a value for a variable:

```
<variable name> = <class_name>()
```

For example:

```
froggy = Frog()
```

Here we have created a variable called *froggy* which has the value of a Frog object. Once we have an object, we can tell it to do things. In this case, we can tell it to display itself:

```
def setup():  
  
    size(300,300)  
  
    background(255)  
  
    def draw():  
  
        froggy = Frog()  
  
        froggy.display()
```

This code creates the Frog object then tells that object to display itself. Running this will now draw a frog in the middle of the screen.



Figure 102: A Frog object

At this point you might be wondering what the point of the class and objects are when you could just this code inside the draw() function. And you'd be right. The power of using objects comes from each one independently functioning from the others. So, let's change our code a bit.

We are going to add a special type of method called an init, or initialisation, method. This method's job is to run automatically when an object is created and we can use it to define something the object should know about itself. In this example, its coordinates:

```
def __init__(self, x, y):
```

```
self.x = x
```

```
self.y = y
```

This initialisation method, sometimes called a constructor, must be named `__init__` (with 2 underscores each side) and it must be given the *self* argument. You can then optionally add any other arguments you want to include. In this example we are expecting to use the x and y coordinates. This means that, when we create a Frog object, it **must** now be given 2 argument values for x and y

```
froggy = Frog(width//2, height//2)
```

```
froggy.display()
```

Note that we don't include the 'self' when calling it. The values, in this example `width//2` and `height//2` are passed to the `init` method and stored as variables `x` and `y`. The notation:

```
self.x = x
```

Means that `x` represents the value given to the method (in this case `width//2`) and then stored in a variable that the object owns, known as an instance variable, also called `x`. This means that `self.x` now holds the value `width//2` and we can use this in the rest of the class code:

```
class Frog:
```

```
    def __init__(self, x, y):
```

```
        self.x = x
```

```
        self.y = y
```

```
    # displays the frog
```

```
def display(self):

strokeWeight(2)

fill(0, 128, 0) # colour

# body

ellipse(self.x, self.y, 90, 45)

# feet

fill(107, 142, 35)

ellipse(self.x - 30, self.y + 15, 45, 23)

ellipse(self.x + 30, self.y + 15, 45, 23)

# Eyes.

fill(255, 255, 255)

ellipse(self.x - 15, self.y - 23, 23, 30)

ellipse(self.x + 15, self.y - 23, 23, 30)

fill(0, 0, 0)

ellipse(self.x - 15, self.y - 23, 8, 8)

ellipse(self.x + 15, self.y - 23, 8, 8)

# Mouth.

arc(self.x, self.y - 2, 30, 5, 0, PI)
```

Notice that we have changed all of the x and y coordinates to be self.x and self.y so that it uses those variables. This means that each frog object has its own self.x and self.y variables, which can be different from the other frogs.

This now means we can create multiple frogs:

```
def draw():  
  
    froggy = Frog(width//2, height//2)  
  
    froggy2 = Frog(100, 100)  
  
    froggy.display()  
  
    froggy2.display()
```

Here we create two Frog objects with the coordinates we want them to be displayed at. When we later use the `display()` method each Frog *knows* where it should display itself.

We can also change these later if we want to by, for example:

```
froggy.y = 250
```

This will permanently change the `y` value of the Frog object we have called *froggy* so that the next time it is displayed the value of the `y` coordinate will have changed. This also means we can find out the current values if needed. For example:

```
print(froggy.x)
```

will print the current `x` coordinate value of the *froggy* object.



Figure 103: Two frog objects

Inheritance

Imagine now that we want to create a new type of frog, one that shares all the same attributes of our Frog class, but adds to it. This type of frog will

look the same but be able to fly, we will call them hover frogs.

This could mean creating a new class, called HoverFrog, and copying all the Frog code into it, then adding our additional code to display them differently. This seems like a waste of effort and introduces a problem in terms of ongoing maintenance of your code - you now have two copies of the same code that, if you wanted to change, you would have to change twice! Fortunately, Python has a mechanism to handle this known as *inheritance*.

When we declare a class based on another, we call the new class a *subclass* and the original one its *superclass*. For Example:

```
class HoverFrog(Frog):
```

Declares a new class called HoverFrog as being a subclass of Frog. It will *inherit* all the features of the superclass, including the variables and the methods. So if all we do is

```
class HoverFrog(Frog):
```

```
pass
```

We now have an exact copy of Frog, called HoverFrog, which is a bit pointless, so let's have it do something else.

Inherited Methods

The usual case for creating a subclass is because you want a class that is *quite* like the superclass but does something extra, or differently. In this case we want to draw hover frogs a bit differently to frogs:

```

class HoverFrog(Frog):

    def display(self):

        # rotor blades

        line(self.x, self.y - 22, self.x, self.y - 50)

        line(self.x - 30, self.y - 63, self.x + 30, self.y - 35)

        line(self.x - 30, self.y - 35, self.x + 30, self.y - 63)

```

Firstly, notice that we didn't need to define the `__init__` method or that `self.x` and `self.y` exist. This is because they are inherited from the superclass `Frog`. If you create an object of `HoverFrog`:

```

def draw():

    froggy = Frog(width//2, height//2)

    froggy.display()

    hover = HoverFrog(100, 100)

    hover.display()

```

and run this though you will notice something else. We only get rotor blades and no frog! Didn't I say that methods would be inherited? Yes, I did.



Figure 104: Partially completed HoverFrog subclass

This is because we created a method called `display()` in `HoverFrog` which tells `HoverFrog` objects to use that `display()` method instead of the one in the superclass. This is known as *overriding* – as in override the inherited method and use this one instead.

As both classes now have different `display()` methods it will use the relevant one depending on which class our object came from (frog or `HoverFrog`).

We have three options here:

1. Call the method something else. This has some inherent problems because `display()` is a good name for what the method does and renaming it means we have two methods, one inherited from `Frog` and the new one
2. Copy the code from `Frog` to `HoverFrog` so it also draws a frog. This would work but means we have completely ignored the benefits of inheriting the method and now we have two methods in different classes that do the same thing. This will be an issue if we need to change it because now we have two places to remember to change
3. Use the inherited method *as well* as the new method even though they have the same name

Option 3 is the best course of action because it utilises the benefits of inheritance, and means we don't need to remember that `Frog` and `HoverFrog` have different methods for drawing themselves.

We can refer to a superclass using its name. For example:

```
Frog.display(self)
```

Will use the `display` method from `Frog`. So we can add this in to our `HoverFrog` code:

```
class HoverFrog(Frog):
```

```
def display(self):  
  
    # rotor blades  
  
    line(self.x, self.y - 22, self.x, self.y - 50)  
  
    line(self.x - 30, self.y - 63, self.x + 30, self.y - 35)  
  
    line(self.x - 30, self.y - 35, self.x + 30, self.y - 63)  
  
    # frog  
  
    Frog.display(self)
```

If you re-run your code, you will see that the frog, and rotor blades, are now drawn even though we didn't add the frog-drawing code to the HoverFrog class.



Figure 105: Frog and HoverFrog

Extending a Subclass

There is no reason a subclass can't have additional methods though, that have not been inherited from the superclass at all. What if we also wanted to tell hover frogs to fly across the screen? Let's add a fly() method to the HoverFrog class:

```
def fly(self):  
  
    self.x += self.x/20
```

This increases the x-coordinate of the object, making it appear as though it is flying across the screen. Having a method like this, that changes the variables of an object, is known as encapsulation: allowing an object to

change itself but having to use the method to do so. This keeps the code isolated from the rest and allows objects to act on their own values.

We just then need to call this method:

```
hover_frog.fly() # fly!
```

However, this will only work for HoverFrog objects as this is the only class we have defined it in.

Here's all our code so far:

```
hover_frog = None # declare the variable but set it to be nothing for now
```

```
def setup():
```

```
    global hover_frog
```

```
    size(300,300)
```

```
    background(255)
```

```
    hover_frog = HoverFrog(100, 100) # create the hover frog object
```

```
def draw():
```

```
    global hover_frog
```

```
    background(255)
```

```
    hover_frog.display() # display the frog
```

```
    hover_frog.fly() # fly!
```

```
class Frog:
```

```
    def __init__(self, x, y):
```

```
        self.x = x
```

```
self.y = y

# displays the frog

def display(self):

    strokeWeight(2)

    fill(0, 128, 0) # colour

    # body

    ellipse(self.x, self.y, 90, 45)

    # feet

    fill(107, 142, 35)

    ellipse(self.x - 30, self.y + 15, 45, 23)

    ellipse(self.x + 30, self.y + 15, 45, 23)

    # Eyes.

    fill(255, 255, 255)

    ellipse(self.x - 15, self.y - 23, 23, 30)

    ellipse(self.x + 15, self.y - 23, 23, 30)

    fill(0, 0, 0)

    ellipse(self.x - 15, self.y - 23, 8, 8)

    ellipse(self.x + 15, self.y - 23, 8, 8)

    # Mouth.

    arc(self.x, self.y - 2, 30, 5, 0, PI)

class HoverFrog(Frog):

    def display(self):
```

```

# rotor blades

line(self.x, self.y - 22, self.x, self.y - 50)

line(self.x - 30, self.y - 63, self.x + 30, self.y - 35)

line(self.x - 30, self.y - 35, self.x + 30, self.y - 63)

# frog

Frog.display(self)

def fly(self):

    self.x += self.x/20

```

The HoverFrog object, which we called `hover_frog`, is declared in the `setup()` stage of the program, but then used in the `draw()` function, so we need this to be accessible in both functions. You will find you can't declare a global variable of type `HoverFrog` so, as a workaround, we have initialised it to type `None` and then changed it in the `setup()` function to create our `HoverFrog` object.

`None`, as you would expect, means nothing (ie no value), but can be replaced with something else later. The reason we have to do this is because we haven't created an object yet to initialise the variable with.

You will see that, because the object is created at `setup()` time, the object remembers its `x` and `y` coordinates so when we use the `fly()` method, it will advance across the screen.



Figure 106: A flying HoverFrog

Lists of Objects

For fun, let's create loads of flying hover frogs! Rather than create lots of variables, one for each frog, we can use something we've looked at already – a list. So we only need the one variable to hold our frogs:

```
frogs = [] # list of frogs
```

We can then populate this list in our `setup()` function using a loop:

```
for i in range(10):  
  
    hover_frog = HoverFrog(random(0, 200), random(0, 250))  
  
    frogs.append(hover_frog)
```

This code uses a for loop to create a number of frogs, in this case 10, to create `HoverFrog` objects at random coordinates, then adds the object to the list. We don't need the `hover_frog` variable after this loop because they can now be found inside the list.

The only other change we need to make is to draw the frogs inside the `draw()` function. Again, we can use a loop for this to extract each frog and display it:

```
for frog in frogs:  
  
    frog.display() # display the frog  
  
    frog.fly() # fly!
```

As we have seen before with lists, you can use a list in a for loop to extract each element (frog):

```
for frog in frogs:
```

```
frog.display() # display the frog
```

```
frog.fly() # fly!
```

As each element in the list is a frog object, when we retrieve it from the list, we can still use its methods. The *frog* variable in the list acts as our variable name for the object and just gets reused each time the loop goes round.

Here is the entire code:

```
frogs = [] # list of frogs
```

```
def setup():
```

```
    global frogs
```

```
    size(300,300)
```

```
    background(255)
```

```
    for i in range(10):
```

```
        hover_frog = HoverFrog(random(0, 200), random(0, 250))
```

```
        frogs.append(hover_frog)
```

```
def draw():
```

```
    global frogs
```

```
    background(255)
```

```
    for frog in frogs:
```

```
        frog.display() # display the frog
```

```
        frog.fly() # fly!
```

```
class Frog:
```

```
def __init__(self, x, y):

    self.x = x

    self.y = y

    # displays the frog

    def display(self):

        strokeWeight(2)

        fill(0, 128, 0) # colour

        # body

        ellipse(self.x, self.y, 90, 45)

        # feet

        fill(107, 142, 35)

        ellipse(self.x - 30, self.y + 15, 45, 23)

        ellipse(self.x + 30, self.y + 15, 45, 23)

        # Eyes.

        fill(255, 255, 255)

        ellipse(self.x - 15, self.y - 23, 23, 30)

        ellipse(self.x + 15, self.y - 23, 23, 30)

        fill(0, 0, 0)

        ellipse(self.x - 15, self.y - 23, 8, 8)

        ellipse(self.x + 15, self.y - 23, 8, 8)

        # Mouth.

        arc(self.x, self.y - 2, 30, 5, 0, PI)
```



```
class HoverFrog(Frog):  
  
    def display(self):  
  
        # rotor blades  
  
        line(self.x, self.y - 22, self.x, self.y - 50)  
  
        line(self.x - 30, self.y - 63, self.x + 30, self.y - 35)  
  
        line(self.x - 30, self.y - 35, self.x + 30, self.y - 63)  
  
        # frog  
  
        Frog.display(self)  
  
    def fly(self):  
  
        self.x += self.x/20
```

You can try altering the number of frogs in the setup() method for more froggy fun!



Figure 107: A lot of hover frogs

Polymorphism, which means ‘many forms’ is an object-oriented principle that allows objects that act the same to be treated the same. For example, both Frog and HoverFrog have a display() method, so we already know that we can use it for both types of objects.

Even if HoverFrog did not have a display() method itself, we still know it must have one because it will have inherited it from Frog, so there is no situation where calling display() on Frog or HoverFrog objects will fail.

As an example, if we change the code we have written to add both Frog and HoverFrog objects to a list:

```
for i in range(10):  
  
    rand = random(0,10)  
  
    if rand < 5:  
  
        frog = HoverFrog(random(0, 200), random(0, 250))  
  
    else:  
  
        frog = Frog(random(0, 200), random(0, 250))  
  
    frogs.append(frog)
```

This will now create either Frog or HoverFrog objects based on a random selection, and add them to the list.

If you run this, you will see that you get an error that says “Frog instance has no attribute ‘fly’”. Can you work out why this is?

Hopefully you realised that the error message is correct. We added Frog and HoverFrog objects to the list, then tried to call `display()` and `fly()` on all of them. Only hover frogs have the `fly()` method, so it is understandable that this would not work on Frog objects, as they do not have this method.

If we comment out the call to the `fly()` method:

```
for frog in frogs:  
  
    frog.display() # display the frog  
  
    #frog.fly() # fly!
```

It works! Of course none of them fly, but the program is able to display both frogs and hover frogs.



Figure 108: Polymorphic frogs and hover frogs

So the reason this is polymorphic (using polymorphism) is that the line:

```
frog.display() # display the frog
```

Works regardless of the type that ‘frog’ actually relates to – Frog or HoverFrog objects. This means that we don’t need to worry about the object type, provided we only use methods that all the types being used have in them.

A Longer Example

I think you can cope with a bigger example, so let’s draw an animated pirate ship scene.



Figure 109: Pirate ship animation

Pirate Ship

Let’s draw a pirate ship! We will start with a fairly basic ship that doesn’t move but will make some of it configurable so we can draw different sizes and colours of ships at different locations in future.

The full code for this programme can be found in Appendix A, but we will walk through it.

First, let's create a class for the pirate ship called `PirateShip` with some variables that we will use to decide where and how to draw the ship. These will be used later when we have multiple ships:

```
class PirateShip:

    # pirate ship variables

    x_position = 0

    y_position = 0

    ship_length = 0

    sails = 0

    sail_colour = 0

    porthole_count = 0

    sail_spacing = 0
```

These variables determine the x- and y-coordinates of the ship at any given time, how long it is, details about the sails (how many, how far apart etc) and the number of portholes. The reason we have created variables for the sails and portholes is so that they can be based on the length of the ship.

Next we should initialise our variables:

```
def __init__(self):

    # initialise the ship

    self.initialise()

    def initialise(self):

        # initialise the ship
```

```
self._position = width//2

self.y_position = height//2

self.ship_length = 60

# sails

self.sails = int(min((self.ship_length//30), 3))

self.sail_colour = color(0, 0, 0)

self.sail_spacing = floor(self.ship_length / 3)

# portholes

self.porthole_count = int(self.ship_length//12)
```

These functions initialise the variables we created for our pirate ship. Notice that the sails and portholes are based on the length of the ship, so longer ships will have more sails and portholes. The reason this is in a separate method, which we've called `initialise()`, is so that we can reset them later when a ship goes off the screen. We can just tell the `__init__()` method to use the `initialise` method so that it does the same thing when a ship is created.

The Hull

Now we need to write the code to actually draw the ship! First of all, the hull:

```
def display(self):

# calculate ship coordinates

front_of_ship_x_position = self.x_position - (self.ship_length / 2)
```

```

back_of_ship_x_position = self.x_position + self.ship_length + (self.ship_length / 3)

# ship hull

# brown lines and fill

stroke(139, 71, 38)

fill(139, 71, 38)

# middle rectangle

rect(self.x_position, self.y_position, self.ship_length, 20)

# front and back

triangle(front_of_ship_x_position, self.y_position, self.x_position, self.y_position, self.x_position,
self.y_position + 20)

triangle(self.x_position + self.ship_length, self.y_position, back_of_ship_x_position, self.y_position,
self.x_position + self.ship_length, self.y_position + 20)

```

This code, in a new method we called `display()`, determines the front and back coordinates of the ship and draws a brown rectangle and two triangles to make a ship's hull shape.



Figure 110: Ship's hull

Sails

Next the sails:

```

# draw the sails

initial_sail_x_position = self.x_position + 15

for i in range(self.sails):

```

```

# black lines

stroke(0)

# sail

fill(self.sail_colour)

quad(initial_sail_x_position + (self.sail_spacing * i), self.y_position - 40, initial_sail_x_position +
(self.sail_spacing * i) + 10, self.y_position - 40, initial_sail_x_position + (self.sail_spacing * i) + 20,
self.y_position - 10, initial_sail_x_position + (self.sail_spacing * i), self.y_position - 10)

# mast

fill(0)

line(initial_sail_x_position + (self.sail_spacing * i), self.y_position - 10, initial_sail_x_position +
(self.sail_spacing * i), self.y_position)

line(initial_sail_x_position + (self.sail_spacing * i), self.y_position - 40, initial_sail_x_position +
(self.sail_spacing * i), self.y_position - 50)

```

Again, we determine the start position. Then, in a loop depending on how many sails we calculated we wanted, this code draws a quad (four-sided shape) that represents a sail and a mast using lines.



Figure 111: Ship with sails

Jolly Roger

For fun, let's also add a flag on top of each sail:

```

# flag

fill(0)

rect(initial_sail_x_position + (self.sail_spacing * i), self.y_position - 50, 10, 5)

```

```
# jolly roger

stroke(255)

line(initial_sail_x_position + (self.sail_spacing * i) + 2, self.y_position - 49, initial_sail_x_position +
(self.sail_spacing * i) + 8, self.y_position - 46)

line(initial_sail_x_position + (self.sail_spacing * i) + 8, self.y_position - 49, initial_sail_x_position +
(self.sail_spacing * i) + 2, self.y_position - 46)
```

This code draws a rectangle to represent a flag, and a white cross to represent a jolly roger. Of course it doesn't really look like a jolly roger, but the ships are quite small, so I think this works.



Figure 112: Ship with flags

Portholes

The last thing to draw to complete our ship is the portholes:

```
# draw the portholes

stroke(0)

for i in range(self.porthole_count):

# portholes

fill(0)

circle(self.x_position+(15*i),self.y_position+10,5)
```

This just uses a loop, using the number of portholes we calculated, to draw some circles.

Movement

We have our first pirate ship! Now let's make it move across the screen.

We already have variables for the coordinates so we can use those to move the ship by changing the coordinates by adding a new method:

```
def move(self):  
  
    self.x_position -= 1
```



Figure 113: Ship with portholes

This method simply takes 1 off the x-coordinate so that it moves left on the screen. We can also configure the ship's speed by adding a new variable:

```
speed = 0
```

Then initialising it in the initialise() method:

```
self.speed = self.ship_length//20
```

This will set the speed of the ship in relation to its length. Adding this into the move() method we created means we can now configure the speed it moves:

```
def move(self):  
  
    self.x_position -= self.speed
```

Of course it will eventually disappear off the edge of the screen, so the last thing for our ship is to account for this. Firstly, let's change the starting position of the ship so it has the whole screen to move:

```
self.x_position = random(600, 700)
```

```
self.y_position = random(50, 350)
```

Then reset the position of the ship when it disappears off the screen by adding some code to the `display()` method:

```
# ship has left the screen

if back_of_ship_x_position <= 0:

    # start again

    self.initialise()
```

The ship will no reappear when it has left the screen by using the `initialise()` the method we created. If you also change the length of the ship to a random number:

```
self.ship_length = random(40, 100)
```

It will now choose a random size for the ship each time it is initialised. As we also set the number of sails and portholes, and the speed, to be calculated from the length, these will change randomly too!

A Fleet of Pirate Ships

We now have the code we need to create a fleet of ships as we can have multiple objects of the same `PirateShip` class, and each one will initialise differently due to the use of the random numbers. As we used variables for the ship's features (coordinates, length etc) it means that each object can have its own variables, giving us different ships.

Instead of a single ship variable, we will need a number of them, so a list makes sense. So, we should change:

```
global ship
```

```
ship = PirateShip()
```

To:

```
global ships
```

```
ships = []
```

Then initialise the list to contain a number of pirate ships, in this case, 10:

```
for i in range(10):
```

```
    ships.append(PirateShip())
```

This will create an array of ten pirate ships. Each one, using the `__init__()` function will initialise with random starting positions and lengths. All we need do then is display and move them all in the `draw()` function by changing:

```
global ship
```

```
ship.display()
```

```
ship.move()
```

To:

```
global ships
```

```
for ship in ships:
```

```
    ship.display()
```

```
    ship.move()
```

As you can see, it is not that different. We are just using an array and a loop for all the ships.



Figure 114: Pirate fleet

They are a bit boring all having the same sails though, right? Easy to change though, because we already have a `sail_colour` variable that we can just initialise to a random colour. So we can change:

```
self.sail_colour = color(0, 0, 0)
```

To:

```
self.sail_colour = color(random(255), random(255), random(255))
```

This will randomise the colour each time it is initialised.



Figure 115: Brightly coloured sails

Waves

Now that we have our brightly coloured pirate ship fleet, we just need to add the waves.

Hopefully by now you can see the advantage of using variables, rather than simply typing values in, as it means they can easily be changed and reused, so we'll do the same for the waves.

Firstly, create global variables in the `setup()` function for the wave heights.

```
# waves

global wave_max_height

wave_max_height = 20

global wave_min_height

wave_min_height = 5

global wave_height

wave_height = wave_max_height
```

These specify the maximum and minimum wave heights, and initialises the wave height to be the maximum.

Then we need a loop, or rather nested loops, to draw waves horizontally and vertically on the screen in the draw() function.

```
# waves

global wave_max_height

global wave_min_height

global wave_height

noFill()

# wave y coordinate

wave_y = wave_max_height//2

# while the y coordinate is less than the height of the window

while wave_y <= height:

# wave x coordinate
```

```
wave_x = 0

# while the x coordinate is less than the width of the window

while wave_x <= width:

# draw a semi=circle arc

arc(wave_x, wave_y, wave_max_height, wave_height, 0, PI)

# increase the x coordinate by 20

wave_x += wave_max_height

# increase the y coordinate by 20

wave_y += wave_max_height
```

These loops draw arcs across the screen, using the variables to draw them and for spacing them out. Running this code will now show waves, although they don't move, so that is the last piece of our programme to solve.

Firstly, the waves will need to be told whether they are currently going up or down with some variables:

```
global up

up = 1

global down

down = -1

global wave_direction

wave_direction = up
```

These variables, declared in the `setup()` function, determine the current direction of the waves. They can then be used in the `draw()` function to change the height of the waves:

```
# change the wave height for next time
```

```
wave_height += wave_direction
```

Of course, this only changes the height in one direction, because `wave_direction` never changes. Some if statements can change that:

```
# determine if the waves should be going up or down
```

```
if wave_height <= wave_min_height:
```

```
    wave_direction = up
```

```
if wave_height >= wave_max_height:
```

```
    wave_direction = down
```

These if statements simply reverse the direction of the waves if the maximum or minimum height is reached. Now we have moving waves and a complete pirate ship programme!



Figure 116: Completed pirate ship programme

The full code for this programme can be found in Appendix A.

CHAPTER 12: TRANSFORMATION

In addition to defining where to draw a shape, using coordinates, Processing has functions that allow you to alter the way shapes are displayed. There are:

- `translate()` – changes the location
- `scale()` – changes the size
- `rotate()` – changes the rotation

Note that these functions don't actually change the shape's details, for example its coordinates, only displays them differently.

Translating

Translating is displaying something at a different place on the screen. Why would you want to do this when you could just change the coordinates? The main reason is simplicity. If you wanted to draw a square and then draw another square 50 pixels away, instead of working out the coordinates you can just translate the second square by 50. For example, what do you think this code will do?

```
square(0, 0, 50)
```

```
translate(50, 50)
```

```
square(0, 0, 50)
```

It will draw a square at coordinates 0, 0 then another square at 50, 50.

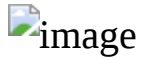


Figure 117: Translated squares

This may seem a little confusing, so let's break it down a bit:

```
square(0, 0, 50)
```

This line works as expected, it draws a square at coordinates (0, 0).

```
translate(50, 50)
```

This line says to draw the next thing 50 pixels across and 50 pixels down from the coordinates specified.

```
square(0, 0, 50)
```

This line says to draw a square at coordinates (0, 0) except that we just translated the display by (50, 50) so it will actually be 50 pixels across (x) and 50 pixels down (y) from the coordinates 0, 0. So it will actually draw the square at coordinates (50, 50).

This can make it easier to visualise because we only need to specify the distance from the last shape, rather than working out coordinates. It also makes it easier to specify coordinates as variables which makes them easier to change.

```
x = 0
```

```
y = 0
```

```
def setup():
```

```
size(200, 200)
```

```
def draw():  
  
    global x  
  
    global y  
  
    square(x, y, 50)  
  
    translate(x + 50, x + 50)  
  
    square(x, y, 50)  
  
    translate(x + 50, x + 50)  
  
    square(x, y, 50)
```

This code uses the x and y variables as coordinates which means that, if we want to change where the starting point is, we can just change those variables at the top and everything will move accordingly.

We have two translate() function calls here, which are both the same but each one incrementally moves the display by 50. So the three squares are all drawn at different locations despite having the same (x, y) coordinates.

Scaling

Scaling works a bit like translating except that, rather than displaying shapes at a different location, it displays them at a different size.



Figure 118: More translated squares

Let's alter our earlier code to add scaling:

```
def draw():
```

```
global x
```

```
global y
```

```
square(x, y, 50)
```

```
translate(x + 50, x + 50)
```

```
scale(1.5) # scale 150%
```

```
square(x, y, 50)
```

```
translate(x + 50, x + 50)
```

```
scale(1.5) # scale 150%
```

```
square(x, y, 50)
```

Scaling works in percentages, so the code:

```
scale(1.5)
```

means scale by 150%, or make the following shape 50% larger if you prefer to think of it like that. This means that scaling by 1.0 will have no effect, and scaling by a number less than 1 will make the following shapes smaller.



Figure 119: Scaled squares

Notice that the entire shape has been scaled in this example. It is also possible to scale different sides by different amounts. For example:

```
scale(1.5, 0.5)
```

This will increase scale by 50% on the x-axis (horizontal) and reduce scale by 50% on the y-axis (vertical). If you try changing your code so it does

this you will see that the squares (or rather, rectangles) get wider and shorter.



Figure 120: Scaling different axes

Rotation

The `rotate()` function, as you might expect, rotates shapes and other graphics. It works similar to how arcs work, by using radians. In this case radians are used to rotate anti-clockwise. For example:

```
rotate(HALF_PI)
```

Means that, using the `rotate()` function, we can alter the appearance of the shapes rotation.

So if we rotate by $\pi/2$ (`HALF_PI`), for example, it would rotate by 90 degrees.

```
def draw():
```

```
    global x
```

```
    global y
```

```
    rect(x, y, 50, 30)
```

```
    translate(x + 50, x + 50)
```

```
    rotate(HALF_PI)
```

```
    rect(x, y, 50, 30)
```



Figure 121: Radians of rotation

This code will draw two identical rectangles but rotate one of them by 90 degrees.



Figure 122: Rotated rectangles

Let's combine these to create a spiral staircase effect.

```
def setup():  
    size(500, 500)  
    background(0)  
  
    def draw():  
        translate(width/2,height/2)  
  
        for i in range(70):  
            rotate(0.1)  
  
            scale(1.01)  
  
            square(0, 0, 100)
```

You might think there is no point in rotating a square but, as you will see if you run this code, you don't have to rotate a full 45 or 90 degrees, so you can achieve an interesting effect by rotating just a small amount.



Figure 123: Spiral staircase

Here we use `translate()` to start in the middle of the screen, rather than have to work out the coordinates, `rotate()` to rotate the square by a small amount and `scale` to progressively make the squares bigger as though they are getting nearer to us.

The Transformation Matrix

As you have seen, the `draw()` function repeats, drawing a frame each time, but each time it repeats it resets itself and draws the frame all over again. If what you draw is the same each frame then it appears as though nothing is moving when in reality it is just drawing the same thing over and over again.

The transformation matrix is what stores the effects of using `scale()`, `translate()` and `rotate()` for the rest of the frame. So if you use `scale()` and draw a series of shapes, they will all be scaled by that amount. As you've seen, if you use `scale()` again in the same frame, it is cumulative so it adds the scale effect to the previous scaling. It works the same for `rotate()` and `translate()`. When the frame is re-drawn, the matrix resets and the effects are applied again. Look again at the spiral staircase code:

```
def setup():  
  
    size(500, 500)  
  
    background(0)  
  
    def draw():  
  
        translate(width/2,height/2)  
  
        for i in range(70):  
  
            rotate(0.1)
```

```
scale(1.01)
```

```
square(0, 0, 100)
```

The code has `translate()`, `rotate()` and `scale()` which are applied on top of each other. The graphics are first translated to the middle of the screen, then the rotation and scale are applied in a for loop. The loop in our code has a range of 70, so it will repeat 70 times. As an example of how the transformation matrix is cumulative, each time the loop repeats, the rotation and scale are applied, but they are applied on top of the previous ones, which is why the squares rotate and get bigger.

When the code in the `draw()` function ends, the transformation matrix is reset and is drawn all over again. In this case we draw the same shapes at the same location so it appears that the image is static.

Resetting the Transformation Matrix

So far, we've used the transformation matrix as a whole with any effects we apply (`rotate()`, `translate()`, `scale()`) applying for all the code that comes after it. What happens here is that the matrix stores the transformations we have applied until the `draw()` function starts again and resets it.

What if we wanted to reset it ourselves in the middle of a `draw()` function? For example, maybe we want to apply transformation effects to one or more shapes, but then want to assume no effects have been applied? We could work all the transformations back so that the system is effectively reset. For example we could apply negative `translate()` and `scale()` until it reaches zero again. Sounds annoying and complicated!

There is a very simple way of doing this using `resetMatrix()`:

```
def draw():  
  
    translate(width/2, height/2)  
  
    square(0, 0, 10)  
  
    resetMatrix()  
  
    square(0, 0, 20)
```

If you run this you will see that the second square does not take on the `translate()` from the first because we reset it. Therefore the coordinates of the second square (0, 0) really means (0, 0) which is the top-left of the window.



Figure 124: Resetting the matrix

You can reset the matrix as many times as you need to.

Remembering the Transformation Matrix

So far we have seen that the transformation matrix gets reset by the `draw()` function. So each time `draw()` is used the matrix starts again. What if we wanted it to remember the transformations we had made? For this we need to use the *matrix stack*.

The matrix stack is what stores the transformation effects. So when we use `rotate()`, `scale()` or `rotate()` the matrix remembers what we have done and this is why the effects are cumulative. When the `draw()` function is used again, it resets the matrix stack.



Figure 125: The matrix stack

We can utilise the matrix stack to pull off a piece of matrix and apply transformations to it, then put it back.

So how do we stop the matrix stack from being reset each time `draw()` is run? We have to put our own bit of matrix at the top of the stack. This is known as pushing. Think of it like a stack of shopping baskets. If we want to store our transformations and not have them reset we need to put our own basket on the top of the stack of baskets.



Figure 126: Pushing an initial matrix on to the stack

Since we have to do this first, it is usually done in the `setup()` function:

```
def setup():  
    size(500, 500)  
    background(0)  
    pushMatrix()
```

When we start to draw we now need to access the matrix we have pushed on the stack. Retrieving it is called popping, using `popMatrix()`. Once we pop the matrix from the stack we are working with the one we pushed rather than the 'normal' transformation matrix

```
def draw():  
    popMatrix()
```



Figure 127: Popping a matrix from the matrix stack

Once we have made our transformations, to make sure we don't lose them when the matrix gets reset, we have to push it back onto the stack.



Figure 128: Pushing transformation on to the matrix stack

We can use this to change our staircase example to make an infinite staircase:

```
def setup():  
  size(500, 500)  
  background(0)  
  pushMatrix() # push a matrix  
  
  def draw():  
    popMatrix() # retrieve the matrix  
    translate(width/2,height/2) # translate  
    rotate(0.1) # rotate the screen  
    translate(-width/2,-height/2) # translate it back again  
    scale(1.01) # scale  
    square(width/2, width/2, 100) # draw a square  
    pushMatrix() # push the matrix back
```

This code extends the staircase example to use the matrix stack rather than a loop. We use the matrix to remember the previous transformations by popping it at the start of draw() and applying transformations over the top of the previous ones. Then pushing it back to the stack.



Figure 129: Infinite staircase

If you remove the pushMatrix() and popMatrix() functions, you will see that it doesn't remember the transformations and will just draw a static square on the screen

Let's try a longer example using shapes, loops and the matrix stack to draw a series of rotating shapes.

```
# sizes of the 2D shapes

hexagon_size = 60

pentagon_size = 40

def setup():

    size(800, 600)

    def draw():

        # define rotation direction variables

        clockwise_direction = frameCount * PI

        anticlockwise_direction = frameCount * -PI

        # reset the background

        background(255)
```

```

# width and height

i = 0

j = 0

while i < width + hexagon_size:

    while j < height + hexagon_size:

        pushMatrix()

        translate(i, j)

        rotate(anticlockwise_direction / 150)

        draw_polygon(6, hexagon_size)

        rotate(clockwise_direction / 100)

        draw_polygon(5, pentagon_size)

        j += hexagon_size * 2

    popMatrix()

    j = 0

    i += hexagon_size * 2

#Draw a 2D polygon with the given number of sides and size

def draw_polygon(sides, size):

    # fill colour

    fill(85, 26, 139, 30)

    # line colour

    stroke(138, 43, 226, 20)

    # starting coordinates

```

```
x = 0

y = 0

# centre coordinates

centre_x = 0

centre_y = 0

# work out the centre coordinates using the coordinates of each vertex

for i in range(sides):

    # work out the coordinates

    x_coord = (x + size * cos(i * 2 * PI / sides))

    y_coord = (y + size * sin(i * 2 * PI / sides))

    # add them all together

    centre_x += x_coord

    centre_y += y_coord

# calculate the centre coordinates

centre_x = (centre_x / sides) * -1

centre_y = (centre_y / sides) * -1

# draw the polygon

beginShape()

# loop for the number of sides + 1 (so we join up the last side to the starting vertex)

for i in range(sides):

    # work out the coordinates of the vertex

    x_coord = (centre_x + size * cos(i * 2 * PI / sides))
```

```
y_coord = (centre_y + size * sin(i * 2 * PI / sides))
```

```
# plot the vertex
```

```
vertex(x_coord, y_coord)
```

```
endShape()
```

This code no doubt looks complex but let's walk through it.

We have a function called `draw_polygon()` that firstly calculates the centre of the shape we are about to draw:

```
# starting coordinates
```

```
x = 0
```

```
y = 0
```

```
# centre coordinates
```

```
centre_x = 0
```

```
centre_y = 0
```

```
# work out the centre coordinates using the coordinates of each vertex
```

```
for i in range(sides):
```

```
# work out the coordinates
```

```
x_coord = (x + size * cos(i * 2 * PI / sides))
```

```
y_coord = (y + size * sin(i * 2 * PI / sides))
```

```
# add them all together
```

```
centre_x += x_coord
```

```
centre_y += y_coord
```

```
# calculate the centre coordinates
```

```
centre_x = (centre_x / sides) * -1
```

```
centre_y = (centre_y / sides) * -1
```

This code uses a for loop to work out what the coordinates of the sides would be, given the number of sides, and calculates the centre x- and y-coordinates.

The function then draws a custom shape, using vertices, to draw a shape with the same number of sides using the calculated centre coordinates, again using a for loop:

```
# draw the polygon
```

```
beginShape()
```

```
# loop for the number of sides + 1 (so we join up the last side to the starting vertex)
```

```
for i in range(sides):
```

```
# work out the coordinates of the vertex
```

```
x_coord = (centre_x + size * cos(i * 2 * PI / sides))
```

```
y_coord = (centre_y + size * sin(i * 2 * PI / sides))
```

```
# plot the vertex
```

```
vertex(x_coord, y_coord)
```

```
endShape()
```

This function is used in the draw() function to draw the shapes. As we have a general function, draw_polygon(), that draws any number of sided shapes,

we can draw a hexagon and a pentagon in the same place but rotate them in different directions using rotate():

```
# width and height

i = 0

j = 0

while i < width + hexagon_size:

    while j < height + hexagon_size:

        resetMatrix()

        translate(i, j)

        rotate(anticlockwise_direction / 150)

        draw_polygon(6, hexagon_size)

        rotate(clockwise_direction / 100)

        draw_polygon(5, pentagon_size)

        j += hexagon_size * 2

    j = 0

    i += hexagon_size * 2
```

This code uses a while loop inside a while loop (nested while loops) to draw shapes across the width and height of the screen. We have to use while loops for this because we want to change the increase in the width and height each time – in this example `hexagon_size * 2` – to ensure the shapes are spaced apart when we draw them. The `translate()` function is used to move where the shapes are drawn.

Notice the use of `resetMatrix()` before the `translate()` to ensure that we start from zero each time. This means that the `draw_polygon()` function doesn't need to know where the shapes are going to be drawn and `translate()` just moves where they are displayed.

Running this gives you a display of hexagons and pentagons, rotating in different directions



Figure 130: Rotating hexagons and pentagons

CHAPTER 13: IMAGINE MANIPULATION

Graphics Formats

Before we can talk about manipulating images, it is important to understand how images are stored. There are two types:

- Raster graphics – made up of coloured dots
- Vector graphics – made up of coordinates and mathematical equations

We've used both types so far. We've looked at pixels (dots) and shapes (vector). The advantage of vector graphics is that we don't need to specify every dot to make a shape. However, Processing uses raster graphics for images since this is the most common way of storing images where there is a wide variation in colour, as you would expect from a photograph or painting.

Each dot in the image is simply a coloured pixel that are combined in such numbers that they give the effect of a single image. The types you can use in Processing are JPEG (.jpg or .jpeg), TGA (.tga), GIF (.gif) and PNG (.png) formats.

Manipulating Pixels

Before we load an image, let's have a look at how we can manipulate individual pixels. It is possible to set the colour of an individual pixel using the `set()` function in the form `set(x, y, colour)`:

```
def setup():
```

```
  size(400, 400)
```

```
background(255)

def draw():

    # choose a random pixel

    random_x = int(random(width))

    random_y = int(random(height))

    # change it to blue

    set(random_x, random_y, color(0, 0, 255))
```

Here we are using the `set()` function to change the colour of the pixel to blue. You may have to leave it running for a while to see the effect. Of course we could have done this using the `dot()` function, so why use the `set()` function at all?



Figure 131: Changing pixel colour

Well, one thing we can also do is retrieve a pixel which we can then use to make decisions. If we draw a white circle in the middle of the screen, we can check if the random pixel we chose if already changed (blue), or unchanged (white).

```
def setup():

    size(400, 400)

    background(0)

    fill(255)

    noStroke()
```

```

ellipse(width/2, height/2, width/2, height/2)

def draw():

    # choose a random pixel

    random_x = int(random(width))

    random_y = int(random(height))

    # get the pixel

    pixel = get(random_x, random_y)

    # if it is white

    if pixel == color(255):

        # change it to blue

        set(random_x, random_y, color(0, 0, 255))

```

Here we are using the `get()` function to retrieve the random pixel, and then using an if statement to check its current colour. As we have drawn a white circle on a black background, we can check if the pixel is white and, if it is, change it to blue. This means that none of the black background pixels will be changed.



Figure 132: Selective pixel colouring

The Pixel List

There is another way of accessing the pixels and that is to use the pixel list. This allows us to retrieve all the pixels at once, as a list, using `loadPixels()` which gives you a list called `pixels[]`.

The pixel list will contain all the pixels from the screen so if you have set a `size(500, 500)` then there will be $500 * 500$ (250,000) pixels in the list.

In addition to manipulating individual pixels using the `set()` and `get()` functions, Processing provides some language structures that are built to speed pixel calculations up and to access the entire set of pixels at once:

- `pixels[]` - list containing the current state of the pixels, each of which can be manipulated
- `loadPixels()` - function that loads the pixels on-screen into the `pixels[]` list
- `updatePixels()` - function that updates the pixels on-screen to represent those in the `pixels[]` list, which may have been changed

The `pixels[]` list organises the pixels of the display window as a sequence of *height* rows of *width* elements long into the list; for example a display window of 500x500 has the pixels list ($\text{width} * \text{height}$) of 250,000 elements long. To locate a specific pixel (x, y) you can use the formula:

```
pixels[(y-1)*width+x-1]
```

So, we could change our previous code to the following:

```
def setup():
```

```
size(400, 400)
```

```
background(0)
```

```
fill(255)
```

```
noStroke()
```

```
ellipse(width/2, height/2, width/2, height/2)
```

```
def draw():  
  
    # choose a random pixel  
  
    random_pixel = int(random(width * height))  
  
    loadPixels()  
  
    # get the pixel  
  
    pixel = pixels[random_pixel]  
  
    # if it is white  
  
    if pixel == color(255):  
  
        # change it to blue  
  
        pixels[random_pixel] = color(0, 0, 255)  
  
    updatePixels()
```

Note the use of `updatePixels()` which makes the changes to the pixel list display on the screen.

You might be thinking ‘but this isn’t easier than using `set(x, y)`’, and you’d likely be right. The power of using the pixel list is the ability to change large numbers of pixels at once and then reflect this in the display. As it is a list, we can use a loop to change lots of pixels. It will also use less computing power and therefore be more efficient.

For example, the following code changes the colour of every pixel to a random colour every time, resulting in a static-like animation:

```
def setup():  
  
    size(500, 500)
```



```
def draw():  
  
    loadPixels()  
  
    # every pixel  
  
    for i in range(len(pixels)):  
  
        # set a random colour  
  
        pixels[i] = color(random(255), random(255), random(255))  
  
    updatePixels()
```

Note that you will always need `loadPixels()` and `updatePixels()` or it won't work.



Figure 133: Changing all pixels

Using Images

Before we can use images in our code, we first need to load the image we want to use. For example:

```
size(500, 500)  
  
img = loadImage("image.jpg")  
  
image(img, 0, 0)
```

This code loads an image called `image.jpg` and displays it at coordinates (0, 0). The image needs to be located in the directory where your code is saved. This is called a 'relative path' because the path (location) of the image is relative to where the code is running.



Figure 134: Relative image path

An alternative to this is to use an ‘absolute’ or ‘full’ path which tells the program exactly where the file is. For example:

```
def setup():  
  
    size(500, 500)  
  
    img = loadImage('C:\\Temp\\image.png')  
  
    image(img, 0, 0)
```

This code located the image in a folder called ‘Temp’ on the C: drive of my computer. This means I don’t need to copy the image to the folder with my program, but it does mean it won’t work for anyone except me, unless they have the same file in the same folder.

You can also display an image as the background:

```
def setup():  
  
    size(500, 500)  
  
    img = loadImage('image.png')  
  
    background(img)
```

However, the image must be the same size as your screen. So, if you specify size (500, 500) then the image must be 500 * 500 pixels. You can resize it though using `resize()`:

```
def setup():  
  
    size(500, 500)  
  
    img = loadImage('image.png')
```

```
resize(500, 500)
```

```
background(img)
```

Or you can use the image function to display it as a given size without altering the original size:

```
def setup():
```

```
size(500, 500)
```

```
img = loadImage('image.png')
```

```
image(img, 0, 0, width, height)
```

This will display the image at coordinates (0, 0) and resize it to the width and height of the display screen. Note that this could mean the image looks squashed or strange if the dimensions of the original image are different to the dimensions of the display screen.

You will usually want to load your image in the setup() function, rather than draw(), because it is an intensive operation and may slow down your program. You can still display the image in the draw() function of course using a global variable for it:

```
def setup():
```

```
global img
```

```
size(500, 500)
```

```
img = loadImage('image.png')
```

```
def draw():
```

```
global img
```

```
image(img, 0, 0, width, height)
```

Exception handling

Let's take a brief aside at this point and talk about what happens when code goes wrong. Or rather, when you try to do something that results in an error you could do something about. This is known as exception handling.

An exception is an 'exceptional event' or something that happens that shouldn't have. With images, it is easy to try to load something that doesn't actually exist. For example, you typed the filename wrong, or forgot to put the image in the folder. Usually this means the end of your program but you *could* choose to 'handle' the exception instead.

Try running the code above but change the filename to something that doesn't exist. The program fails with an error, right?

Now you probably just want to fix the code, but you can also handle this exception by telling the program to *attempt* to display the image but do something else if it fails. For example:

```
def setup():  
  
    global img  
  
    size(500, 500)  
  
    img = loadImage('image.png')  
  
    def draw():  
  
        global img  
  
        try: # attempt  
  
            image(img, 0, 0, width, height)
```

```
except: # if it fails
```

```
background(0, 240, 120)
```

This code will:

- (attempt) to display the image, but
- (if it fails) create a background colour instead

You *can* do this for any kind of errors, but you *should* only use it for recoverable errors, ie ones you can do something about. Otherwise, you should just fix the code that caused the error instead.

Images as Pixels

As the image file formats supported in Processing (jpg, gif, tga, png) are all raster graphic formats, we can manipulate the pixels of images in the same way.

We can still get specific pixels using `get()`. This could be used to retrieve portions of an image using `get(x, y, w, h)` where `w` is the width of a rectangle of pixels you want to get and `h` is the height:

```
def setup():
```

```
  global img
```

```
  size(500, 500)
```

```
  img = loadImage('image.jpg')
```

```
  img.resize(500, 500)
```


```
def draw():
```

```

global img

image(img, 0, 0, width, height)

newImage = img.get(300, 300, 200, 200) # get part of the image

image(newImage, 0, 0) # display it over the top
image

```

Figure 135: Using part of an image

This code copies the pixels making up part of the image and creates a new image of just those pixels.

We can also manipulate individual pixels using the pixel list as we did before. For example, we could change the colour of the pixels. However, we now don't *need* to load and update all the pixels, because images already have a `pixels[]` list associated with them, so we can simply use that by using the variable name for the image (`img` in this example) `img.pixels`:

```

def setup():

    global img

    size(500, 250)

    img = loadImage('image.jpg')

    img.resize(250, 250)

    def draw():

        global img

        image(img, 0, 0) # display the original image

        newImage = createImage(250, 250, RGB) #create a new image with pixels

```

```

for i in range(len(img.pixels)): # for every pixel in the original image

# access current pixel colour

current_pixel = img.pixels[i]

# access current RGB of the pixel

red_pixel = red(current_pixel)

blue_pixel = blue(current_pixel)

green_pixel = green(current_pixel)

# create new colour by swapping RGB values

newColour = color(blue_pixel, red_pixel, green_pixel)

newImage.pixels[i] = newColour # add new pixels for the new image

image(newImage, 250, 0) #display the new image

image(newImage, 250, 0)

```

Here we create a new image, using `createImage()`, and accesses the pixels from the original image by using the pixel list. Each pixel is then altered slightly by swapping the RGB elements and the pixel is altered and copied onto the new image.



Figure 136: Altering the pixels to create a new image

Once we have the pixels for an image we can use them however we like. We could, for example, use them to create a reflection of the image by redisplaying them upside-down.

```
def setup():
```

```

global img

size(250, 500)

img = loadImage('image.jpg')

img.resize(250, 250)

def draw():

    global img

    image(img, 0, 0)

    # create new image the same size

    newImage = createImage(img.width, img.height, RGB)

    # every pixel wide and high

    for i in range(newImage.width):

        for j in range(newImage.height):

            # set new pixel based on the original

            newImage.set(i, newImage.height-j-1, img.get(i, j))

    image(newImage, 0, 250) # display new image

```

This code makes a copy of each pixel and places it in a new location in a new image, creating the effect of a reflection of the original.



Figure 137: Copying pixels to create a reflection

Image Manipulation Functions

We can create some interesting effects by manipulating individual pixels, but Processing also has a range of functions that can we use for image manipulation.

Masking

Masking is where you place an image, or a colour, over an image. The mask isn't opaque though, it is partially transparent so can be used to alter the colour effect of the image. Note that if using two images, they must be the same size.

```
def setup():  
  
  global inputImage  
  
  size(480, 640)  
  
  inputImage = loadImage("image.jpg")  
  
  inputImage.resize(480, 640)  
  
  noStroke()  
  
  def draw():  
  
    global inputImage  
  
    fill(255)  
  
    ellipse(300, 200, 100, 100) # draw a circle  
  
    loadPixels()  
  
    inputImage.mask(pixels) # apply mask to image  
  
    background(0) # apply a black background  
  
    image(inputImage, 0, 0) # display image
```

This example uses a white circle as the mask. The mask (the white circle) is drawn first and the image is drawn over the top. Finally we have drawn a black background over the top, which would normally mean all we could see is black.

However, because of the mask, the white circle will appear as a transparent circle, so we can see the image through the circle, a bit like shining a light directly on a picture. You could even create a ‘scratchcard’ type effect, by adding more circle until the image is finally revealed.



Figure 138: Blacked-out image with circular mask

Tinting

Tinting, as you might expect, applies a transparent tint over the top of the image

```
def setup():  
  
  global inputImage  
  
  size(480, 640)  
  
  inputImage = loadImage("image.jpg")  
  
  inputImage.resize(480, 640)  
  
  def draw():  
  
    global inputImage  
  
    image(inputImage, 0, 0)  
  
    tint(0, 150, 255) # apply tint
```

Here we apply a blue-green tint to the image.



Figure 139: Image with tint applied

Filters

Filters, a little like you might do with a modern camera or graphics design software, allows you to apply predefined filters to your images. For example, you could apply a blur effect.

The function is used like:

```
filter(MODE)
```

or:

```
filter(MODE, level)
```

where **MODE** is one of the below and **level** is the intensity of the filter.

For example, the **THRESHOLD** mode

```
image(inputImage, 0, 0)
```

```
filter(THRESHOLD)
```

The available filters are:

- **THRESHOLD** – takes the strongest colours and makes them white. Less intense colours are turned black. The level determines the point at which to use white or black
- **BLUR** – blurs the image. The level determines the amount of blur

- POSTERIZE – limits the colour values (red, green, blue) by the level. The minimum is 2. Using a value of 255 would mean it had no effect
- INVERT – inverts the colours in the image. Does not use a level
- GRAY – turns the image into a black-and-white one using grayscale (amounts of grey). Does not use the level
- ERODE – reduces light areas. Does not use the level
- DILATE – enhances the light areas. Does not use the level



Figure 140: Image filters applied to the same image

Blending

Blending is combining two images into one. We'll describe the images as source, destination, and result, but this is just for the explanation as the result will be a blend of the two images. The resulting image will depend on the blend mode used.

```
def setup():

    global source_img

    global destination_img

    size(640, 480)

    source_img = loadImage('stimpy.PNG')

    destination_img = loadImage('MrK.PNG')

    source_img.resize(640, 480)

    destination_img.resize(640, 480)

    def draw():
```

```
global source_img
```

```
global destination_image
```

```
blendMode(OVERLAY)
```

```
image(source_img, 0, 0)
```

```
image(destination_img, 0, 0)
```

As an example this code uses the OVERLAY blend mode.

.  image

Figure 141: Images blended in OVERLAY mode

You can also just blend an image with whatever graphics are on the screen, it doesn't have to be two images:

```
def setup():
```

```
global source_img
```

```
size(500,500);
```

```
source_img = loadImage("image.jpg")
```

```
def draw():
```

```
strokeWeight(20)
```

```
stroke(255,150,150)
```

```
fill(150,150,200)
```

```
ellipse(230, 230, 250, 250)
```

```
line(150, 310, 310, 150)
```

```
blendMode(OVERLAY)
```

```
image(source_img, 0, 0)
```

To achieve this, Processing blends each pixel together. To do this, the regions of the source and destination have to be the same size. If the regions are different sizes, the source will be automatically resized to fit the destination size. In the above example, the source image is much smaller than the destination, so the source is expanded to fit the size of the destination.



Figure 142: Image blended with screen contents

Blend Modes

Blend modes, such as OVERLAY that we've already seen work by blending the pixels of the images together. Each mode performs the blending differently.

General modes

REPLACE mode simply replaces one image with another, it doesn't really blend them at all.

BLEND mode is used for a smooth blending of two images. This is known as 'linear interpolation' or 'lerping'.

Darken modes

Darken modes generally favour darker pixels and will result in a darker image.

MULTIPLY mode multiplies the colours of the source and destination images which will make the resulting image darker.

SUBTRACT mode subtracts the source pixels from the destination image, resulting in a darker image. The lighter the source pixel is, the darker it will become in the result.

DARKEST mode preserves the darkest pixels from both images. This makes the darker pixels stand out more.

BURN mode tends to intensify colours but generally makes the resulting image darker.

Lighten Modes

Lighten modes, as you might expect, is the opposite to darken modes and result in lighter images.

SCREEN mode works like projecting the same image twice on to the same place on a wall. This has the effect of brightening light colours but preserving dark colours.

ADD mode simply adds colours together, resulting in a brighter image.

LIGHTEST mode makes the lighter pixels from both images stand out more. This enhances the lighter pixels.

DODGE Mode is the opposite of BURN mode. The lighter the source colour, the more colour is used.



Figure 143: Blend modes

Contrast Modes

Contrast modes alter the contrast of the image by changing how light and dark colours are treated.

OVERLAY mode we have already seen. This simply increases contrast, lightening the light pixels and darkening the dark ones.

HARD_LIGHT mode uses a combinations effect, using SCREEN mode where the pixel is more than 50% grey, otherwise it uses MULTIPLY mode. Naturally works better if there is a lot of grey

SOFT_LIGHT mode is like a softer version of OVERLAY mode. It is a mixture of DARKEST and LIGHTEST, so tends to enhance colours.

Comparative Modes

Comparative modes compare pixel colours between source and destination and are naturally good at comparing images.

DIFFERENCE mode compares the values of pixel colours and works out how different they are. Colours are subtracted from the resulting image.

EXCLUSION mode is like DIFFERENCE mode but less extreme.



Figure 144: DARKEST vs LIGHTEST blend modes

CHAPTER 14: EVENT HANDLING

Event handling is the idea that programmes can detect external ‘events’, which are things that happen during the time the program is running but that the programme is not pre-aware of when these will happen. The program will detect when an event occurs and act accordingly.

The events that Processing can handle are those that are created by user input – that is, mouse and keyboard input. In your programs so far, there hasn’t been any ability to control them using the keyboard or mouse. These are already being detected, but we haven’t told the program what to do.

Keyboard Events

Keyboard events occur whenever anything is typed on the keyboard. What our programs will need to do is to decide what will happen when a key is pressed. To be able to do this, Processing provides a set of built-in variables, and some event functions.

Keyboard Event Actions

keyPressed

The keyPressed variable is a built-in Boolean variable that is True if a key has been pressed (any key) and false if it has not. It can be used in a program to check if a key is being pressed. The variable is reset every time the draw() function starts and is set accordingly at that point. So if someone is pressing a key when draw() starts, the variable will be set to True. As it is a boolean, you can use it in an if statement:

```
if (keyPressed):
```

```
# do something
```

As an example, the code below moves a shape that looks a bit like a spaceship.

```
x = 350
```

```
y = 400
```

```
def setup():
```

```
    size(800, 600)
```

```
def draw():
```

```
    global x
```

```
    global y
```

```
    background(0) # black
```

```
    fill(0, 0, 255) # blue
```

```
    ellipse(x, y, 25, 25)
```

```
    triangle(x, y-18, x-20, y+10, x+20, y+10)
```

```
    # thrusters
```

```
    fill(255,255,0)
```

```
    ellipse(x-10, y+10, 5, 5)
```

```
    ellipse(x+10, y+10, 5, 5)
```

```
    if keyPressed:
```

```
        # move up the screen
```

```
        y -= 1
```

When you are pressing a key, it will move up the screen. If you stop pressing the key, it will stop moving.



Figure 145: Key controlled spaceship

keyPressed()

As well as the variable, there is also a keyPressed() function. The function will be called whenever a key is pressed. These kinds of functions are called event handlers.

```
x = 350
```

```
y = 400
```

```
def setup():
```

```
  size(800, 600)
```

```
  def draw():
```

```
    global x
```

```
    global y
```

```
    fill(0, 0, 255) # blue
```

```
    ellipse(x, y, 25, 25)
```

```
    triangle(x, y-18, x-20, y+10, x+20, y+10)
```

```
    # thrusters
```

```
    fill(255,255,0)
```

```
    ellipse(x-10, y+10, 5, 5)
```

```
ellipse(x+10, y+10, 5, 5)
```

```
def keyPressed():
```

```
    global y
```

```
    # move up the screen
```

```
    y -= 1
```

Note that we never call the `keyPressed()` function, it is automatically called when you press a key.

If you try this, you'll see it works exactly the same as the previous example using the `keyPressed` variable, which might leave you wondering why both exist. The reason is due to timing. In our original program, the `keyPressed` Boolean is only checked when the program reaches the `if` statement, whereas the `keyPressed()` function is used *when* we press the key. For most programs this distinction won't matter, but if your program was very intensive, it may take some time to reach the `if` statement, which could result in a laggy program. There is also an argument for the function keeping key-pressing code nicely separate from the rest of the program so you know where to find it. It's up to you though, really.

As well as `keyPressed` and `keyPressed()` there are also other types of keyboard events, `keyTyped()` and `keyReleased()`.

keyTyped()

The `keyTyped()` function (note that there is no `keyTyped` variable) is basically the same as `keyPressed()` except that it only works for keys that you could actually type. So it won't register Alt, Ctrl, Shift or arrow keys,

for example, but will register for Enter, Space and any alphanumeric keys. It is an easy change to make to our program so you can try it:

```
def keyTyped():
```

```
# move up the screen
```

```
Y -= 1
```

keyReleased()

You might be able to guess by now what the `keyReleased()` function is for - it detects when a key is released (having already been pressed). We might, for example, want our spaceship to change colour when we release a key.

```
x = 350
```

```
y = 400
```

```
ship_colour = color(0, 0, 255) # blue
```

```
def setup():
```

```
size(800, 600)
```

```
def draw():
```

```
global x
```

```
global y
```

```
global ship_colour
```

```
background(0) # black
```

```
fill(ship_colour)
```

```
ellipse(x, y, 25, 25)
```

```
triangle(x, y-18, x-20, y+10, x+20, y+10)
```

```

# thrusters

fill(255,255,0)

ellipse(x-10, y+10, 5, 5)

ellipse(x+10, y+10, 5, 5)

def keyPressed():

    global y

    # move up the screen

    y -= 1

def keyReleased():

    global ship_colour

    ship_colour = color(random(255), random(255), random(255))

```




Figure 146: Changing colour with keyReleased()

This will trigger the keyReleased() function when a key is released and change the colour of the ship. You may notice that, if you keep your finger on the key, it will not change colour as it will only detect this when you let go.

Key Event Variables

There are some additional built-in variables you can use in your programs that allow you define what action is being taken, to detect which key is being used, rather than having pressed any key as we have up to now.

key

The 'key' variable stores *which* key is being pressed, typed or released, so we can use this to define different behaviour depending on the key by using an if statement, for example:

```
if key == 'z':  
  
    # do something
```

The key variable will always contain key most recently used. This only works for keys that you might type (so keys that can be typed, rather than just pressed).

We might use this to rotate our ship, using the z (rotate left) and x (rotate right) keys

```
x = 350  
  
y = 400  
  
ship_colour = color(0, 0, 255) # blue  
  
rotation = 0.0  
  
def setup():  
  
    size(800, 600)  
  
    def draw():  
  
        global x  
  
        global y  
  
        global ship_colour  
  
        global rotation  
  
        background(0) # black
```

```
# only rotate this bit

pushMatrix()

# rotate round centre of triangle

translate(x, y)

rotate(rotation)

translate(-x, -y)

# ship

fill(ship_colour)

ellipse(x, y, 25, 25)

triangle(x, y-18, x-20, y+10, x+20, y+10)

# thrusters

fill(255,255,0)

ellipse(x-10, y+10, 5, 5)

ellipse(x+10, y+10, 5, 5)

popMatrix()

# end rotate

def keyPressed():

    global rotation

    if key == 'z':

        # rotate clockwise

        rotation += 0.1

    elif key == 'x':
```

```
# rotate anti-clockwise
```

```
rotation -= 0.1
```

```
else:
```

```
global y
```

```
# move up the screen
```

```
y -= 1
```

This enhances our spaceship code by adding a check on the key variable to check which key is being pressed. When you press specific keys, z or x in this example, the spaceship now rotates by changing a new variable we added called rotation. If any other key is pressed, it still moves up the screen.

The rotation is performed inside pushMatrix() and popMatrix() so that we can use translate() to locate the centre of the triangle and rotate around that, but not rotate anything else on the screen.



Figure 147: Rotating spaceship

keyCode

You might be thinking ‘wouldn’t it be better to use arrow keys’ instead of some random x and z keys?’. You may well be right! To do this, we need another built-in variable, keyCode. This allows you to use arrow keys, delete, shift, control, alt etc. These are called ‘coded’ keys. For example:

```
if keyCode == UP:
```

You can, optionally, also check that the key being pressed is a coded one using the key variable:

```
if key == CODED:
```

This allows you to separate the coded keys from the others if needed.

We can use this to control our spaceship better.

```
def keyPressed():
```

```
    global rotation
```

```
    global y
```

```
    global x
```

```
    if key == CODED:
```

```
        if keyCode == UP:
```

```
            y -= 1 # up
```

```
        elif keyCode == DOWN:
```

```
            y += 1 # down
```

```
        elif keyCode == LEFT:
```

```
            # rotate anti-clockwise
```

```
            rotation -= 0.1
```

```
        elif keyCode == RIGHT:
```

```
            # rotate clockwise
```

```
            rotation += 0.1
```

If you add this into the code you should now be able to control the direction and rotation of the ship.

Mouse Events

As you would expect, mouse events detect mouse movement and button usage. Similarly to keyboard events, there are some built-in variables and some functions you can define.

Mouse Event Actions

mousePressed

The mousePressed variable is like keyPressed, except for mouse buttons. It will be set to True if a mouse button is pressed, and False if it is not. This can be any mouse button. It has the same potential issue as keyPressed in that it will only be detected when the program gets to that line of code.

Let's add to our spaceship code by demonstrating mouse events with a second spaceship. This spaceship has a special ability though, it can cloak itself so it can't be seen!

```
x_2 = 100
```

```
y_2 = 100
```

```
ship_2_colour = color(255, 68, 0) # orange
```

```
rotation_2 = 0.0
```

```
cloaked = False
```

The cloaking device starts as False so we can see it.

```

if not cloaked:

# mouse ship

fill(ship_2_colour)

ellipse(x_2, y_2, 25, 25)

triangle(x_2, y_2 + 18, x_2 - 20, y_2 - 10, x_2 + 20, y_2 - 10)

# thrusters

fill(255, 255, 0)

ellipse(x_2+10, y_2-10, 5, 5)

ellipse(x_2-10, y_2-10, 5, 5)

```

The code above, when added to the draw() function, will draw the new ship we have called ‘mouse ship’. Note the use of:

```

if not cloaked:

```

So, the ship is drawn if the cloaked variable is False because ‘not’ reverses the Boolean value of the cloaked variable.



Figure 148: Two spaceships

We can then just add a check on the mousePressed variable to set the cloaked variable:

```

if mousePressed:

cloaked = True

```

mousePressed()

As you'd expect, there is also a `mousePressed()` function that can be used. You can replace the above code to use the function if you wish which does the same thing.

```
def mousePressed():
```

```
    global cloaked
```

```
    cloaked = True
```

`mouseReleased()`

The `mouseReleased()` function, as you might expect, is triggered when the mouse button is released, similarly to `keyReleased()`. We can add one of these to deactivate the cloaking device, so the ship will only remain cloaked while the mouse button is pressed.

```
def mouseReleased():
```

```
    global cloaked
```

```
    cloaked = False
```

`mouseClicked()`

The `mouseClicked()` function might be a little less obvious. It is basically a combination of `mousePressed()` and `mouseReleased()` but is triggered when you click a mouse button, rather than press and hold it.

This means we could replace our previous `mousePressed()` and `mouseReleased()` code into one function that means we don't have to hold down the button.

```
def mouseClicked():
```

```
global cloaked
```

```
cloaked = not cloaked
```

This will detect a mouse button click and set the cloaked variable to the opposite of what it is, often know as a ‘toggle’, using Boolean logic. So, if cloaked is True, then not True is False, and vice versa.

mouseWheel()

The mouseWheel() function detects scrolling of the mouse wheel. This one is a little more complicated as it has a variable that contains the mouse event itself:

```
def mouseWheel(event):
```

The mouse event, denoted by the variable name ‘event’ in this example, contains details of the event that was detected. On this case we can use it to retrieve how much the wheel was scrolled using its getCount() function:

```
def mouseWheel(event):
```

```
    global rotation_2
```

```
    rotation_2 += event.getCount()
```

If we now surround the mouse ship with the same rotation code we did for the keyboard ship, we can rotate the ship just using the mouse wheel.

Our entire code, so far, looks like the code below:

```
x = 350
```

```
y = 400
```



```
ship_colour = color(0, 0, 255) # blue

rotation = 0.0

x_2 = 100

y_2 = 100

ship_2_colour = color(255, 68, 0) # orange

rotation_2 = 0.0

cloaked = False

def setup():

    size(800, 600)

    def draw():

        # keyboard ship

        global x

        global y

        global ship_colour

        global rotation

        background(0) # black

        # only rotate this bit

        pushMatrix()

        # rotate round centre of triangle

        translate(x, y)

        rotate(rotation)

        translate(-x, -y)
```

```
# ship

fill(ship_colour)

ellipse(x, y, 25, 25)

triangle(x, y-18, x-20, y+10, x+20, y+10)

# thrusters

fill(255,255,0)

ellipse(x-10, y+10, 5, 5)

ellipse(x+10, y+10, 5, 5)

popMatrix()

# end rotate

# mouse ship

global x_2

global y_2

global ship_2_colour

global cloaked

if not cloaked:

# mouse ship

# only rotate this bit

pushMatrix()

# rotate round centre of triangle

translate(x_2, y_2)

rotate(rotation_2)
```

```
translate(-x_2, -y_2)

# ship

fill(ship_2_colour)

ellipse(x_2, y_2, 25, 25)

triangle(x_2, y_2 + 18, x_2 - 20, y_2 - 10, x_2 + 20, y_2 - 10)

# thrusters

fill(255, 255, 0)

ellipse(x_2+10, y_2-10, 5, 5)

ellipse(x_2-10, y_2-10, 5, 5)

popMatrix()

# end rotate

def mouseClicked():

    global cloaked

    cloaked = not cloaked

def mouseWheel(event):

    global rotation_2

    rotation_2 += event.getCount()

def keyPressed():

    global rotation

    global y

    global x

    if key == CODED:
```

```
if keyCode == UP:

    y -= 1 # up

elif keyCode == DOWN:

    y += 1 # down

elif keyCode == LEFT:

    # rotate anti-clockwise

    rotation -= 0.1

elif keyCode == RIGHT:

    # rotate clockwise

    rotation += 0.1
```

We now have 2 ships, one controlled by the keyboard, and one controlled by the mouse, albeit it can only cloak and rotate at present.

The other functions can also utilise the mouse event variable if needed. For example, to capture the number of mouse clicks.



Figure 149: Mouse and keyboard controlled ships

Mouse Event Variables

As well as the mousePressed variable, there are other built-in variables that can be used.

mouseButton

Just as we could detect which key was being pressed, we can do the same with mouse buttons using the `mouseButton` variable but with fewer options. We can detect the RIGHT, LEFT or CENTER mouse button.

```
def mouseClicked():  
  
    global cloaked  
  
    if mouseButton == RIGHT:  
  
        cloaked = not cloaked
```

So now we have defined our cloaking button as being the right mouse button. As we haven't defined the other buttons, they won't do anything.

mouseX and mouseY

The `mouseX` and `mouseY` variables capture the current location of the mouse pointer using x- and y-coordinates. These can be used anywhere. If used in the `draw()` function it would be wherever the mouse pointer is at that point. If used in an event function, `mouseClicked()` for example, it would be the coordinates of the mouse pointer when the button was clicked.

Mouse Movement Functions

As well as functions for mouse buttons, there are also functions for mouse movement.

mouseMoved()

The `mouseMoved()` function is triggered whenever the mouse is moved, which could be a lot, so you should be careful what you put in this function so that your program doesn't slow down too much.