

**CompSoc Presents**  
**LEARN TO CODE**



**Created by:**  
**Amanda Deacon**  
**Emily Price**  
**and Caitlin Duncan**  
**2016**

## A quick introduction...

Welcome to CompSoc's second workshop of the year! We're very excited to be able to present our first ever workshop on learning to code. We hope that you find this workshop interesting, enlightening, educational, or all of the above!

We understand that learning something new takes time, and by no means are you expected to be a pro by the end of this session. Don't feel like you HAVE to complete this entire document by the end of this workshop, if you do we will be extremely impressed! The one thing we do ask is that you work through it in the order we've laid it out. Each section builds on the previous ones, and the Python section assumes you've already worked through the Scratch section.

## PART ONE - SCRATCH

### Getting Started with Scratch

#### *What is Scratch?*

Scratch is, simply put: a fun, visual programming language. Scratch is a great first language as it allows you to learn about programming without the hassle of memorising how to type confusing and sometimes counter-intuitive commands.

#### *How do I open it?*

For this tutorial we'll be using the online version of Scratch. The lab computers in here do have Scratch installed, but this is an older version. For sake of everyone's sanity (and a more enjoyable experience), we encourage you to use the online version, available at the following link: <https://scratch.mit.edu/projects/editor/>

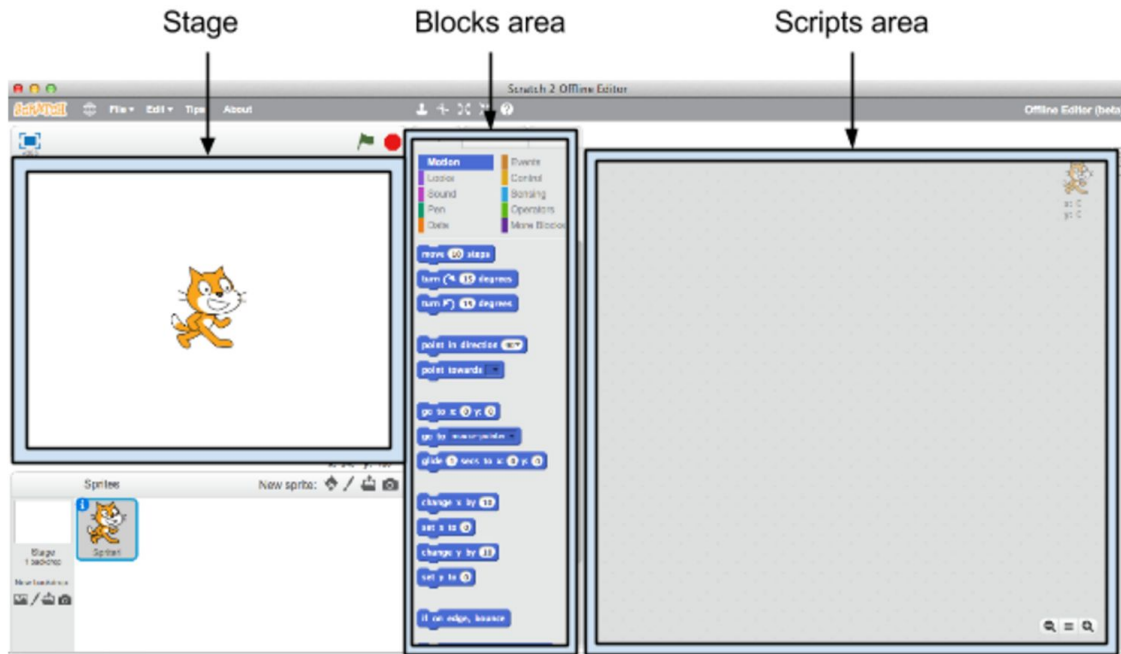
If you want to save any programs you create you can download them from the website, or if you're really keen you can create a Scratch account and save them there.

#### *How do I use it?*

The best part about Scratch is that you only have to do minimal typing. It's an interactive, Lego-type programming language: You snap the bricks together to make a beautiful creation.

Scratch has three major areas (see picture below):

1. **The Stage** - On the left hand side of the screen. This is where we see our code brought to life.
2. **The Blocks area** - Drag and drop blocks for building up a program.
3. **The Scripts area** - The drawing board for our programs. This is where you drag the blocks to in order to snap them together. Also known as where the magic happens.



When you open up the Scratch editor, you'll see Scratchy the Cat on the stage area. Let's see what Scratchy can do.

## Blocks

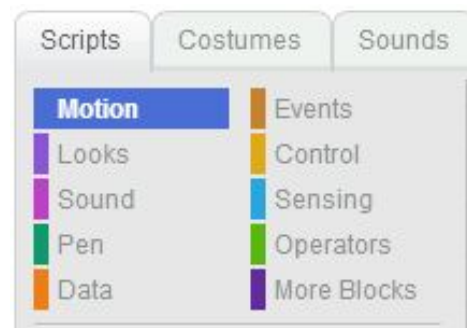
Programming languages are a lot like natural (spoken) languages. When we speak, we combine words together to make sentences, so that we can express more complex ideas. In programming, we take small ideas, and combine them to create something bigger and more awesome, like games, apps, websites, and animations.

In Scratch, these small ideas are called “blocks”, and a combination of blocks is called a “script”. There are lots of different blocks, and they are colour-coded according to what they do. A block can be a command like “*move 10 steps*”, or “*say Hello!*”, or “*switch costume to tuxedo*”. A block can also be something which controls the flow of the program (by pausing, looping, or skipping), or which senses events (such as mouse clicks or key presses).

### Blocks Task 1

Have a look at Blocks area - this is the central column in the Scratch editor. Blocks are grouped together into categories, such as “Motion”, “Looks”, and “Sound”. Click on a few of these different categories, and have a look at the blocks they contain.

Note: A lot of blocks can seem confusing at first, so don't worry! Aim to get a general overview of the area, rather than understanding all the details.



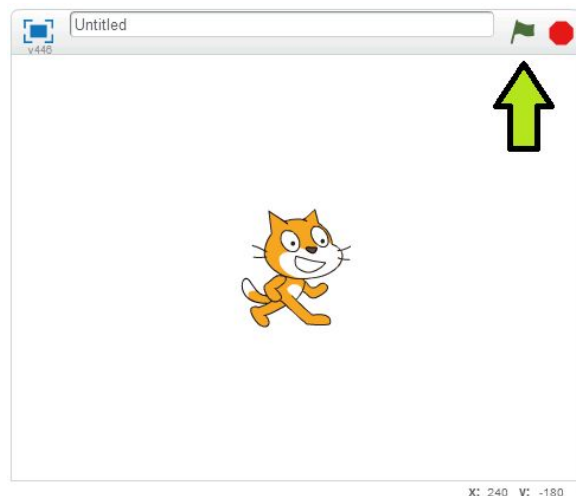
## Blocks Task 2 - Hello World

Let's make something happen!

- Select the **Events** category in the Blocks area. Click on the “when <green flag> clicked” block, and drag it over to the Scripts area.
- Now select the **Looks** category in the Blocks area. Place the “say Hello! for 2 secs” block in the Scripts area. We can change the “Hello!” text into something different, like “Hello World!”, or “Cats are awesome!!!!”.
- Stick these two blocks together, like jigsaw pieces. You should now have something like this:



- In the Stage area, click on the green flag to see your script in action.



## Blocks Task 3 - Spacebar

Great! Now we have a piece of code which will run (or “execute”) when we click on the green flag. Try to alter this code so that instead, it runs when we press the spacebar.

Hint: Look in the Events category of the block area.

## Input and Output

Loads of computer programs are interactive. We can give **input** to a program by typing or clicking, and we can also get **output** back from the program. Often, our input can be stored and used again later. The Sensing category of blocks is great for interactions.

### I/O Task 1 - What's your name?

Write a script where Scratchy asks for your name, and then repeats it back to you.

What do we need for this to happen? Let's break it down into steps. First, Scratchy will ask us a question, and we'll type an answer (**input**). Then, Scratchy will need to remember our name and use it in a reply.

Have a look at the Sensing category of blocks. Underneath the "ask What's your name?" and wait" block, there's another small block, "answer". Click on the check mark next to this. Later, when run the script and answer the question, our name will get stored in this *answer* block.



Here are all the blocks you'll need. Try rearranging them to get the script working:



Note that the "say" block is exactly the same as the "say hello" block with all its text removed. Also note that the "answer" block isn't shaped like a regular puzzle piece - hmm, it looks like it'll need to be placed **inside** one of the other blocks!

Hang onto this script - we'll be re-visiting it in the next section.

## I/O Task 2

Now we're going to make your program a bit smarter! Normally when you introduce yourself to someone they don't just repeat it back to you do they? Instead we'll make Scratchy nice and polite.

We can use another block called "join" to stick some text together with the name that's stored in the "answer" block.



You can find "join" in the "operators" tab. By adding the "join" block and moving the "answer" block see if you can get Scratchy to say "Hello ----" where ---- is your name. Or make Scratchy say "Good morning", "g-day", or any other greeting you prefer.

## Variables

So far, we've used the *answer* block to remember a bit of information. But, this block can only store the most recent answer to a question. What if we want to store more than just one thing at a time? And what if we want to store different kinds of information?

Here we find ourselves in the magical world of **variables**. A variable is like a box used to store a piece of information. We write a helpful description on the box (e.g. `user_name`), and then we store a value inside the box (e.g. "Caitlin"). Later on, if we need the information again, we can retrieve it by looking inside the box.

In a given program, we might want to store:

- **Words and sentences** - We call these "strings".
- **Numbers** e.g. Game scores, the results of calculations.
- **Files**
- **On / Off switches** e.g. For LEDs and speakers.

### *Variables Task 1 - What's your name? Enhanced!*

Before, we made a script where Scratchy asked for our name. Let's store our answer in a new variable.

Look at the Data category in the Blocks area. Make a new variable called "user\_name".



Now, in the Data category, we have a new "user\_name" block to use! The Blocks area will contain some new blocks:



Choose the "set ..." block and drop it into the Script area. We want to store the value from "answer" inside this new "user\_name" block. See if you can get your script looking like this:



Great! Now, we've captured our name within "user\_name", and other parts of script can access this value. But, hang on a second! Is this *really* any different to what we were doing before? Good question!

Remember how "answer" can only store the single most recent answer to a question? Now, we can grab that answer and store it in its own variable name. And then, we could ask *another* question, and store *that* answer too.

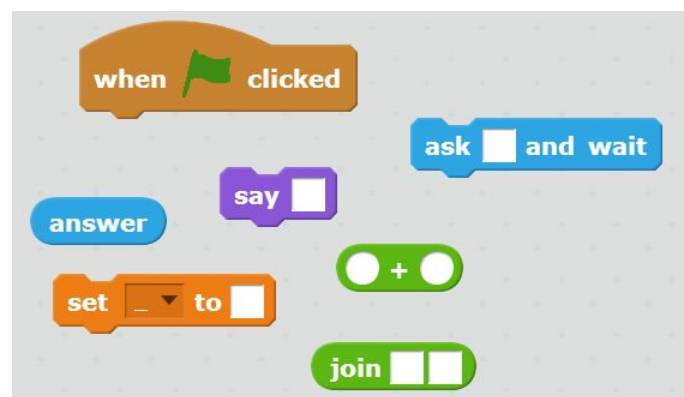
This is awesome! Computer programs would be incredibly restricted if they could only ever store one piece of information at a time. Imagine if you were making a game, and could only store either the player's name, or the player's score, but not both.

### Variables Task 2

We can store more than just words in our variables though, we can also put in numbers! With this program you should be able to see why we need variables, and just using the answer block isn't enough.

Scratchy is going to do some maths for us and add up two numbers the user types in, we'll sort of be making them into a very simple little calculator! Have Scratchy asks the user "Enter the first number", and then after the user has typed their number in ask "Enter the second number". Scratchy should then tell the user what these two numbers add up to e.g. the user enters 2 and then 4, Scratchy will say "The sum of those numbers is 6".

Here are the blocks you'll need, you'll need more than one of some of them, and you'll need to create some variables too!



Time for an extension challenge! Have Scratchy tell you the sum of the numbers, and then the square root of that sum!

## Loops

Let's say we've created a script where Scratchy jumps forward, meows, and jumps backward. Something like this:



What if we wanted Scratchy to jump and down multiple times? We could do this:



Or, we could be a bit sneaky... In programming, we can **loop** over chunks of code. Have a look in the Control category of the Blocks area. We can take the "repeat 10" block, and wrap it around a section of our code:

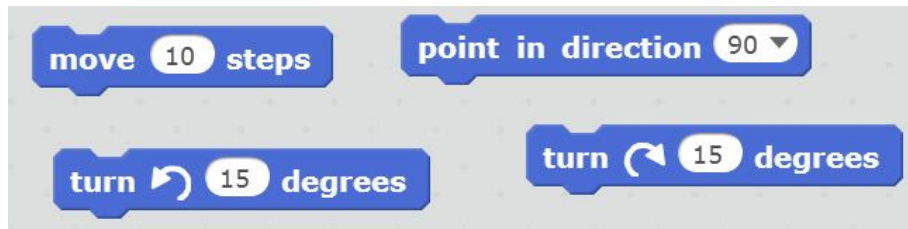


Run this code and see what happens. Loops mean you can make your programs do a lot more without having to write A LOT more code!



### Loops Task 1

Let's have Scratchy do a dance. Choose 2-4 of these "motion" blocks and make a simple dance for Scratchy.



Now you have the basic steps, have Scratchy repeat this 10 times so it looks like a proper dance!

### Loops Task 2

Now try changing your code so that Scratchy asks the user how many times they should do the dance, and then repeat it that number of times!

## Conditionals

Sometimes we need to be able to compare our variables against something. For example: checking if the answer to a question is "Yes"; or checking to see if a player's score in a game is greater than 10. We want to be able to control which parts of our code are run, and which parts get skipped over. This is where we start using **conditional** pieces of code.

Have a think about the following script:

- Scratchy asks the user "Do you love pancakes?". **If** the user enters "Yes", then Scratchy says "Woohoo!!!".

Here, Scratchy should only cheer if a certain condition has been met (i.e. the user says "Yes"). If the user says something else, then sadly, no cheering happens. Ok, first let's learn how to make comparisons or checks. Look in the **Operators** category of the Blocks area.

### Mathematical operators

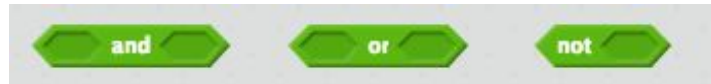


With our pancakes scenario, we want to check whether the user's answer to Scratchy's question equals "Yes". Our code will look like this:



## Boolean operators

We also have “boolean operators”. These are great when you want to link multiple conditions together. We won’t go into detail here, but talk to a tutor if you’re keen to learn more.



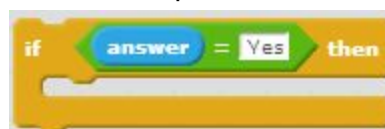
## (Back to) Conditionals

Great, so now we can now check our “*answer*” variable. **If** the answer is what we want, **then** we’ll run the code which makes Scratchy say “Woohoo!”. Otherwise, we won’t run any code.

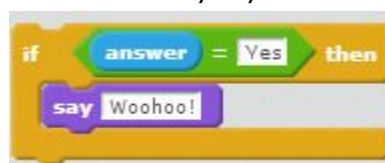
Check out these blocks within the Control category:



The first block gives us exactly what we need. It has an “if ... *<condition>* ... then” structure. We can take our condition from before, and plonk it in like so:



Now, we insert the code which makes Scratchy say “Woohoo!”:



And we’re almost there! Here’s an example of what the complete code might look like:



## Project time!

Phew! We've covered a lot of ground. Take a break, stretch your legs, rest your eyes!

When you're ready, would you like to try creating a game in Scratch? In the Scratch Editor, on the far right-hand side, you'll see a tutorial pane (if it isn't open, click on the small question mark in the top right corner). Select "All tips" at the top, and you'll see a list of projects you can try out. We suggest that the best way to practice your newfound skills is by trying out one of the games (Pong, Catch, Hide-and-Seek), or the Virtual Pet!

## PIZZA BREAK!

We plan to break halfway for pizza and some socialising, however if you zoom through the first section, or take it a little slower, all good! We want to help you work at your own pace. And no matter what point you're up to, when we say it's time for pizza, that includes you.

# PART TWO - PYTHON

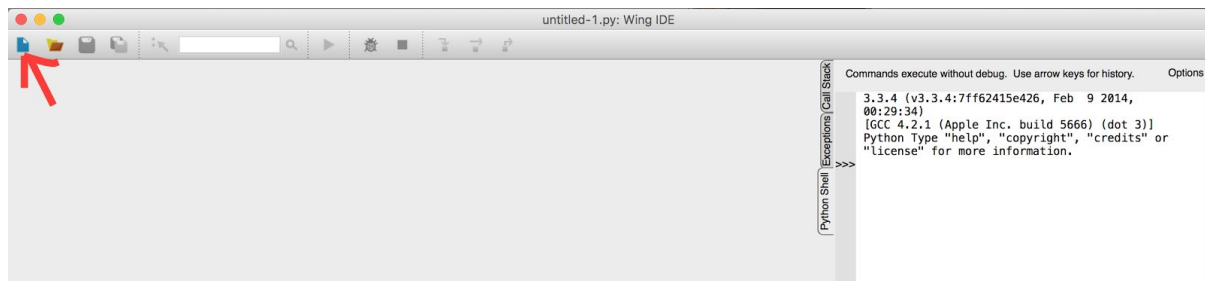
## Getting Started with Python

The previous section covered a lot of key concepts. These *concepts* don't change when you change programming languages. What does change is the way you create the code, and the way the computer understands it.

Python is different to Scratch as you type the commands rather than dragging and dropping. However, you still build up your programs - with theoretical blocks rather than literal ones :)

The most important thing to remember about writing Python is that spacing matters. We will go into detail about what spacing to use and when, in each section.

So, if you want to experiment with Python, you're going to need to jump on a lab computer. Open up the list of all your programs and search for "Wing". We want to open the program that has a picture of a feather, and is named something like "Wing 101" or "Wing IDE".



This is what most of your screen should look something like. To start a new code file, we're gonna click on the leftmost blue icon, the one with a red arrow pointing at it.

You should now be able to type on the left hand side of the screen. This is where we'll write our code!

The right hand side of the screen is where your code will show its *outputs*. We'll explain this in more detail very soon.

To run your code, you'll want to click the little green play arrow near the top of the screen.

If you're using a computer that doesn't have Python installed, or you want to work through more examples than we give in this tutorial then head to [www.codecademy.com/learn/python](http://www.codecademy.com/learn/python) and you can write Python without having to install it on your computer. If you want to keep practicing though then you can install python here: [www.python.org/downloads/](http://www.python.org/downloads/), and Wing 101 here: [wingware.com/downloads/wingide-101](http://wingware.com/downloads/wingide-101)

## Input and Output

The major way that we will be interacting with input and output in this workshop is through the print statement. We use the print statement by typing the word `print` followed by a pair of brackets `()`. Inside the brackets, we put whatever it is we want to show.

### Output Task 1

We're going to write a line of code that shows our name in the output area of the screen. For example, if I wanted to introduce myself, my code would look something like this:

```
print("Hi, my name is Emily")
```

Try it out with your name. Or, whatever else you want to try writing!

**Tip:** We need to surround our text with the double quote marks (or speech marks) so that Python knows we're telling it *say this exactly*. Otherwise it might get confused and think you're trying to show variables - those are coming up next.

## Variables

In Python, you create variables like this:

1. `variable_number = 12`
2. `variable_string = "hello"`
3. `variable_boolean = True`

This is called an assignment statement. In these lines, you are 'assigning' the value on the right hand side of the equals to be referred to by the thing on the left hand side of the equals. To refer back to our previous box analogy, you are *labelling* the box with the text on the left of the equals sign, and whatever's on the right of the equals sign is what goes *in* the box.

The examples also do a nice job of demonstrating three different, important variable types. These are numbers, strings, and booleans.

### Numbers

Fairly self-explanatory, this is any variable where you're storing any kind of number. Whether that's integers (1, 2, 3...) or decimals (0.001, 0.01, 0.1...)

### Strings

A "string" can be thought of as text. This can be one word, or it can be a bunch of words. Regardless, a string is always surrounded by quotation marks - as shown in example 2.

## Booleans

Booleans are a straightforward variable with a confusing name. Booleans are True or False - always typed with a capital first letter, no quote marks necessary.

## Lists

Another important variable type is a list. A list stores several values instead of only one - just like a real world list. For example, if you wanted to store your to-do list in a Python variable, it might look something like this:

```
todo = ["Take the cat for a walk",
        "Watch Keeping up with the Kardashians",
        "Organise wardrobe by colour",
        "Take over the world"]
```

As you can see, each value is separated from the others by a comma, and all the values are wrapped up together inside a pair of square brackets: []

## Variable Naming

You may be wondering why many of the variable names that we've shown have `_` characters in them. There are three major rules to follow when thinking of variable names in Python:

1. Your variable name **must** start with a letter (a through z),
2. After this, your variable name must consist only of letters, underscores, and numbers
3. Variable names are case sensitive - `VARIABLE_NAME` is *not* the same as `variable_name`

As an example of using variables, what would you expect to happen after the following two lines of code?

```
variable_number = 12
print(variable_number)
```

Give it a go and see what happens...

## Variables Task 1

We're going to use variables to complete a simple maths equation. Let's say we want to display the answer to the equation  $3 * 5$ .

First of all, we can set the following variable:

```
answer = 3 * 5
```

Now that we have the value we want stored in the box labelled “answer”, all we have to do is tell Python to print the answer:

```
print(answer)
```

And you should get the result 15!

### Variables Task 2

Use *variables* to print the phrase “Hello, World!”. This means you *cannot* use the following line of code:

```
print("Hello, World!")
```

**Tip:** If you can't type “Hello, World!” directly into print's brackets, think about how else you can get it there...

### Variables Task 3

Implement the necessary variables to make the following statement output 12:

```
print(first_num * second_num)
```

**Tip:** There's several different options for this. Think about your times tables!

### Variables Task 4

Create a list of cupcakes! Remember to use the square brackets [ ] and name the list variable something useful. This list will be used for a future task...

## Conditions and Conditionals

Conditions in Python work pretty similarly to those in Scratch. Here's some of the syntax...

### Mathematical Conditions

Each of the following conditions will be either true or false. Can you tell which ones are true and which are false?

1.  $10 < 5$
2.  $3 == (9 / 3)$
3.  $4 != 6$

You can check your answers with the following line:

```
print(the equation goes here)
```

### Boolean Conditions

The same boolean conditions apply in Python as in Scratch. And their use is almost exactly the same:

```
this and that
```

```
this or that
```

```
not this
```

Where `this` and `that` are variables or expressions.

## Python Conditional Statements

Using the above conditions, we can build conditional statements.

```
if condition:
    do something
elif another_condition:
    do something else
else:
    do something different
```

Where `condition` and `another_condition` are things that are either true or false.

Although the example has “if”, “elif”, and “else” cases, a Python conditional does not need to have all of these. A conditional can be:

- An “if” statement by itself
- An “if” statement with an “else” statement
- An “if” statement with **any number** of “elif” statements
- An “if” statement, followed by any number of “elif” statements, followed by a single “else” statement

## Loops

In Python there are two major types of loop: For loops, and While loops.

For loops are good for when you want to consider each thing in a group of things: some examples are the letters in a string, or the items in a list.

While loops are good when you want to do something so long as a certain condition holds true. For example, repeat something while a numeric variable is below a certain value.

Loop syntax is as follows:

### For Loops

```
for thing in group:
    do something with thing
```



## While Loops

```
i = 1
while i < 11 :
    use i in some way
    i = i + 1
```

You'll see above that loops follow a certain convention. Stuff that you want to happen as part of the loop (the **body** of the loop):

1. Comes after a colon
2. Starts on a new line
3. Is indented

To continue writing code outside of the loop, ie only happens once, we want to *dedent* our code. This means removing the whitespace in front of the code.

```
i = 1
while i < 5:
    print("Times run through the loop: " + i)
    i = i + 1
print("I am now outside the loop!")
```

## Loops Task 1

To start off with, we're going to use a "while" loop to print the numbers from 1 - 10. To do this, we'll start off with a variable "counter", which will count which number of the loop we're on. We'll start this variable off with a value of 1.

```
counter = 1
```

Next, we need to create a while loop with the right condition. In this case, we want to keep using our loop until our counter is bigger than ten. So we can say:

```
while counter <= 10:
```

Next, we want to print each number...

```
print(counter)
```

... and move onto the next number:

```
counter = counter + 1
```

Everything going to plan, Your output should look something like this:

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

**Tip:** Don't forget to follow the rules for indenting your code!

### Loops Task 2

Now for some maths! Yay maths!

It's up to you to write a loop that prints out the *square* of each number from 1 to 10.

If you need a refresher, the square of a number is a number times itself. For example,  $5 \times 5 = 25$ . So the square of 5 is 25.

**Tip:** Don't forget to increase your counter at the end of the body of the loop!

### Loops Task 3

Okay, enough of the numbers. I was taught to program by a maths teacher. He loved his mathematical examples. Please forgive me.

Remember the list of cupcakes you made earlier? Well, now we're going to eat them. Not literally. Sorry about that.

Use an appropriate loop to print the phrase "Now eating...", followed by a type of cupcake, for each cupcake in your list.

**Tip:** For this one, we *won't* be using a "while" loop.

## Functions

Functions are pretty complicated and if you're overwhelmed, we would not recommend looking into functions right now. However, if you're game, let's go!

A function is a small snippet of code that can be used multiple times, without having to copy and paste a big old chunk of code. A function has a name, optional parameters, and may return a variable. Functions are best understood in practice, so we'll explain each piece by running through the creation of a small function. Our function will return the area of a rectangle.

### Naming functions:

A function is created by typing the keyword `def` (short for define), followed by a name for the function, followed by two round brackets and a colon.

```
def area_of_rectangle():
```

This line is called a function declaration, as it *declares* that there is a function called `area_of_rectangle`.

### Taking parameters:

Inside the two round brackets of a function declaration, you can choose to pass in *parameters*. Parameters are variables that come from somewhere else, and will be used in the function. This makes functions much more versatile.

To use parameters, simply add them inside the brackets. The following declares a function that takes two parameters:

```
def area_of_rectangle(length, width):
```

So now we know our function `area_of_rectangle` will have access to variables `length` and `width`.

### Returning variables:

Quite often, you want a function to *return* something. This is usually some sort of calculation or manipulation of variables. We can then use the *returned* value later on, in another piece of code!

```
def area_of_square(length, width):  
    return length * width
```

### Calling Functions:

Say we want to print the result of our `area_of_square` function. To do this, we need to *make a call* to the function. To run the function, you need to pass it what you want the values of the parameters to be. Say we have a rectangle of length 4 and width 10.

Calling our function works as follows:

```
area_of_square(4, 10)
```

However, all this does is *run* the function. It doesn't do anything with the value it returns. A handy thing to do with functions is assign their return value to a new variable. We can then use that variable to do other things with. Assuming the area of a square function is correctly defined, what will the following code print?

```
area = area_of_square(4, 10)
print (area)
twice_area = 2 * area
print (twice_area)
```

### Functions Task 1

Remember earlier, when we calculated the square of numbers? We're going to write a function that returns the square of any number.

To start off, we want to define a function that takes in one parameter: the number to be squared.

```
def square_of_number(number):
```

Next up, we need to implement the code to calculate the square of this number:

```
    square = number * number
```

And then we return this variable:

```
    return square
```

The only thing left to do is call the function with a number:

```
result = square_of_number(12)
print(result)
```

And it should print out 144.

**Tip:** Don't forget to indent your *function*, and not the function *definition* or *call*.

### Functions Task 2

Create a function named `hello_person` that takes in a name as a parameter, and returns a string of the type "Hello, Emily". Assign the result of this function to a variable, and print it!

**Tip:** You can add strings together with the `+` sign. So, "blah" + "blah" would be "blahblah".

## WHERE TO GO FROM HERE?

So, what's next? Computer science is a massive, diverse field, full of wondrous surprises and endless fun. The programming skills we've explored today are just a small taster of the sorts of things you can do. If you enjoyed yourself and would like to learn more, check out the tips below.

### Staying Connected

#### *Study Computer Science*

If you're keen on Python and want to take it further, the university has a great introductory course. COSC121 (Introduction to Computer Programming) runs twice a year, which means you can in fact take it in Semester Two!

#### *Clubs*

If you aren't a member of CompSoc, consider signing up! The club is open to any student interested in computing, and you don't need to be a computer science or software engineering student to join. CompSoc runs regular academic, social, and industry events throughout the year. Find us here: [www.compsoc.org.nz](http://www.compsoc.org.nz)

#### *Group Projects*

Having a project is great! You get to build up your programming skills and create something awesome, like a game, animation, or website. Working in a group can help keep you motivated, and it also means people can share knowledge with each other.

Here are a few ideas:

- Make a game!
  - You could start by re-creating something familiar (like Hangman, or Solitaire).
  - Or come up with something new! Try mashing two different genres together to come up with unique ideas (like "Super Tic-Tac-Toe RPG").
- Make an animation!
  - Create a story of your own.
  - Choose a scene from a favourite book or movie, and animate it in your own style.

### Resources

There are plenty of resources out there for learning more about programming. And a lot of these are free to use! Here are just a few.

Websites:

- **Code Club** ([www.codeclub.nz](http://www.codeclub.nz)) has resources for continuing to learn Scratch.

- **Interactive Python** ([www.interactivepython.org](http://www.interactivepython.org)) contains awesome interactive textbooks. Check out “*How to Think Like a Computer Scientist*”.
- **Code Academy** ([www.codecademy.com](http://www.codecademy.com)) offers interactive courses in several different programming languages.

Books:

- The university’s Engineering and Physical Sciences Library has a huge amount resources for computer science and software engineering.

## Feedback

Thanks for coming to the workshop today! We hope you had a great time. Please take a couple of minutes to fill out this feedback form (<http://goo.gl/forms/X4vaPoLvBkl0A1vn1>). Let us know what you liked, what you didn’t like quite so much, and what you’d like to see in the future!