Introduction

Welcome to the first session of the Code First: Girls Python course! I'm really gald that you can be here.

Over the next eight weeks you're going to learn about a programming language called Python. You'll cover the basics of programming with Python, try a few programming challenges, build a project with a few team mates, and hopefully have some fun along the way.

Never written a program before in your life? No problem. The course is for complete beginners and assumes no prior knowledge. Through the course we'll guide you to build up your knowledge and confidence of programming.

The course is split up into eight sessions. The first four sessions are focussed entirely on introducing Python. Your instructors will introduce core concepts to you and support you through a series of programming exercises.

The course handbook is here to support you, providing written explanations of what you learn in each session. It also has some extra useful bits of info if you want to go into more depth.

There's also optional homework, which is designed to reinforce what you've learned in the taught sessions.

The final four sessions are all dedicated to your team projects. You'll work in groups of two or three people to create simple applications. Focussing on a project is a great way to develop your programming skills. In the final week you'll get the opportunity to present your projects to the other teams and maybe win a prize.

You can see a brief overview of what you'll learn in each session below.

In the next part of the handook you'll see how to set up your computer ready for the first session.

Course Content

Session 1

The first session is a gentle introduction to the basics of Python. Through short exercises you will become familiar with variables, data-types, and the PyCharm IDE. You are also taught how to read error messages, which will help you fix your programs when they aren't working.

1. Running Python with files and the console
2. Data-types (Integers, Floats and Strings)
3. Maths operations
4. Understanding error messages
5. Variables
6. User input

Session 2

The second session focuses on building problem solving skills through drawing. The Turtle library is used to create basic drawings with Python. Using this library you will be introduced to lists, loops, and functions.

1. Importing other libraries
2. Problem solving with Turtle
3. Lists
4. For Loops
5. Functions

Session 3

In the third session you will learn about logic and dictionaries. By using logical comparators and if statements, you can automate basic decision making in your programs.

1. Logical comparators
2. If statements
3. Dictionaries

Session 4

The fourth session is the last session that focuses only on Python. You will have covered the core concepts of programming in the first three sessions. In this session you are introduced to working with third-party libraries and APIs.

1. Pip package manager
2. APIs
3. Flask and Jinja

Session 5

Session five is about for your group projects. In the first half of the session you will be given an introduction to the most essential parts of git. In the second half you will form a group and start planning you projects and set achievable goals.

1. Git
2. Planning your project

Session 6-7

In sessions six and seven you will work as part of your team to build your project.

1. Project

Session 8

In the final session you'll have a short amount of time to make final changes to your team projects. In the second half your team will present your project to the other students.

1. Project
2. Presentations

Setup

This part of the guide will explain how to install the software required for the course. It will show you how to install:

- Python 3
- Git
- PyCharm Community Edition

I'll also show you how to create a new PyCharm project and test that everything is installed OK.

Python 3

Python is frequently updated. Some older versions of Python, such as version 2.7, are no longer supported. Even if you already have a version of Python installed you should follow these instructions to install an up to date version of Python.

In a web browser go to https://www.python.org/downloads/

Click on the button that says `Download Python 3` (the number on the button might say something like 3.7.2 instead).

The installer for Python should now download. Once the download is complete, open the installer. Tick the `Add Python 3.7 to PATH` box and click `Install Now`.

Git

Git is a tool that is used by developers to share and collaborate when writing programs. You will need to install git for sesison 5 and your group project.

To install Git, open your web browser and go to https://git-scm.com/download/. Select your operating system and click on the download link for 64-bit.

Once the installer has download, open it and follow the instructions

PyCharm

When writing Python programs there are a lot of programs you can use. Some developers like to use text editors like Sublime Text or Atom, others prefer powerful (yet complicated) editors like Vim or Emacs. For this course you will be writing and running your Python programs with PyCharm.

PyCharm is an editor that is designed specifically for Python. It comes with lots of built-in tools that help you work with Python (for example it can highlight typos in your code).

To install PyCharm, go to https://www.jetbrains.com/pycharm/download/

There are two versions of PyCharm. You will be using the free Community Edition for this course. Click the `Download` button under the Community Edition. The installer should now download.

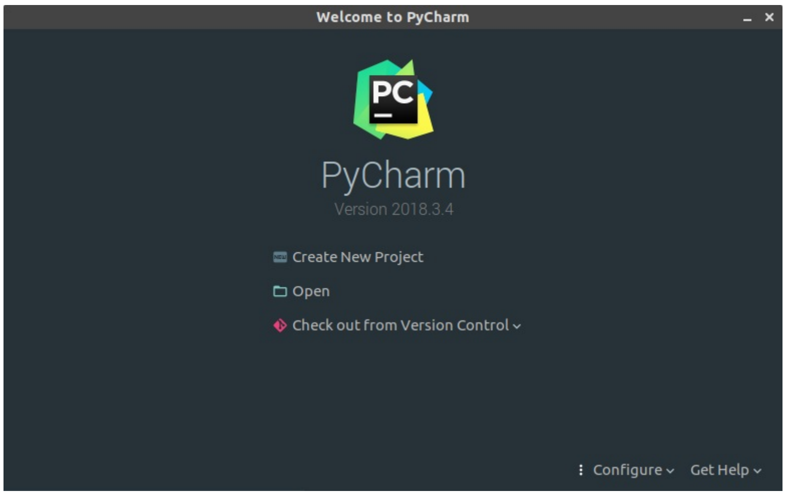
Once the installer has downloaded, run it and follow the instructions to install PyCharm.

Creating a PyCharm Project and Testing the Installations

So that you're prepared for the first session, we're going to create a new PyCharm project and check that Python has installed correctly.

> If you have any problems, double check that you've followed the installation instructions correctly. After doing this if there are still issues with the installation make sure you tell the instructors when at the very start of the first session and they should be able to help you.

Open PyCharm. You should see a window like the one below:

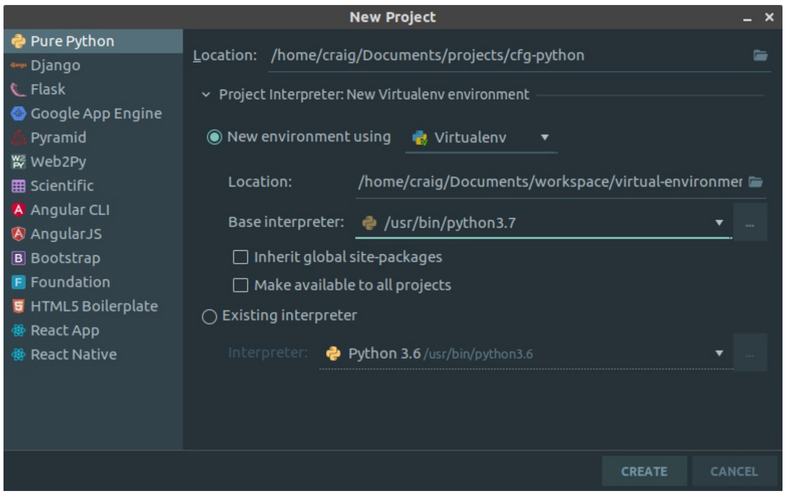


Click on the `Create New Project` button.

On the next screen in the `Location:` field name your project `cfg-python`. You need to do one more thing before clicking `Create`.

You also need to make sure you're project will use the correct version of Python. Click on the `Project Interpreter: New Virtual Environment` dropdown to see more option for your project.

In the `Base interpreter` field click the drop-down and select `Python 3.7`.



Location: /home/craig/Documents/projects/cfg-python

Project Interpreter: New Virtualenv environment

New environment using Virtualenv

Location: /home/craig/Documents/workspace/virtual-environmer

Base interpreter: /usr/bin/python3.7

Inherit global site-packages

Make available to all projects

Existing interpreter

Interpreter: Python 3.6 /usr/bin/python3.6

Click `Create` to start your project.

PyCharm will now create the project. You may need to wait a while depending on the speed of your computer. There might be a progress bar at the bottom-right of the PyCharm window, which you can follow the progress.

Once the new project is ready, it's time to check that Python is working correctly. We'll use a tool called the *terminal*.

On the menu bar at the top of the screen click on `View > Tool Windows > Terminal`.

You should see the terminal panel pop-up at the bottom of the window.

In that window type `python --version` and press enter to check which version of Python you are using. The output should look similar to this:

```
Python 3.7.2
```

> If you the output says you are using Python 2.7 or earlier you will need to check two things. First check that you downloaded and installed Python in the step earlier. If you definitely did this you may have selected the wrong option when creating your project. Go to `File > New Project` and follow the Creating a PyCharm Project instructions above

Next type `git --version` and press enter to check that Git is installed. The output should look something like this

```
git version 2.17.1
```

This is just to check the installation is OK, the version of Git is not important here.

You should now be ready for the first session. Remember, if you have any trouble with the setup instructions let your instructors know at the very start of the first session.

I hope you enjoy the course!

Session 1.

Getting Started with Python

In this session:

1. Running Python files
2. Data types
3. Maths operations
4. Understanding Error Messages
5. Variables
6. User input

Before you start Session 1, make sure that you've followed the setup instructions in the Introduction.

Why Python?

There are lots of really great programming languages out there. You may have hear of ones like Java, Ruby, JavaScript (weirdly unrelated to Java), and C. They're all languages (just like human languages such as English and French), but instead of being used to communicate from one person to another, they're used to communicate instructions to a computer.

All languages have a set of rules that specify how you write them. Each language has different rules, but there are some core concepts that are shared between them.

Programming Language: A language with a set of rules that communicates instructions to a computer

So what's so special about Python?

Python has a lot going for it. Firstly, Python is designed to be readable. The people who design the Python language have put a lot of care and effort this (so say thanks if you ever see them). One of the benefits of this is that there is a gentler learning curve for beginners compared to some other languages.

Secondly, Python has a large number of third-party libraries. "What's a library?" I hear you ask. In basic terms, a library is a collection of code that other people have written that you can reuse. This saves you a lot of time as you don't have to write that bit of the code yourself.

In Python there are libraries that are designed to build websites, analyse large amounts of data, draw pictures, hack into computers, make a velociraptor appear, and many more.

> **Library:** Reusable collection of code that someone else has written which you can use

Finally, Python is a very popular language. It's used in a diverse areas like science, machine learning, finance, motion pictures and many more. The skills that you learn as a beginner are a fundamental part of any career that uses Python.

Your First Python Program

Time to write your first Python program!

To write and run your first Python program follow each of the four steps below. Refer back to these steps whenever you want as you'll need to use them throughout the course.

Step One: First let's create a new Python file in Pycharm.

Right click on the `cfg_python` folder on the Project Tool Window and select `New > Python File`.

Step Two: In the window enter `hello` as the name and click `OK`. PyCharm will automatically add the `.py` for you.

Step Three: Now that you've create your file, let's add some Python code. In the file type the following:

```python
print('Hello, World!')
```

Make sure that `print` has no capital letters. There should also be two speech marks (one either side of the `Hello, World` text) and an opening and closing bracket.

Step Four: Now that you've written your Python code, it's time to run it!

To run you program, right click anywhere in the file and click on `Run hello`. You should see PyCharm's Run window appear with the text `Hello, World!`.

Congratulations! You've just run your first Python program.

That's cool and everything, but what's going on?

In your program there are two main bits:

1. The `print()` bit
2. The `'Hello, World!'` bit that's inside of the `print()` bit.

The `print()` bit is an example of a *function*. A function does a specific task. The task of the `print()` function is to display some data value to you, the programmer. If you removed the `print()` your program would still run OK, it just wouldn't display the output to you.

> **Function:** a pre-written set of instructions to complete a certain task

In this case the data value that is being displayed is `'Hello, World!'`. This is an example of a *string*. You'll learn about strings in more depth very soon. For the time you just need to know that strings are used to represent text in Python.

By changing the text in the string, you change the value that is output. For example I've changed the value of the string to output a description of my favourite socks:

```python
print('My favourite socks have blue diamonds')
```

Try changing the value to whatever you want!

Numbers and Operators

When writing programs you'll often want to work with numbers and do calculations on those numbers. That's what this section is all about. I'll introduce you to two (yes, two!) types of numbers and a bunch of different operators.

The first type of numbers you'll get to know are *Integers*. Integers are whole numbers without decimal points. For example `1`, `4329884` and `-63` are all integers.

The second type of numbers are called *Floats*. A float is any number that has a decimal point. For example `1.65`, `82.0` and `-9.3` are all floats.

Before you move on I want to tell you about *data types*. A data type refers to what kind of thing a piece of data is. For example `7` has an integer data type and `'Hello, World!'` has a string data type.

So far you've seen three data types: strings, integers, and floats. You'll cover a few more data types throughout the course. It's important to know what data types you're working with as different data types can have different operations performed on them.

> **Data Type**: The kind of data that determines what values it can be and what operations can be performed with it
>
> **Integer:** a Python data type for whole numbers
>
> **Float:** a Python data type for decimal numbers

Now that you've seen the two types of number data, let's take a look at *maths operators*.

A maths operator combines two number values to calculate a result. For example if I wanted to see the result of adding the integer values `5` and `17` together I could use the *addition operator* (`+`) like so:

```
print(5 + 17)
```

When I run this like of code the value `22` would be displayed.

> **Maths Operator:** An operation that calculates the result of combining two number values

I can also use float values with the addition operator. Here my code will display the result of adding the float values `8.3` and `10.12`:

```
print(8.3 + 10.12)
```

The value `18.42` should be displayed when this program is ran.

There are a handful of maths operators in Python. Let's do an exercise to find out how they behave.

Exercise 1.1: Operators in the Python Console

In this task you will use different maths operators to discover their purpose. You'll also learn about a different way to run Python code using the Python console.

Python has two main ways to run code: files and the console. You've already used files, let's take a look at the console.

When a Python file is run it will start at the top of the file and run each line one at a time. When it reaches the end of the file the program finishes running. The only way for you the programmer to see the output of the program is to use the `print()` function.

The Python Console is different to files. It immediately shows the result of a line of code without needing to use the `print()` function.

To view the Python Console, go to the top menu and click `View > Tool Window > Python Console`. Each line in the Python Console begins with `>>>`. You can type Python code in and press enter to run it. The Python Console will immediately show the result.

For example if I enter `9 + 3` on the Python Console it would look like this:

```
>>> 9 + 3
12

>>>
```

As the Python Console console immediatly shows results it is useful for quickly exploring how new bits of code work. Files on the other hand are useful when you want to run the same code multiple times.

Let's use the Python Console to explore how different Maths Operators work.

In your Python Console type the following lines of code one at a time:

```
5 - 6
8 * 9
6 / 2
5 / 0
5.0 / 2
5 % 2
2 * (10 + 3)
2 ** 4
```

Look at the output for each line. What do you think each operator does? Are there any outputs that you didn't expect?

Now that you've tried out the operators, here's a bit more detail about them.

The subtraction operator (`-`) minuses one number from another. For example, the result of `8 - 3` would be `5`.

The multiplication operator (`*`) times two numbers together. The result of `7 * 5` is `35`

To divide one number by another, the division operator (`/`) is used. For example `8 / 4` is `2` and `5 / 2` is `2.5`. Note that you will get an error if you divide by zero that looks like this `ZeroDivisionError: division by zero `.

The modulo (`%`) operator is similar to division. Instead of saying how many times one number divides by another it works out how many times the first number can be divided prefectly by the second and returns the difference. For example `4` divides perfectly by `2` twice with no remainder so the result is `0`, while `5` divides by `2` twice with a remainder or `1`, so the result is `1`.

The exponent (`**`) operator calculates the power of one number by another. For example `2 ** 3` is `8`, which is equivalent of `2 * 2 * 2`.

By default Python will evaluate operators in the following order: 1. Brackets 1. Exponent 1. Multiplication 1. Division 1. Modulo 1. Addition 1. Subtraction

Brackets can be used to change the order that calculations are done. For example the result of this calculation will be `23`:

```
2 * 10 + 3
```

By adding brackets around `(10 + 3)` the addition is now calculated first, changing the result to `26`:

```
2 * (10 + 3)
```

Strings

Remember back to a short while ago when you wrote you first Python program. The program looked like this:

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

The `'Hello, World!'` part of the program is a data-type is called a *string*. The string data-type in Python is used to represent letters, numbers, symbols and other characters.

String: a Python data type for **text** and **characters**.

All strings begin and end with either single (`''`) or double (`""`) speech marks.

For example, this is a string...

```
'Clap clap clap'
```

...and so it this...

```
"20 cheese cakes"
```

When writing strings you may sometimes forget to include the speech marks entirely. Depending on what's in your string, you will see different errors. Here I haven't included any speech marks for a single word:

```
Clap
```

I get this error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Clap' is not defined
```

The type of error is stated on the last line and is a `NameError`. Python thinks that my string is actually a variable (which you'll learn about in a short while). If you see this error message when using a string, you might have forgotten the speech marks. To fix it, just add the speech marks:

```
'Clap'
```

Here's another example with multiple words where I've forgotten the speech marks:

```
Clap clap clap
```

When I run the code in the Python console I get this error:

```
  File "<stdin>", line 1
    Clap clap clap
            ^
SyntaxError: invalid syntax
```

This time I get a `SyntaxError`. A `SyntaxError` means that I am not following Python's rules for how to arrange commands. Notice how on the fourth line the error message points to the part of the code that confused Python. Remember to look for this when you see this error message as it can help identify what you need to fix. In this case again the fix is just to put speech around the words:

```
'Clap clap clap'
```

In the previous exercise you learned about different Python operators and how they behave with number data-types. Some of these operators can be used with strings. In the next exercise you will explore how different operators work with string as well as try some special string commands called methods.

Exercise 1.2: String Operators and Methods

In your **Python console** type each of these commands:

```
"Cat"
"Cat" + " videos"

"Cat" * 3
"Cat" + 3

"Cat".upper()
"Cat".lower()

"the lord of the rings".title()
```

What is the output for each line and why?

One of the lines causes an exception. Read the exception message. What do you think it means?

Concatentation:

Method:

Dot Notation:

`str()`

Variables

You're about to be introduced to a very important concept called a *variable*. Are you ready?

Variable: A reusable label for a piece of data

A variable is a reusable label for a piece of data. A variable helps Python programs remember pieces of data so that you can reuse them multiple times in your programs.

Look at this line of Python code:

```
oranges = 12
```

When creating a variable you need three things: 1. A name for the variable 1. An equals sign 1. A value

The name of the variable is `oranges`. The value of the variable is `12`. In between the name and the value there is an equals sign (`=`), which tells Python that you are assigning a value to the variable name.

In the above example I created a variable with an integer value. You can use any data type with variables. Here's a variable called `name` that has a string value of `"Jonesy"`:

```
name = 'Jonesy'
```

Once you create a variable it can be used like any other value. Variables and values are interchangable. Wherever you can use a data value you can use a variable.

Let's say that I have 12 oranges and I want to output that. Here I've written a program to do that:

```
oranges = 12
print(oranges)
```

See how I've put the variable name `oranges` inside the `print()` function's bracket? This is the same as writing `print(12)`. Either way the output will be `12`. The benefit of using a variable is that it can be reused and updated.

I now want to calculate the cost of purchasing oranges. Each orange costs `0.5`. I want to output the total cost and say how many oranges the cost is for. Here's the code:

```
oranges = 12
cost_per_orange = 0.5

total_cost = oranges * cost_per_orange

print(str(oranges) + ' oranges')
print('costs ' + str(total_cost))
```

The output will look something like this:

```
12 oranges
costs 6.0
```

Notice how I've created the `oranges` variable on line 1 and reused it on lines 4 and 6. This means that if I change the value of the `oranges` variable, the value will update everywhere else that it is used.

Here I've changed the value of `oranges` to 20:

```python
oranges = 20
cost_per_orange = 0.5

total_cost = oranges * cost_per_orange

print(str(oranges) + ' oranges')
print('costs ' + str(total_cost))
```

The rest of the code stays the same, but the output is different:

```
20 oranges
costs 10.0
```

To understand why reusing variables is important here's the code rewritten without the `oranges` variable:

```python
cost_per_orange = 0.5

total_cost = 12 * cost_per_orange

print(str(12) + ' oranges')
print('costs ' + str(total_cost))
```

I've used the value `12` on lines 3 and 5 instead of the `oranges` variable. If I want to update the number of oranges I now have to update the code in two places instead of one.

This may seem trivial for such a short program, but when you're working with larger and more complex programs you may forget to update a value and cause the program to behave incorrectly.

Another this you may have noticed is that variables are useful for explaining the purpose of values. In the example without the `oranges` variable, you don't know what the value `12` means without additional explanation. By using a variable the name gives the value context. With a name like `oranges` you can probably tell that the `12` value means the number of oranges.

To recap, variables are labels for pieces of data. Varialbes have a name and a value and can be used wherever you can use a data value. Variables allow you to reuse pieces of data and are useful for indicating the purpose of a value to other people.

Exercise 1.3: Cat Food

In a new Python **file** called `cat_food.py`, create a program that calculates how many cans of cat food you need to feed 10 cats

Your will need: 1. A **variable** for the number of **cats** 1. A **variable** for the number of **cans** each cat eats in a day 1. A `print()` function to output the result

Extension: change the calculation to work out the amount needed for 7 days

---

String Formatting

There a few different ways to do string formatting in Python

[F-STRINGS]

User Input

Session 2.

This session:

1. Importing libraries
2. Problem solving with Turtle
3. For Loops
4. Lists
5. Functions

Python Libraries

Libraries allow you to reuse code that other people have written for a specific purpose. This can save you time and effort. In this session you'll use a library called `turtle` that can create simple drawings.

> **Library:** Structured code that someone else has written that you can reuse in your programs

With Python there are two types of libraries. The first is called a standard library and is pre-installed when you install Python. The other type of library are not part of the standard library and need to be installed separately. We'll cover this other type of library in a later session. The `turtle` library is part of the standard library so you don't need to do any more steps to install it.

The first thing that you do when using libraries with your Python program is to *import* it. This tells Python to load the library so that you can use it in your program.

This is what a basic import looks like in Python:

```python
import turtle
```

The import has two parts:

1. The `import` statement
2. The name of the library that is being imported

Now that the turtle library is imported you can start using it's functions in your program. When run, this program draws one line, turns one-hundred and thirty degrees and draws a second line. When you run the program you should see an arrow that draws the shapes. This arrow is the turtle.

```python
import turtle

turtle.forward(100)
turtle.right(130)
turtle.forward(100)

turtle.done()
```

The `turtle.forward()`, `turtle.right(130)`, and `turtle.done()` commands all start with `turtle.`. This tells Python that we want to use the command from the `turtle` library.

The `turtle.forward()` funciton makes the turtle move forward a number of pixels, leaving a line behind it. In the program above the turtle moves forward `100` pixels twice. To turn the turtle you use `turtle.right()` and specify a number of degrees. There's a also a `turtle.left()` function that turns it in the opposite direction.

The `turtle.done()` command tells the turtle that you've finished giving it commands. Without this it will wait for new commands (if run from the shell) or disappear (if run from file).

When playing with the turtle it can be useful to slow it down so that you can see what it going on. The turtle's speed can be set with `turtle.speed(1)` to make it move slowly. Here's the same program with a slower turtle:

```python
import turtle

turtle.speed(1)

turtle.forward(100)
turtle.right(130)
turtle.forward(100)

turtle.done()
```

Problem Solving (with Turtles)

In this part of the session I'll show you how to use the turtle to draw a square with Python. This may not seem impressive, but it's a good stepping stone for learning so really powerful stuff in Python.

Copy this code into a new Python file in PyCharm (you might want to save it in a file named `square.py`):

```python
import turtle

turtle.forward(100)
turtle.right(90)

turtle.forward(100)
turtle.right(90)

turtle.forward(100)
turtle.right(90)

turtle.forward(100)
turtle.right(90)

turtle.done()
```

It should draw a square when you run it.

Let's look at what's going on. A square has **four** sides and an angle of **ninety** degrees. The main part of the program is this:

```python
turtle.forward(100)
turtle.right(90)
```

It draws a line and then turns the turtle `90` degress to the right. These two lines of code are repeated four times to create the square.

Since the same two lines of code are repeated four times, it's a good time to use variables. Here's the program rewritten to use a variable for `side_length` and `angle`:

```python
import turtle

side_length = 200
angle = 90

turtle.forward(side_length)
turtle.right(angle)

turtle.forward(side_length)
turtle.right(angle)

turtle.forward(side_length)
turtle.right(angle)

turtle.forward(side_length)
turtle.right(angle)

turtle.done()
```

Now if I want to change the size or angle of the shape I only need to change it in one place, instead of four places.

The turtle library comes with lots of neat little things for your drawings. With the `turtle.color()`, `turtle.begin_fill()`, and `turtle.end_fill()` functions you can play around the shape's line and fill colours:

```python
import turtle

side_length = 200
angle = 90

turtle.color('red', 'pink')
turtle.begin_fill()

turtle.forward(side_length)
turtle.right(angle)

turtle.forward(side_length)
turtle.right(angle)

turtle.forward(side_length)
turtle.right(angle)

turtle.forward(side_length)
turtle.right(angle)

turtle.end_fill()

turtle.done()
```

In the next exercise you'll take what you've learned about turtle and adapt the above program to create a triangle.

Exercise 1.1: Create a new file called `triangle.py`. Using `turtle` draw a triangle.

A triangle has **three** sides and an angle of **120** degrees

Extension: Make the triangle blue

Much harder extension Make a circle (hint: circles aren't one sided shapes, they are three-hundred and sixty sided shapes)

For Loops

Repetition is a very common thing in programming. In this section you'll learn about a very powerful feature in Python called a *for loop*. A for loop is used to repeat a block of code a certain number of times.

For loops are usually used with something called a list. You'll learn about lists later on in this session.

`for` loop: allows you to repeat a block of code for every item in a list

To write a for loop you need five things:

1. A `for` operator
2. A variable name that stores each list value one at a time
3. An `in` operator
4. A list of values
5. A body (indented four spaces)

Here's an example of a four loop that will print the number `0`, `1`, `2`, `3`, and `4`:

```
for number in range(5):
    print(number)
```

Let's break down what's going on here. Look at the `range(5)` function. This function is used to tell the for loop how many times to repeat. Here I've used the value `5` so the for loop will repeat 5 times.

The `number` part of the for loop works like a variable. The value of the variable is set by the `range()` function and changes each time the loop repeats.

The first time the loop runs the value of `number` is set to `0`. The body of the loop is then run. In this case the body is `print(number)`, which will print the value of the `number` variable (`0`). After the body has completed running the loop starts again.

The second time the loop runs the value of `number` is set to `1`. The body then runs, once again printing the value of `number`. The loops repeats three more times in total, setting the value value of `number` to `2`, `3`, and `4` in the third, fourth, fifth repeats respectively.

After the fifth loop has finished the `range()` function doesn't return any more numbers so the loop stops.

For loops are really useful for repeating code. Notice that in the original code for the square I repeated the same bit of code four times:

```python
import turtle

side_length = 200
angle = 90

turtle.forward(side_length)
turtle.right(angle)

turtle.forward(side_length)
turtle.right(angle)

turtle.forward(side_length)
turtle.right(angle)

turtle.forward(side_length)
turtle.right(angle)

turtle.done()
```

Using a for loop I can simplify the program by using a for loop to do the repetition:

```python
import turtle

side_length = 200
angle = 90

for side in range(4):
    turtle.forward(side_length)
    turtle.right(angle)

turtle.done()
```

This code will draw an identical square. The benefit here is the code is more concide. If the shape became more complex, for example if it had 360 sides instead of 4, the code would look almost the same:

```python
import turtle

side_length = 1
angle = 1

for side in range(360):
    turtle.forward(side_length)
    turtle.right(angle)

turtle.done()
```

All that's change here are the values of `side_length`, `angle`, and the number inside the `range()` function. Now imagine instead that I wanted to write this same program without the for loop. It would be over 700 lines long, making it hard to change and hard to spot errors.

Exercise 1.2: Choose your sides

In this exercise you'll create a program that can draw shapes with any number of sides.

When you run the program it will ask you to input the number of sides that the shape should have. The program will then calculate the correct angle for the shape and draw it for you.

I've started the program for you:

```python
import turtle

sides = int(input('Number of sides: '))

angle = 360 / sides
side_length = 60

# Add the for loop here
turtle.forward(side_length)
turtle.right(angle)

turtle.done()
```

Extension: Create a new file called `spiral.py` and write a program to create a 100 sided spiral with an angle of 90 degrees

---

Lists

So far you've seen three data-types: strings, integers, and floats. You're now going to learn about a new, very special, data-type called a list.

Like a list in the real world, a Python list is a collection of multiple pieces of data kept together. Lists are a special data-type in that they can contain multiple values of other data-types.

List: an ordered collection of values

List are written inside square brackets and separated by commas. Here's a list of integer values that I've named `lottery_values` using a variable:

```
lottery_numbers = [4, 8, 15, 16, 23, 42]
```

The next list has a collection of string values to create a list of students' names:

```
student_names = ['Diedre', 'Hank', 'Helena', 'Salome']
```

Lists can be made up of values of one or more data types. In this list I'm recording my friend's name as a string and their age as an integer:

```
friend = ['Hilda', 27]
```

You'll often want to get values out of a list. List values can be accessed using their **index** in square brackets:

```
student_names = ['Diedre', 'Hank', 'Helena', 'Salome']

print(student_names[2])
```

When this program runs it will output `'Helena'`. The `students[2]` part of the code gets the value in the second index of the list. In this case the value is `'Helena'`.

Why is the value in index 2 `'Helena'` and not `'Hank'`?

List indexes start counting from 0. In the above list the value of `'Diedre'` is at index `0`, `'Hank'` is at index `1`, `'Helena'` at `2`, and `'Salome'` at `3`.

I've formatted the program and added comments to illustrate this clearer:

```
student_names = [
    'Diedre',    # index 0
    'Hank',      # index 1
    'Helena',    # index 2
    'Salome'   # index 3
]

print(student_names[0])
```

There is a reason that list indexes start at `0` instead of `1`. Old computers had a very small memory. To save memory list indexes started at `0` instead of `1`. With modern computers memory isn't as much of an issue, but the tradition of starting list indexes at `0` was still kept around.

For Loops â™¥ Lists

Using lists and for loops together

```
student_names = ['Diedre', 'Hank', 'Helena', 'Salome']

for student_name in student_names:
    print(student_name)
```

Exercise 1.3: I have a load of cats and I want you to create a piece of art with all of their names on it.

You need to use the turtle library to write the cats' names on each corner of a square.

The `turtle.write()` function will write a string using turtle. I've started the code for you, you need to add the for loop:

```
import turtle

cat_names = ['Fluffy', 'Ginger', 'Whiskers', 'Rod']

# Add for loop here

turtle.write(cat_name)
turtle.forward(100)
turtle.right(90)

turtle.done()
```

Functions

Function: A reusable block of code

```
import turtle


def square():
    side_length = 100
    angle = 90

    for side in range(4):
        turtle.forward(side_length)
        turtle.right(angle)
```

All functions have 1. a `def` operator 1. a name 1. brackets 1. a colon 1. body (indented 4 spaces)

If you ran the above code it wouldn't do anything. All you've done is define a function, you actually need to *call* the function to use it.

To call a function you write the function's name followed by brackets like the call to `square()` on the last line of this program:

```
import turtle


def square():
    side_length = 100
    angle = 90

    for side in range(4):
        turtle.forward(side_length)
        turtle.right(angle)


square()
```

Functions can be called many times

```
import turtle


def square():
    side_length = 100
    angle = 90

    for side in range(4):
        turtle.forward(side_length)
        turtle.right(angle)


square()
turtle.forward(150)
square()
```

Exercise 1.4: Create a function that draws a triangle using turtle.

Solution:

```python
import turtle


def triangle():
    side_length = 100
    angle = 120

    for side in range(3):
        turtle.forward(side_length)
        turtle.right(angle)


triangle()
```

Argument: A parameter used to change the behaviour of a function

Arguments go inside the brackets and behave like variables

```python
import turtle


def square(side_length):
    angle = 90

    for side in range(4):
        turtle.forward(side_length)
        turtle.right(angle)

square(60)
square(100)
```

Exercise 1.5: Modify your triangle function so that you can set the **side length** using an argument

Extension: Use a second argument to set the **colour** of the triangle

---

Functions can have multiple arguments seperated by commas

```python
import turtle


def square(side_length, colour):
    angle = 90

    turtle.color(colour, colour)
    turtle.begin_fill()

    for side in range(4):
        turtle.forward(side_length)
        turtle.right(angle)

    turtle.end_fill()

square(400, 'red')
square(300, 'pink')
square(200, 'blue')
square(100, 'yellow')
```

Values can be returned from functions using the `return` operator

```python
def add(num_1, num_2):
    return num_1 + num_2

my_height = 182
friend_height = 160

total_height = add(my_height, friend_height)

print(total_height)
```

Exercise 1.6: Complete the function to return the area of a circle

Use the comments to help you

```
def circle_area():  # add the radius argument inside the 
    area = 3.14 * (radius ** 2)
    # return area here


area =  circle_area(10)

print(area)
```

Recap

This session:

1. Importing libraries
2. Problem solving with Turtle
3. For Loops
4. Lists
5. Functions

If tkinter isn't installed on Linux:

```
sudo apt-get install python3-tk
```

Solutions

Session 1