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.

Sessions

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.

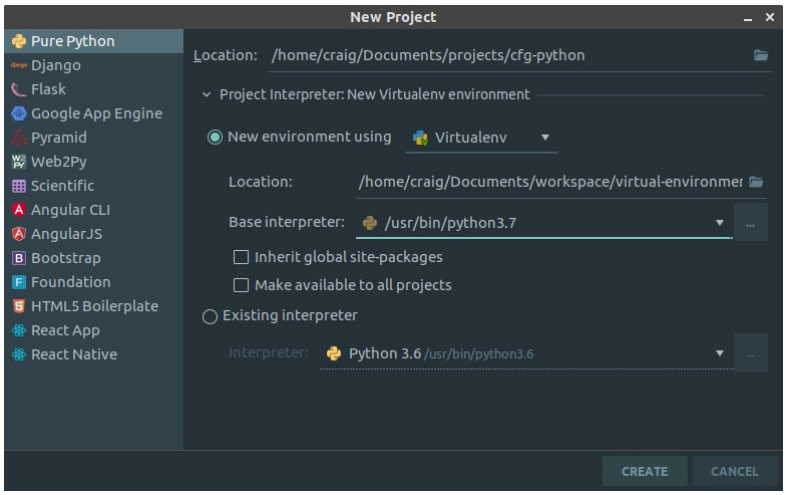
Open PyCharm. You should see a window like the one below:

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`.



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:

```
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:

```
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 one and why?

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

> **Concatentation:**
>
> **Method:**
>
> **Dot Notation:**

`str()`

Variables

Labels

Variables work with all data-types

String Formatting

There a few different ways to do string formatting in Python

[F-STRINGS]

User Input

If tkinter isn't installed on Linux:

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

Session 3.

Comparisons

Conmparisons are all about checking if something is `True` or `False`. This has many applications in programming. Is the password a user has just entered correct? Is there enough money in my account to buy a pair of socks? Can I fit another jelly bean in my mouth?

Notice how all of the above examples are yes or no questions. In Python you can represent a yes as a `True` value and a no as a `False` value.

The `True` and `False` values are both Boolean data types. Named after George Bool, an English Math wizard, Boolean is pronounced like Cool Ian, but with a B.

Like all data types in Python you can use variables to label the values and explain their meaning to other programmers:

```
am_i_hungry = True
is_it_hot = False
```

On thing to note is that the values `True` and `False` should start with a capital letter. If you miss forget to capitalise the value Python will think the value is a variable and you'll get an error like this:

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

Checking Stuff is True or False

A lot of the time you won't know if something is `True` or `False` until you check it. This is where comparators come in.

In their most basic form comparators will compare one value with another, then will return `True` or `False` depending on the result.

Say I have five forks and six friends, and I want to know if I have enough forks to throw a dinner party. To do this I can check whether the number of forks I have is greater than my number of friends:

```
forks = 5
friends = 6

enough_forks = forks >= friends

print('You have enough forks: {}'.format(enough_forks))
```

The output of this program is `You have enough forks: False`

The part of the program that does the comparison is `forks >= friends`. The `>=` bit compares if the value on the left is greater than or equal to the value on the right.

The result of this case is returned and I use the `enough_forks` variable to remember this result. The reason that the result is `False` in this case is because `5` is not greater than or equal to `6`.

I either need more forks or fewer friends.

After a long deliberation I decided to buy one more fork (it was a tought decision):

```
forks = 6
friends = 6

enough_forks = forks >= friends

print('You have enough forks: {}'.format(enough_forks))
```

Now the output of the program is `You have enough forks: True`

The value of `forks` is now `6`, which is equal to the value of the `friends` variable. So when the `forks >= friends` comparison is made, the value of `enough_forks` is set to `True`.

Exercise: Do I have enough eggs to make a really big omlette?

My dinner party is going to be a huge success. My friends will all love how they don't have to share cutlery. I just need to decide what to cook.

To really wow my friends I'm planning to cook a really big omlette. But do I have enough eggs?

I have 17 edible eggs in my fridge. I estimate that I will need three eggs of my friends (remember that I have exactly 6 friends).

```
eggs_in_fridge = 17

friends = 6
required_eggs = friends * 3

enough_eggs =

print('The party omlette has enough eggs: {}'.format(enoug
```

Your task is to finish my program that calculates whether I have enough eggs for my party omlette. To do this you need to check whether `eggs_is_fridge` is greater than or equal to `required_eggs`. The result should be assigned to the `enough_eggs` variable.

Be creative with the wording of the output for extra points. Just makes sure it shows the result as `True` or `False`.

More Ways to Compare Stuff

In the last example I checked whether the number of forks was greater than or equal to the number of friends I have.

Not surprisingly the `>=` comparator in Python is called the `greater than or equal` comparator.

There are other comparators in Python. Let's take a tour of the exciting line-up of comparators that Python has available.

Equal To (==)

First up is the `==` or `equal to` comparator. This nifty little comparator checks whether two values are equal.

Let's say I've built a chair and need to check it has exactly four legs:

```
required_legs = 4
legs_i_made = 5

right_amount_of_legs = required_legs == legs_i_made

print(right_amount_of_legs)
```

The output of this program is `False`. The values `4` and `5` are not the same so the the result of the comparison is `False`. Oh no I have too many legs on my chair. People are going to give me weird looks.

I've hatily cut off one of the legs with a saw:

```
required_legs = 4
legs_i_made = 4

right_amount_of_legs = required_legs == legs_i_made

print(right_amount_of_legs)
```

Now the result is `True` as the values `4` and `4` are the same.

Notice how the `equal to` comparator is two equals signs `==`. This looks similar, but is in fact very different, to when you use a single equals sign `=` to create a variable.

Always remember to use a single equals sign `=` when assigning a value to a variable and a double equals sign `==` when comparing two values.

Exercise: I'm on chair building spree

I'm really into building chairs now. If I don't find a way to stop myself I'll have too many chairs for my friends to sit on. They won't know what to do.

Let's write a program to check I have exactly four chairs.

```
chairs_made = 2
desired_chairs = 6

stop_making_chairs =

print('Stop making chairs right now: {}'.format(stop_makir
```

While I hammer and glue bits of wood, it's your job to finish this program.

You'll need to check `chairs_made` is equal to `desired_chairs`. The result of this comparison should be assigned to the `stop_making_chairs` variable. Only then will I know when to stop making chairs.

Checking Things Are Not the Same (!=)

The `equal to` comparator has an inverse version of itself, known as the `not equal to` comparator.

In short the `not equal to` will evaluate to `False` is the two values being compared are the same. It will be `True` when the two compared values are different. This is the opposite of the `equal to` comparator.

The `equal to` and `not equal to` comparators can be used to compare strings and number (integers and floats) data types.

Here I'm checking that the film my friend wants to watch is not The Shape of Water (I can't watch it with other people, I cry too much at the end)

```
friends_film_choice = 'The Shape of Water'
film_that_makes_me_cry = 'The Shape of Water'

no_crying = friends_film_choice != film_that_makes_me_cry

print('You will not cry at this film: {}'.format(no_crying
```

Uh-oh, looks like they've chosen the film that will make me cry so the output is `You will not cry at this film: False`

One thing to note is that when comparing strings the comparators are really strict. A single character difference will change the result of the comparison. For example if I only have a capital letter difference between the two strings, they will be compared as totally different:

```
friends_film_choice = 'the shape of water'
film_that_makes_me_cry = 'The Shape of Water'

no_crying = friends_film_choice != film_that_makes_me_cry

print('You will not cry at this film: {}'.format(no_crying
```

The output will be `You will not cry at this film: True`, which is strange because as humans we know they're the same thing. Python is strict in these comparisons because in most cases it does actually matter.

There is a way around this. If you remember back to the first session you learned how to change the case of a string with the `.lower()` method. Using this method you can force both of the strings to be in lower-case, making the comparison consistent with what we expect:

```
friends_film_choice = 'the shape of water'.lower()
film_that_makes_me_cry = 'The Shape of Water'.lower()

no_crying = friends_film_choice != film_that_makes_me_cry

print('You will not cry at this film: {}'.format(no_crying
```

The output is back to what I expected `You will not cry at this film: False`. That's that sorted. Now I can focus on crying at an artful story of fish-man romance.

Exercise: Is the omlette safe to eat?

So that I don't give my friends food poisoning I need to check that the omlette is safe to eat. How am I going to do that? Well, like all of my other problems, with a Python program!

Using my in-depth knowledge of food science, I have determined that a omlette is safe to eat if it is not runny.

I started writing the program, but then the post man arrived to deliver my eBay purchase of a single spoon. He's what I got so far:

```
is_runny = input('Is the omlette runny? (y/n)')

is_edible =
```

I want you to finish my omlette safety checking program. Don't worry, I trust you. Te worst that could happen is my friends never stop telling the story of how I served them plates of raw eggs at a dinner party.

To finish my program you need to use the `not equal` comparison. Use this to check the value of `is_runny` is not equal to `'y'`. The result of this comparison should be assigned to the `is_edible` variable.

You'll also need to add output in this format `The omlette is safe to eat: True`.

Let's Cover Greater, Less and Their Variants at the Same Time

Our tour of comparators is realy picking up pace. We're looping back on ourselves and covering the `greater than or equal to` comparator and its close siblings.

The `greater than or equal to` comparator has a twin. Their parents named it `less than or equal to` and it looks like `<=`. Like all other comparators it compares two values. In this case is checks whether the value on the left is smaller than or the same as the one on the right.

I'm like average levels of tall. It's only a problem when I'm visiting really small houses and Hobbiton.

Let's see if I can fit through a door:

```
my_height = 182.1
door_height = 190

can_fit = my_height <= door_height

print('I can walk through this door: {}'.format(can_fit))
```

The output is `I can walk through this door: True`. Nice. No bending over for me.

Oh, wait, here comes another door!

```
my_height = 182.1
door_height = 182.0

can_fit = my_height <= door_height

print('I can walk through this door: {}'.format(can_fit))
```

This time the output is `I can walk through this door: False`. Ouch.

There are another two comparators that you need to know about. The `less than` and `greater than` comparators. They're just like the `less than or equal to` and `greater than or equal to` comparators, but they won't return `True` if the compared values are the same.

The `greater than` comparator looks like this `>` and the `less than` comparator looks like this `<`. You might get direction of these mixed up from time to time. I found it was helpful to image that the symbol is a shark's mouth and it wants to eat the larger number.

The final thing to note about the four comparators (`>`, `<`, `>=` and `<=`) is that they work with number data types (integers and floats), but don't work with strings. It's easy for Python to know if `5` is greater than `6`, but not so much for Python to know if `"Air freshener"` is greater than `"Tomato"`.

Exercise: How good was the party out of 5 stars

I've spent months preparing for this dinner party. I've counted numerous eggs multiple times, poured my blood sweat and tears into chairs that are definitely safe to sit on (don't let anyone tell you otherwise), and invested in a new spoon.

When the night is over I want to check that all my hard work paid off. That's right, I'm going to send my friends a survey.

Like all good reviews my dinner party will be rated using 1 to 5 stars. After I've received the ratings I will be heart broken if I receive 3 or less stars.

It's your job to write the program that will tell me that my friends rated my party more than 3 stars. Here you go. I've started it for you.

```
friends_rating = 4
three_stars = 3

good_rating =
```

Choose a comparator to decide how you want to compare my `friends_rating` to `three_stars`. Remember a score of `3` or lower is bad, whereas a score of more than `3` is good.

Make sure you output the result with a sentence.

Checking Something is in a List

Our final stop on our tour of comparators is a special one. It's the `in` comparator.

This one is special in that it can check whether one value is in a list of other values.

I have a bunch of things in my backpack. I need to check if I have a pen in there:

```
backpack = ['laptop', 'keys', 'pen', 'notebook']

has_pen = 'pen' in backpack

print('I have a pen: {}'.format(has_pen))
```

There is a value `'pen'` in the `backpack` list, so the output is `I have a pen: True`

The `in` comparator can also check if one string is inside another string.

I want to check if a name has the letter `'e'` in it:

```
name = 'Francis Bacon'

has_e = 'e' in name

print('{} has an e in it: {}'.format(name, has_e))
```

The output for this is `Francis Bacon has an e in it: False`. There is no letter `'e'` in `'Francis Bacon'` so the result is `False`.

Exercise: Guest List

My dinner parties have become very popular. People line up around the block to eat my omlette and use one of my six forks.

To restrict who comes into my dinner parties, I need to use a guest list.

I've started writing a program to check the guest list:

```
name = raw_input('Please enter name: ')

guests = ['Sarah', 'Joan', 'James', 'Niamh']

on_the_list =
```

You need to finish the guest list program. Using the value in the `name` variable, check that it's in the `guests` list. Put the result in `on_the_list` variable and print the result.

Comparing Multiple Things and Checking Opposites

Comparators compare two things. Is one thing greater than another? Are these two values the same?

Quite often you'll want to combine multiple comparisons together. For example a friend asks me if I want to see a film at the cinema next Wednesday. I would want to check two things: Am I available on Wednesday; and have I already seen the film?

Boolean operators in Python allow you to join multiple Boolean values together and return a single result. The `and` and `or` Boolean operators are used for this, which you will see in a moment.

There is also a third Boolean operator, `not`, which behaves differently. It swaps `True` for `False` values and `False` for `True` values. You'll also see this in just a moment.

and

The `and` operator is used to combine two Boolean values into a single result.

I'm about to board a plane flight for my holidays. To get on the flight I need two things: my ticket and my passport.

```
has_ticket = True
has_passport = True

can_fly = has_ticket and has_passport

print('Can fly: {}'.format(can_fly))
```

The `and` operator goes between two values. In the above example it goes between `has_ticket` and `has_passport`. The result of this

depends on whether or not both values are `True`.

In this example both `has_ticket` and `has_passport` are `True` so the result will be `True`. If either or both of these values were `False` then the result would be `False`.

For example if I forgot my passport then I wouldn't be allowed on my flight:

```
has_ticket = True
has_passport = False

can_fly = has_ticket and has_passport

print('Can fly: {}'.format(can_fly))
```

As the `has_passport` value is `False` the result is `Can fly: False`. Even though the `has_ticket` value is `True`, when it is combined with the `False` value of `has_passport`, the final result is `False`.

You can see the different results when you combine different Boolean values with the `and` operator in the following table:

Values	Result
True and True	True
True and False	False
False and True	False
False and False	False

The comparators that you covered earlier all return a boolean value, which means they can be combined with Boolean operators.

I want to ride a roller coaster while I'm on holiday. The roller coaster has a minimum height check and costs $2.50 per ride. I need to check I'm tall enough and have enough money.

```
cost = 2.5
min_height = 152

my_money = 3.22
my_height = 182

can_ride = my_money >= cost and my_height >= min_height

print('Can ride the roller coaster: {}'.format(can_ride))
```

When Python runs this line `can_ride = my_money >= cost and my_height >= min_height` it does it in several steps. First it runs `my_money >= cost`, which in this case has the values `3.22 >= 2.5`.

The result is `True`.

Next it checks `my_height >= min_height`. Substituting in the values you can see `182 >= 152` also results in `True`.

Python then sees that there is an `and` operator and know check if both results are `True`. In this case both results are `True` so the final result is `True`, which is assigned to the `can_ride` variable.

or

The `or` Boolean operator combines two Boolean values into a single result. If at least one of the Boolean values is `True` then the final result will be `True`. The only time that the result will be `False` is if both values are `False`.

I imagine you're trying to enter an ice-cream shop to buy some delicious and well priced ice-cream. The shop has two doors. To get into the shop only one of the doors needs to be open. If either of the doors or both of the doors is open you can get in.

```
door_1_open = True
door_2_open = False

is_ice_cream_shop_open = door_1_open or door_2_open

print('I can have some ice-cream {}'.format(is_ice_cream_s
```

Because door 1 is open, I can get ice-cream. Even though the second door is locked, I can still get into the building for an icy treat.

You can see the result of the different combinations for Boolean values with the `or` operator in the following table:

Values	Result
True or True	True
True or False	True
False or True	True
False or False	False

Just like with the `and` operator, the `or` operator can be used with comparators.

To summarise, the `and` Boolean operator will result in `True` only if **all** values are `True`. The `or` operator will result in `True` if *any* values are `True`.

not

Values	Result
not True	False

```
not False True
```

Putting It All Together

Comparators and Boolean operators

Brackets

Keeping It Clean

When using operators and comparators together you can actually do a lot in a single line:

```
valid_qty =  flowers_ordered > 0 and flowers_ordered <= 10
print('Valid qty of flowers {}'.format(valid_qty))
```

Any idea what the result of that is? No? Me neither. I just wrote that and it takes me quite a while to understand what's going on.

This is bad.

As you progress on your journey as a developer you'll find that being able to work quickly and without simple mistakes is linked to how readable the code is.

Breaking down complex statements like the one above into smaller parts is one way to make the code more descriptive and easier to understand:

```
flowers_ordered = 12

at_least_one_flower =  flowers_ordered > 0
no_more_than_100_flowers = flowers_ordered <= 100

in_multiples_of_five = flowers_ordered % 5 == 0
in_multiples_of_twelve = flowers_ordered % 12 == 0

not_too_many_or_few = at_least_one_flower and no_more_thar
correct_multiples = in_multiples_of_five or in_multiples_c

valid_qty = not_too_many_or_few and correct_multiples

print('Valid qty of flowers {}'.format(valid_qty))
```

Now each part has a named variable that clearly explains the purpose

of each comparison before they're all joined into the final result.

I've managed to work out that I need to buy more than `0` flowers and `100` or less flowers. I also need to buy the flowers in multiples of `5` or `12`.

Although there are more lines, the purpose of each one is clearly explained by the name of the variable. Although it takes me slightly longer to read, it is much faster for me to understand what it is doing and why it is doing it.

We can take this one step further by putting it all together in a function.

```
def valid_qty(flowers_ordered):
    at_least_one_flower =  flowers_ordered > 0
    no_more_than_100_flowers = flowers_ordered <= 100

    in_multiples_of_five = flowers_ordered % 5 == 0
    in_multiples_of_twelve = flowers_ordered % 12 == 0

    not_too_many_or_few = at_least_one_flower and no_more_
    correct_multiples = in_multiples_of_five or in_multipl

    return not_too_many_or_few and correct_multiples


is_valid = valid_qty(12)

print('Valid qty of flowers {}'.format(is_valid))
```

Now all of my code is in a function it can be reused. It also makes the other parts of my code cleaner as the function name explains what it does concisely. It also means the other parts of the code don't need to know how to do this calculation, so the complexity can be kept hidden away from the code that uses the function.

If Statements

Truthy and Falsey

Problem Solving Exercises